International Workshop on Foundational and Practical Aspects of Resource Analysis FOPARA '09

3rd of November
Eindhoven, The Netherlands
A satellite event
of the 16th International Symposium on Formal Methods

Marko van Eekelen and Olha Shkaravska

www.aha.cs.ru.nl/fopara

Workshop organiser:
Computing Science Department of Radboud University Nijmegen

Preface

The FOPARA[1] workshop serves as a forum for presenting original research results that are relevant to the analysis of resource (time, space) consumption by computer programs. FOPARA aims to bring together the researchers that work on foundational issues with the researchers that focus more on practical results. Therefore, both theoretical and practical contributions have been encouraged. The contributions cover the following topics: resource analysis for embedded systems, logical and machine-independent characterisations of complexity classes, logics closely related to complexity classes, type systems for controlling complexity, semantic methods to analyse resources, incl. quasi- and sup-interpretations, practical applications of resource analysis, etc. After the workshop the program committee will select papers for final proceedings. These final proceedings will be published by Springer in a volume of the Lecture Notes in Computer Science series.

Up to now, a few similar events have taken place. In 2006 and 2008 application-oriented resource analysis workshops (an EmBounded Open Workshop in Budapest, 2006, and a Resource Analysis Workshop in Hertfordshire, 2008) were held as affiliated events of International Symposium on the Implementation and Application of Functional Languages (IFL) . Participators of these workshops were the University of St. Andrew (UK), Heriot-Watt University of Edinburgh (UK), Ludwig-Maximilians University of Munich (Germany), University Complutense of Madrid (Spain) and the Polytechnical University of Madrid (Spain). Another group of researchers were active in the series of workshops on Implicit Computational Complexity, see, for instance, WICC'08 in Paris. That series gathers researchers working in theoretical foundations of resource analysis, mainly from in France (Universities of Paris Diderot and Paris Nord, LORIA Nancy), Italy (Universities of Bologna and Turin), Norway, Germany and Portugal.

FOPARA aims to bring together these various directions in resource analysis addressing a larger resource analysis community. FOPARA wants to enable fruitful exchange of ideas between more practical and more theoretical groups. We are happy that FOPARA has indeed a program with a significant number of contributions from both the more practically oriented and the more theoretically oriented research groups. We wish everyone a very inspiring workshop. We say to all of you: "Have a nice FOPARA!".

October 2009

Marko van Eekelen, Program Chair
Olha Shkaravska, Program Co-Chair
FOPARA'09

[1] The FOPARA workshop is partly funded by the AHA project which is sponsored by the Netherlands Organisation for Scientific Research (NWO) under grant nr. 612.063.511 and the EU Artemis funded CHARTER project (nr. 100039).

Program Committee

Marko van Eekelen	Radboud University and Open University, NL, PC chair
Olha Shkaravska	Radboud University, NL, PC co-chair
Patrick Baillot	ENS-Lyon, France
Armelle Bonenfant	IRIT, France
Ugo Dal Lago	University of Bologna, Italy
Kevin Hammond	University of St. Andrews, UK
Martin Hofmann	LMU, Munich, Germany
Thomas Jensen	IRISA, Rennes, France
Tamas Kozsik	Eötvös Loránd University of Budapest, Hungary
Hans-Wolfgang Loidl	LMU, Munich, Germany
Kenneth MacKenzie	University of Edinburgh, UK
Jean-Yves Marion	Loria, Nancy, France
Greg Michaelson	Heriot-Watt University, Edinburgh, UK
Ricardo Peña	University Complutense Madrid, Spain
German Puebla	Politechnical University of Madrid, Spain
Luca Roversi	University of Turin, Italy
Phil Trinder	Heriot-Watt University, Edinburgh, UK

Program and the table of content

The Reachability-Bound Problem

Sumit Gulwani, Microsoft Research. Invited talk.

The "reachability-bound problem" is the problem of finding a symbolic worst-case bound on the number of times a given control location inside a procedure is visited in terms of the inputs to that procedure. This has applications in bounding resources consumed by a program such as time, memory, network-traffic, power, as well as estimating quantitative properties (as opposed to boolean properties) of data in programs, such as amount of information leakage or uncertainty propagation.

Our approach to solving the reachability-bound problem brings together two very different techniques for reasoning about loops in an effective manner. One of these techniques is an abstract-interpretation based iterative technique for computing precise disjunctive invariants (to summarize nested loops). The other technique is a non-iterative proof-rules based technique (for loop bound computation) that takes over the role of doing inductive reasoning, while deriving its power from use of SMT solvers to reason about abstract loop-free fragments.

We have implemented our solution to the reachability-bound problem in a tool called SPEED, which computes symbolic computational complexity bounds for procedures in .Net code-bases.

A Space Consumption Analysis By Abstract Interpretation ⋆

Manuel Montenegro, Ricardo Peña and Clara Segura
`montenegro@fdi.ucm.es` `{ricardo,csegura}@sip.ucm.es`

Universidad Complutense de Madrid, Spain

Abstract. *Safe* is a first-order functional language with an implicit region-based memory system and explicit destruction of heap cells. Its static analysis for inferring regions, and a type system guaranteeing the absence of dangling pointers have been presented elsewhere.
In this paper we present a new analysis aimed at inferring upper bounds for heap and stack consumption. It is based on abstract interpretation, being the abstract domain the set of all n-ary monotonic functions from real non-negative numbers to a real non-negative result. This domain turns out to be a complete lattice under the usual $\sqsubseteq$ relation on functions. Our interpretation is monotonic in this domain and the solution we seek is the least fixpoint of the interpretation.
We first explain the abstract domain and some correctness properties of the interpretation rules with respect to the language semantics, then present the inference algorithms for recursive functions, and finally illustrate the approach with the upper bounds obtained by our implementation for some case studies.

1 Introduction

The first-order functional language *Safe* has been developed in the last few years as a research platform for analysing and formally certifying two properties of programs related to memory management: absence of dangling pointers and having an upper bound to memory consumption. Two features make *Safe* different from conventional functional languages: (a) a region based memory management system which does not need a garbage collector; and (b) a programmer may ask for explicit destruction of memory cells, so that they could be reused by the program. These characteristics, together with the above certified properties, make *Safe* useful for programming small devices where memory requirements are rather strict and where garbage collectors are a burden in service availability.

The *Safe* compiler is equipped with a battery of static analyses which infer such properties [12, 13, 10]. These analyses are carried out on an intermediate language called *Core-Safe* explained below. We have developed a resource-aware operational semantics of *Core-Safe* [11] producing not only values but also exact figures on the heap and stack consumption of a particular running. The code generation phases have been certified in a proof assistant [5, 4], so that there is

⋆ Work partially funded by the projects TIN2008-06622-C03-01/TIN (STAMP), S-0505/ TIC/ 0407 (PROMESAS) and the MEC FPU grant AP2006-02154.

a formal guarantee that the object code actually executed in the target machine (the JVM [9]) will exactly consume the figures predicted by the semantics.

Regions are dynamically allocated and deallocated. The compiler 'knows' which data lives in each region. Thanks to that, it can compute an upper bound to the space consumption of every region and so and upper bound to the total heap consumption. Adding to this a stack consumption analysis would result in having an upper bound to the total memory needs of a program.

In this work we present a static analysis aimed at inferring upper bounds for individual *Safe* functions, for expressions, and for the whole program. These have the form of n-ary mathematical functions relating the input argument sizes to the heap and stack consumption made by a *Safe* function, and include as particular cases multivariate polynomials of any degree. Given the complexity of the inference problem, even for a first-order language like *Safe*, we have identified three separate aspects which can be independently studied and solved: (1) Having an upper bound on the size of the call-tree deployed at runtime by each recursive *Safe* function; (2) Having upper bounds on the sizes of all the expressions of a recursive *Safe* function. These are defined as the number of cells needed by the normal form of the expression; and (3) Given the above, having an inference algorithm to get upper bounds for the stack and heap consumption of a recursive *Safe* function.

Several approaches to solve (1) and (2) have been proposed in the literature (see the Related Work section). We have obtained promising results for them by using rewriting systems termination proofs [10]. In case of success, these tools return multivariate polynomials of any degree as solutions. This work presents a possible solution to (3) by using abstract interpretation. It should be considered as a *proof-of-concept* paper: we investigate how good the upper bounds obtained by the approach are, provided we have the best possible solutions for problems (1) and (2). In the case studies presented below, we have introduced by hand the bounds to the call-tree and to the expression sizes.

The abstract domain is the set of all monotonic, non-negative, n-ary functions having real number arguments and real number result. This infinite domain is a complete lattice, and the interpretation is monotonic in the domain. So, fixpoints are the solutions we seek for the memory needs of a recursive *Safe* function. An interesting feature of our interpretation is that we usually start with an over-approximation of the fixpoint, but we can obtain tighter and tighter safe upper bounds just by iterating the interpretation any desired number of times.

The plan of the paper is as follows: Section 2 gives a brief description of our language; Section 3 introduces the abstract domain; Sections 4 and 5 give the abstract interpretation rules and some proof sketches about their correctness, while Section 6 is devoted to our inference algorithms for recursive functions; in Section 7 we apply them to some case studies, and finally in Section 8 we give some account on related and future work.

2 Safe in a Nutshell

Safe is polymorphic and has a syntax similar to that of (first-order) Haskell. In *Full-Safe* in which programs are written, regions are implicit. These are inferred when *Full-Safe* is desugared into *Core-Safe* [13]. The allocation and deallocation of regions is bound to function calls: a *working region* called *self* is allocated when entering the call and deallocated when exiting it. So, at any execution point only a small number of regions, kept in an invocation stack, are alive. The data structures built at *self* will die at function termination, as the following treesort algorithm shows:

```
treesort xs = inorder (mkTree xs)
```

First, the original list `xs` is used to build a search tree by applying function `mkTree` (not shown). The tree is traversed in inorder to produce the sorted list. The tree is not part of the result of the function, so it will be built in the working region and will die when the `treesort` function returns. The *Core-Safe* version of `treesort` showing the inferred type and regions is the following:

```
treesort :: [a] @ rho1 -> rho2 -> [a] @ rho2
treesort xs @ r = let t = mkTree xs @ self
                  in inorder t @ r
```

Variable `r` of type `rho2` is an additional argument in which `treesort` receives the region where the output list should be built. This is passed to the `inorder` function. However `self` is passed to `mkTree` to instruct it that the intermediate tree should be built in `treesort`'s *self* region.

Data structures can also be destroyed by using a destructive pattern matching, denoted by !, or by a **case!** expression, which deallocates the cell corresponding to the outermost constructor. Using recursion, the recursive portions of the whole data structure may be deallocated. As an example, we show a *Full-Safe* insertion function in an ordered list, which reuses the argument list's spine:

```
insertD x []! = x : []
insertD x (y:ys)! | x <= y = x : y : ys!
                  | x > y  = y : insertD x ys!
```

Expression $ys!$ means that the substructure pointed to by ys in the heap is reused. The following is the (abbreviated) *Core-Safe* typed version:

```
insertD :: Int -> [Int]! @ rho -> rho -> [Int] @ rho
insertD x ys @ r = case! ys of
                    []   -> let zs = [] @ r in let us = (x:zs) @ r in us
                    y:yy -> let b = x <= y in case b of
                      True ->  let ys1 = (let yy1 = yy! in let as = (y:yy1) @ r in as) in
                               let rs1 = (x:ys1) @ r in rs1
                      False -> let ys2 = (let yy2 = yy! in insertD x yy2 @ r) in
                               let rs2 = (y:ys2) @ r in rs2
```

This function will run in constant heap space since, at each call, a cell is destroyed while a new one is allocated at region `r` by the (:) constructor. Only when the new element finds its place a new cell is allocated in the heap.

In Fig. 1 we show two *Core-Safe* big-step semantic rules in which a resource vector is obtained as a side effect of evaluating an expression. A judgement has the form $E \vdash h, k, td, e \Downarrow h', k, v, (\delta, m, s)$ meaning that expression e is evaluated in an environment E using the td topmost positions in the stack, and in a heap

$$\frac{\begin{array}{c}E \vdash h, k, 0, e_1 \Downarrow h', k, v_1, (\delta_1, m_1, s_1) \\ E \uplus [x_1 \mapsto v_1] \vdash h', k, td+1, e_2 \Downarrow h'', k, v, (\delta_2, m_2, s_2)\end{array}}{E \vdash h, k, td, \mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_2 \Downarrow h'', k, v, (\delta_1 + \delta_2, \max\{m_1, |\delta_1| + m_2\}, \max\{2 + s_1, 1 + s_2\})}\ [\mathit{Let}_1]$$

$$\frac{E\ x = p \quad C = C_r \quad E \uplus [\overline{x_{r_i} \mapsto v_i}^{n_r}] \vdash h, k, td + n_r, e_r \Downarrow h', k, v, (\delta, m, s)}{E \vdash h \uplus [p \mapsto (j, C\ \overline{v_i}^{n})], k, td, \mathbf{case!}\ x\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^{n} \Downarrow h', k, v, (\delta + [j \mapsto -1], \max\{0, m-1\}, s + n_r)}\ [\mathit{Case!}]$$

Fig. 1. Two rules of the resource-aware operational semantics of *Safe*

(h, k) with $0..k$ active regions. As a result, a heap (h', k) and a value v are obtained, and a resource vector (δ, m, s) is consumed. A heap h is a mapping between pointers and constructor cells $(j, C\ \overline{v_i}^{n})$, where j is the cell region. The first component of the resource vector is a partial function $\delta : \mathbb{N} \to \mathbb{Z}$ giving for each active region i the signed difference between the cells in the final and initial heaps. A positive difference means that new cells have been created in this region. A negative one, means that some cells have been destroyed. By $dom(\delta)$ we denote the subset of $\mathbb{N}$ in which δ is defined. By $|\delta|$ we mean the sum $\sum_{n \in dom(\delta)} \delta(n)$ giving the total balance of cells. The remaining components m and s respectively give the *minimum* number of fresh cells in the heap and of words in the stack needed to successfully evaluate e. When e is the main expression, these figures give us the total memory needs of a particular run of the *Safe* program. For a full description of the semantics and the abstract machine see [11].

3 Function Signatures

A *Core-Safe* function is defined as a $n + m$ argument expression:

$$\begin{array}{l} f :: t_1 \to \ldots t_n \to \rho_1 \to \ldots \rho_m \to t \\ f\ x_1 \cdots x_n\ @\ r_1\ \cdots r_m = e_f \end{array}$$

A function may charge space costs to heap regions and to the stack. In general, these costs depend on the *sizes* of the function arguments, where the size of a term of an algebraic type is the number of cells of its recursive spine, the size of a natural number is its value, and the size of a boolean value is zero. For example,

```
copy xs @ r = case xs of []   -> [] @ r
                         y:ys -> let zs = copy ys @ r in
                                 let rs = (x:zs) @ r in rs
```

charges as many cells to region r as the length of its input list.

As a consequence, all the space costs and needs of f can be expressed as n-ary functions $\eta : (\mathbb{R}^+ \cup \{+\infty\})^n \to \mathbb{R} \cup \{+\infty, -\infty\}$. Infinite costs will be used to represent that we are not able to infer a bound (either because it does not exist or because the analysis is not capable). Costs can be negative if the function destroys more cells than it builds. Currently we are restricting ourselves to functions where for each destructed cell at least a new cell is built in the same region. This covers many interesting functions where the aim of cell destruction is space reuse instead of pure destruction, e.g. function `insertD` shown in the previous section. This restriction means that the domain of the space cost functions is the following:

$$\mathbb{F} = \{\eta : (\mathbb{R}^+ \cup \{+\infty\})^n \to \mathbb{R}^+ \cup \{+\infty\} \;\mid\; \eta \text{ is monotonic}\}$$

The domain $(\mathbb{F}, \sqsubseteq, \bot, \top, \sqcup, \sqcap)$ is a complete lattice, where $\sqsubseteq$ is the usual order between functions, and the rest of components are standard. Notice that it is closed by the operations $\{+, \sqcup, *\}$. By abuse of notation we abbreviate $\lambda\overline{x_i}^n.c$ by just c, where $c \in \mathbb{R}^+$.

Function f above may charge space costs to a maximum of $n{+}m{+}1$ regions: It may destroy cells in the regions where $x_1 \ldots x_n$ live; it may create/destroy cells in any output region $r_1 \ldots r_m$, and additionally in its *self* region. Each region r has a region type ρ. We denote by R^f_{in} the set of input region types, and by R^f_{out} the set of output region types. For example, $R^{treesort}_{in} = \{\rho_1\}$ and $R^{treesort}_{out} = \{\rho_2\}$. Looked from outside, the charges to the *self* region are not visible, as this region disappears when the function returns.

Summarising, let $R_f = R^f_{in} \cup R^f_{out}$. Then $\mathbb{D} = \{\Delta : R_f \to \mathbb{F}\}$ is the complete lattice of functions that describe the space costs charged by f to every visible region. In the following we will call abstract heaps to the functions $\Delta \in \mathbb{D}$.

Definition 1. *A function* signature *for f is a triple $(\Delta_f, \mu_f, \sigma_f)$, where Δ_f belongs to $\mathbb{D}$, and μ_f, σ_f belong to $\mathbb{F}$.*

The aim is that Δ_f describes (an upper bound to) the space costs charged by f to every visible region, and μ_f, σ_f respectively describe (an upper bound to) the heap and stack *needs* of f in order to execute it without running out of space. By $[\,]_f$ we denote the empty function $\lambda\rho.\lambda\overline{x_i}^n.0$, where we assume $\rho \in R_f$. By $|\Delta|$ we mean $\sum_{\rho \in dom(\Delta)} \Delta\ \rho$.

4 Abstract Interpretation

In Figure 2 we show the abstract interpretation rules for the most relevant *Core-Safe* expressions. There, an atom a represents either a variable x or a constant c, and $|e|$ denotes the function obtained by the size analysis for expression e. We can assume that the abstract syntax tree is decorated with such information.

When inferring an expression e, we assume it belongs to the body of a function definition $f\ \overline{x_i}^n\ @\ \overline{r_j}^m = e_f$, that we will call the *context* function, and that only already inferred functions $g\ \overline{y_i}^l\ @\ \overline{r_j}^q = e_g$ are called. Let Σ be a global environment giving, for each *Safe* function g in scope, its signature $(\Delta_g, \mu_g, \sigma_g)$, let Γ be a typing environment containing the types of all the variables appearing in e_f, and let td be a natural number. The abstract interpretation $[\![e]\!]\ \Sigma\ \Gamma\ td$ gives a triple (Δ, μ, σ) representing the space costs and needs of expression e. The statically determined value td occurring as an argument of the interpretation and used in rule *App* is the size of the top part of the environment used when compiling the expression $g\ \overline{a_i}^l\ @\ \overline{r_j}^q$. This size is also an argument of the operational semantics. See [11] for more details.

Rules [*Atom*] and [*Primop*] exactly reflect the corresponding resource-aware semantic rules [11]. When a function application $g\ \overline{a_i}^l\ @\ \overline{r_j}^q$ is found, its signature

$$[\![a]\!]\ \Sigma\ \Gamma\ td = ([\,]_f, 0, 1) \quad [Atom]$$

$$[\![a_1 \oplus a_2]\!]\ \Sigma\ \Gamma\ td = ([\,]_f, 0, 2) \quad [Primop]$$

$$\frac{\begin{array}{l}\Sigma\ g = (\Delta_g, \mu_g, \sigma_g) \qquad \theta = unify\ \Gamma\ g\ \overline{a_i}^l\ \overline{r_j}^q \\ \mu = \lambda \overline{x}^n.\mu_g\ (\overline{|a_i|\ \overline{x}^n}^l) \quad \sigma = \lambda \overline{x}^n.\sigma_g\ (\overline{|a_i|\ \overline{x}^n}^l) \quad \Delta = \theta \downarrow_{\overline{|a_i|\ \overline{x}^n}^l} \Delta_g\end{array}}{[\![g\ \overline{a_i}^l\ @\ \overline{r_j}^q]\!]\ \Sigma\ \Gamma\ td = (\Delta, \mu, \sqcup\{l+q, \sigma - \ td + l + q\})}\ [App]$$

$$\frac{[\![e_1]\!]\ \Sigma\ \Gamma\ 0 = (\Delta_1, \mu_1, \sigma_1) \quad [\![e_2]\!]\ \Sigma\ \Gamma\ (td+1) = (\Delta_2, \mu_2, \sigma_2)}{[\![\mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_2]\!]\ \Sigma\ \Gamma\ td = (\Delta_1 + \Delta_2, \sqcup\{\mu_1, |\Delta_1| + \mu_2\}, \sqcup\{2+\sigma_1, 1+\sigma_2\})}\ [Let_1]$$

$$\frac{\Gamma\ r = \rho \quad [\![e_2]\!]\ \Sigma\ \Gamma\ (td+1) = (\Delta, \mu, \sigma)}{[\![\mathbf{let}\ x_1 = C\ \overline{a_i}^n\ @\ r\ \mathbf{in}\ e_2]\!]\ \Sigma\ \Gamma\ td = (\Delta + [\rho \mapsto 1], \mu+1, \sigma+1)}\ [Let_2]$$

$$\frac{(\forall i)\ [\![e_i]\!]\ \Sigma\ \Gamma\ (td+n_i) = (\Delta_i, \mu_i, \sigma_i)}{[\![\mathbf{case}\ x\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^n]\!]\ \Sigma\ \Gamma\ td = (\bigsqcup_{i=1}^n \Delta_i, \bigsqcup_{i=1}^n \mu_i, \bigsqcup_{i=1}^n (\sigma_i + n_i))}\ [Case]$$

$$\frac{\Gamma\ x = T\overline{t_k}^l @\rho \quad (\forall i)\ [\![e_i]\!]\ \Sigma\ \Gamma\ (td+n_i) = (\Delta_i, \mu_i, \sigma_i)}{[\![\mathbf{case!}\ x\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^n]\!]\ \Sigma\ \Gamma\ td = ([\rho \mapsto -1] + \bigsqcup_{i=1}^n \Delta_i, \sqcup(0, \bigsqcup_{i=1}^n \mu_i - 1), \bigsqcup_{i=1}^n (\sigma_i + n_i))}\ [Case!]$$

Fig. 2. Space inference rules for expressions with non-recursive applications

$\Sigma\ g$ is applied to the sizes of the actual arguments, $\overline{|a_i|\ \overline{x_j}^n}^l$ which have the $\overline{x}^n$ as free variables. Due to the application, some different region types of g may instantiate to the same actual region type of f. That means that we must accumulate the memory consumed in some formal regions of g in order to get the charge to an actual region of f. In Figure 2, *unify* $\Gamma\ g\ \overline{a_i}^l\ \overline{r_j}^q$ computes a substitution θ from g's region types to f's region types. If $\theta\ \rho_g = \rho_f$, this means that the generic g's region type ρ_g is instantiated to the f's actual region type ρ_f. Formally, if $R_g = R^g_{in} \cup R^g_{out}$ then $\theta :: R_g \to R_f \cup \{\rho_{self}\}$ is total. The extension of region substitutions to types is straightforward.

Definition 2. *Given a type environment Γ, a function g and the sequences $\overline{a_i}^l$ and $\overline{r_j}^q$, we say that $\theta = unify\ \Gamma\ g\ \overline{a_i}^l\ \overline{r_j}^q$ iff*

$$\Gamma\ g = \forall \overline{\alpha}.\overline{t_i}^l \to \overline{\rho_j}^q \to t \text{ and } \forall i \in \{1 \ldots l\}.\theta\ t_i = \Gamma\ a_i \text{ and } \forall j \in \{1 \ldots q\}.\theta\ \rho_j = \Gamma\ r_j$$

As an example, let us assume $g :: ([a]@\rho^g_1, [[b]@\rho^g_2]@\rho^g_1)@\rho^g_3 \to \rho^g_2 \to \rho^g_4 \to \rho^g_5 \to t$ and consider the application $g\ p\ @\ r_2\ r_1\ r_1$ where $p :: ([a]@\rho^f_1, [[b]@\rho^f_2]@\rho^f_1)@\rho^f_1$, $r_1 :: \rho^f_1$ and $r_2 :: \rho^f_2$. The resulting substitution would be:

$$\theta = [\rho^g_1 \mapsto \rho^f_1, \rho^g_2 \mapsto \rho^f_2, \rho^g_3 \mapsto \rho^f_1, \rho^g_4 \mapsto \rho^f_1, \rho^g_5 \mapsto \rho^f_1]$$

The function $\theta \downarrow_{\overline{\eta_i\ \overline{x}^n}^l} \Delta_g$ converts an abstract heap for g into an abstract heap for f. It is defined as follows:

$$\theta \downarrow_{\overline{\eta_i\ \overline{x_j}^n}^l} \Delta_g = \lambda\rho\ .\ \lambda\overline{x_j}^n. \sum_{\substack{\rho' \in R_g \\ \theta\ \rho' = \rho}} \Delta_g\ \rho'\ \overline{\eta_i\ \overline{x_j}^n}^l \qquad (\rho \in R_f \cup \{\rho_{self}\}, \eta_i \in \mathbb{F})$$

In the example, we have:

$$\Delta\ \rho^f_2 = \lambda\overline{x}^n.\Delta_g\ \rho^g_2\ \overline{(|a_i|\ \overline{x}^n)}^l$$
$$\Delta\ \rho^f_1 = \lambda\overline{x}^n.\Delta_g\ \rho^g_1\ \overline{(|a_i|\ \ \overline{x}^n)}^l + \Delta_g\ \rho^g_3\ \overline{(|a_i|\ \ \overline{x}^n)}^l + \Delta_g\ \rho^g_4\ \overline{(|a_i|\ \overline{x}^n)}^l + \Delta_g\ \rho^g_5\ \overline{(|a_i|\ \ \overline{x}^n)}^l$$

Rules [*Let₁*] and [*Let₂*] reflect the corresponding resource-aware semantic rules in [11]. Rules [*Case*] and [*Case*!] use the least upper bound operators $\bigsqcup$ in order to obtain an upper bound to the charge costs and needs of the alternatives.

5 Correctness of the Abstract Interpretation

Let $f\ \overline{x_i}^n\ @\ \overline{r_j}^m = e_f$, be the *context* function, which we assume well-typed according to the type system in [12]. Let us assume an execution of e_f under some E_0, h_0, k_0 and td_0:

$$E_0 \vdash h_0, k_0, td_0, e_f \Downarrow h_f, k_0, v_f, (\delta_0, m_0, s_0) \tag{1}$$

In the following, all $\Downarrow$–judgements corresponding to a given sub-expression of e_f will be assumed to belong to the derivation of (1).

The correctness argument is split into three parts. First, we shall define a notion of *correct signature* which formalises the intuition of the inferred (Δ, μ, σ) being an upper bound of the actual (δ, m, s). Then we prove that the inference rules of Figure 2 are correct, assuming that all function applications are done to previously inferred functions, that the signatures given by Σ for these functions are correct, and that the size analysis is correct. Finally, the correctness of the signature inference algorithm is proved, in particular when the function being inferred is recursive.

In order to define the notion of correct signature we have to give some previous definitions. We consider *region instantiations*, denoted by Reg, $Reg', \ldots$, which are partial mappings from region types ρ to natural numbers i. Region instantiations are needed to specify the actual region i to which every ρ is instantiated at a given execution point. A instantiation Reg is *consistent* with a heap h, an environment E and a type environment Γ if Reg does not contradict the region instantiation obtained at runtime from h, E and Γ, i.e. common type region variables are bound to the same actual region. Formal definition of consistency can be found in [12], where we also proved that if a function is well-typed, consistency of region instantiations is preserved along its execution.

Definition 3. *Given a pointer p belonging to a heap h, the function size returns the number of cells in h of the data structure starting at p:*

$$size(h[p \mapsto (j, C\ \overline{v_i}^n)], p) = 1 + \sum_{i \in RecPos(C)} size(h, v_i)$$

where $RecPos(C)$ denotes the recursive positions of constructor C.

We assume that $size(h, c) = 0$ for every heap h and constant c.

Definition 4. *Given a sequence of sizes $\overline{s_i}^n$ for the input parameters, a number k of regions and a region instantiation Reg, we say that*

- *Δ is an upper bound for δ in the context of $\overline{s_i}^n$, k and Reg, denoted by $\Delta \succeq_{\overline{s_i}^n, k, Reg} \delta$ iff $\forall j \in \{0 \ldots k\} : \sum_{Reg\ \rho = j} \Delta\ \rho\ \overline{s_i}^n \geq \delta\ j$;*
- *μ is an upper bound for m, denoted $\mu \succeq_{\overline{s_i}^n} m$, iff $\mu\ \overline{s_i}^n \geq m$; and*

- *σ is an upper bound for s, denoted $\sigma \succeq_{\overline{s_i}^n} s$, iff $\sigma\ \overline{s_i}^n \geq s$.*

Definition 5 (Correct signature). *Let $(\Delta_g, \mu_g, \sigma_g)$ the signature of a function definition $g\ \overline{y_i}^l\ @\ \overline{r'_j}^q = e_g$. This signature is said to be* correct *iff for all h, h', k, $\overline{v_i}^l$, $\overline{i_j}^q$, v, δ, m, s, Γ, t such that:*

1. *$E_g = [\overline{y_i \mapsto v_i}^l, \overline{r'_j \mapsto i_j}^q, self \mapsto k{+}1] \vdash h, k{+}1, l{+}q, e_g \Downarrow h', k{+}1, v, (\delta, m, s)$.*
2. *$\Gamma_g \vdash e_g : t$*
3. *$\forall i \in \{1 \ldots l\} : s_i = size(h, v_i)$*

then $\Delta_g \succeq_{\overline{s_i}^l, k, Reg} \delta|_k \ \wedge\ \mu_g \succeq_{\overline{s_i}^l} m \ \wedge\ \sigma_g \succeq_{\overline{s_i}^l} s$ for every region instantiation Reg consistent with h, E_g and Γ_g.

The following theorem establishes the correctness of the abstract interpretation for non-recursive functions.

Theorem 1. *Let f a non-recursive context function. For each subexpression e of e_f and E, Σ, Γ, td, Δ, μ, σ, h, ,h', v, ,t, δ, m and s such that:*

1. *Every function call $g\ \overline{a_i}^l\ @\ \overline{r'_j}^q$ in e satisfies $g \in dom\ \Sigma$ and $\Sigma(g)$ is correct*
2. *$[\![e]\!]\ \Sigma\ \Gamma\ td = (\Delta, \mu, \sigma)$*
3. *$E \vdash h, k_0, td, e \Downarrow h', k_0, v, (\delta, m, s)$, belonging to (1)*
4. *$\Gamma \vdash e : t$*

then $\Delta \succeq_{\overline{s_i}^n, k_0, Reg} \delta$, $\mu \succeq_{\overline{s_i}^n} m$ and $\sigma \succeq_{\overline{s_i}^n} s$, where $s_i = size(h, E_0\ x_i)$ for each $i \in \{1 \ldots n\}$, and each region instantiation Reg consistent with h, E and Γ such that $dom\ Reg = dom\ \Delta$.

Proof. By structural induction on e. The proof uses the fact that the size functions are monotonic, and relies on the correctness of the size analysis. □

In order to prove the correctness of the algorithms shown in the following section for recursive functions we need the abstract interpretation to be monotonic with respect to function signatures.

Lemma 1. *Let f be a context function. Given Σ_1, Σ_2, Γ, and td such that $\Sigma_1 \sqsubseteq \Sigma_2$, then $[\![e]\!]\ \Sigma_1\ \Gamma\ td \sqsubseteq [\![e]\!]\ \Sigma_2\ \Gamma\ td$.*

Proof. By structural induction on e, because $+$ and $\sqcup$ are monotonic. □

6 Space Inference Algorithms

Given a recursive function f with $n+m$ arguments, the algorithms for inferring Δ_f and σ_f do not depend on each other, while the algorithm for inferring μ_f needs a correct value for Δ_f. We will assume that μ_f, σ_f, and the cost functions in Δ_f, do only depend on arguments of f non-increasing in size. The consequence of this restriction is that the costs charged to regions, or to the stack, by the most external call to f are safe upper bounds to the costs charged by all the lower level internal calls. This restriction holds for the majority of programs occurring in the literature. Of course, it is always possible to design an example where the charges grow as we progress towards the leafs of the call-tree.

We assume that, for every recursive function f, there has been an analysis giving the following information as functions of the argument sizes $\overline{x_i}^n$:

$$
\begin{array}{l}
\mathit{splitExp}_f\ [\![e]\!] = (e, \#) \qquad \text{if } e = c, x, C\ \overline{a_i}^n\ @\ r,\ \text{ or } g\ \overline{a_i}^n\ @\ \overline{r_j}^m \text{ with } g \neq f \\
\mathit{splitExp}_f\ [\![f\ \overline{a_i}^n\ @\ \overline{r_j}^m]\!] = (\#, f\ \overline{a_i}^n\ @\ \overline{r_j}^m) \\
\mathit{splitExp}_f\ [\![\mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_2]\!] = (e_b, e_r) \\
\qquad \mathbf{where}\ (e_{1b}, e_{1r}) = \mathit{splitExp}_f\ [\![e_1]\!] \\
\qquad \qquad \quad (e_{2b}, e_{2r}) = \mathit{splitExp}_f\ [\![e_2]\!] \\
\qquad \qquad \quad e_b = \begin{cases} \# & \text{if } e_{1b} = \# \text{ or } e_{2b} = \# \\ \mathbf{let}\ x_1 = e_{1b}\ \mathbf{in}\ e_{2b} & \text{otherwise} \end{cases} \\
\qquad \qquad \quad e_r = \begin{cases} \# & \text{if } e_{1r} = \# \text{ and } e_{2r} = \# \\ \mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_{2r} & \text{if } e_{1r} = \# \text{ and } e_{2r} \neq \# \\ \mathbf{let}\ x_1 = e_{1r}\ \mathbf{in}\ e_2 & \text{if } e_{1r} \neq \# \text{ and } e_{2r} = \# \\ \bigsqcup \left\{ \begin{array}{l} \mathbf{let}\ x_1 = e_{1b}\ \mathbf{in}\ e_{2r} \\ \mathbf{let}\ x_1 = e_{1r}\ \mathbf{in}\ e_2 \end{array} \right\} & \text{otherwise} \end{cases} \\
\mathit{splitExp}_f\ [\![\mathbf{case}(!)\ x\ \mathbf{of}\ \overline{\mathit{alt}_i}^n]\!] = (e_b, e_r) \\
\qquad \mathbf{where}\ (\overline{\mathit{alt}_{ib}}^n, \overline{\mathit{alt}_{ir}}^n) = \mathit{unzip}\ (\mathit{map}\ \mathit{splitAlt}_f\ \overline{\mathit{alt}_i}^n) \\
\qquad \qquad \quad e_b = \begin{cases} \# & \text{if } \mathit{alt}_{ib} = \# \to \# \text{ for all } i \in \{1 \ldots n\} \\ \mathbf{case}(!)\ x\ \mathbf{of}\ \overline{\mathit{alt}_{ib}}^n & \text{otherwise} \end{cases} \\
\qquad \qquad \quad e_r = \begin{cases} \# & \text{if } \mathit{alt}_{ir} = \# \to \# \text{ for all } i \in \{1 \ldots n\} \\ \mathbf{case}(!)\ x\ \mathbf{of}\ \overline{\mathit{alt}_{ir}}^n & \text{otherwise} \end{cases} \\
\mathit{splitAlt}_f\ [\![C\ \overline{x_j}^n \to e]\!] = (\mathit{alt}_b, \mathit{alt}_r) \\
\qquad \mathbf{where}\ (e_b, e_r) = \mathit{splitExp}_f\ e \\
\qquad \qquad \quad \mathit{alt}_b = \begin{cases} \# \to \# & \text{if } e_b = \# \\ C\ \overline{x_j}^n \to e_b & \text{otherwise} \end{cases} \\
\qquad \qquad \quad \mathit{alt}_r = \begin{cases} \# \to \# & \text{if } e_r = \# \\ C\ \overline{x_j}^n \to e_r & \text{otherwise} \end{cases}
\end{array}
$$

Fig. 3. Function splitting a *Core-Safe* expression into its base and recursive cases

1. nc_f, an upper bound to the number of calls to f.
2. bf_f, the branching factor of f, i.e. maximum number of internal calls to f for every external call. If $bf_f = 1$ then f has linear recursion.
3. nr_f, an upper bound to the number of calls to f invoking f again. It corresponds to the internal nodes of f's call tree.
4. nb_f, an upper bound to the number of *basic* calls to f. It corresponds to the leaves of f's call tree.
5. len_f, an upper bound to the maximum length of f's call chains. It corresponds to the height of f's call tree.

In general, these functions are not independent of each other. For instance, if $bf_f = 1$ then $nr_f = nc_f - 1$, $nb_f = 1$, and $len_f = nc_f$. However, we will not assume a fixed relation between them. If this relation exists, it has been already used to compute them. We will only assume that each function is a correct upper bound to its corresponding runtime figure.

6.1 Splitting *Core-Safe* expressions

In order to do a more precise analysis, we separately analyse the base and the recursive cases of a *Core-Safe* function definition. Fig. 3 describes the functions *splitExp* and *splitAlt* which, given a *Safe* expression return the part of its body contributing to the base cases and the part contributing to the recursive cases. We introduce an empty expression # in order not to lose the structure of the original one when some parts are removed. These empty expressions charge null

$$
\begin{array}{l}
\mathit{splitBA}_f\ [\![e]\!] = [\,] \qquad\qquad \text{if } e = \#, c, x, C\ \overline{a_i}^n\ @\ r,\ \text{ or } g\ \overline{a_i}^n\ @\ \overline{r_j}^m \text{ with } g \neq f \\
\mathit{splitBA}_f\ [\![\sqcup_{i=1}^n e_i]\!] = \mathit{concat}\,[\mathit{splitBA}\ e_i \mid i \in \{1 \dots n\}] \\
\mathit{splitBA}_f\ [\![f\ \overline{a_i}^n\ @\ \overline{r_j}^m]\!] = [(f\ \overline{a_i}^n\ @\ \overline{r_j}^m, \#)] \\
\mathit{splitBA}_f\ [\![\mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_2]\!] = A +\!\!+ B \\
\qquad \mathbf{where}\ (e_{1b}, e_{1r}) = \mathit{splitExp}_f\ [\![e_1]\!] \\
\qquad\qquad (e_{2b}, e_{2r}) = \mathit{splitExp}_f\ [\![e_2]\!] \\
\qquad\qquad e_{1r,split} = \mathit{splitBA}\ [\![e_{1r}]\!] \\
\qquad\qquad e_{2r,split} = \mathit{splitBA}\ [\![e_{2r}]\!] \\
\qquad\qquad A = [(\mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_{2r,b}, \\
\qquad\qquad\qquad \mathbf{let}\ x_1 = \#\ \mathbf{in}\ e_{2r,a}) \mid (e_{2r,b}, e_{2r,a})\ \in\ e_{2r,split}] \\
\qquad\qquad B = \begin{cases} [\,] & \text{if } e_{2b} = \# \\ [(\mathbf{let}\ x_1 = e_{1r,b}\ \mathbf{in}\ \#, & \\ \qquad \mathbf{let}\ x_1 = e_{1r,a}\ \mathbf{in}\ e_{2b}) \mid (e_{1r,b}, e_{1r,a})\ \in\ e_{1r,split}] & \text{otherwise} \end{cases} \\
\mathit{splitBA}_f\ [\![\mathbf{case}(!)\ x\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \rightarrow e_i}^n]\!] = \\
\qquad\qquad \big[\big(\mathbf{case}(!)\ x\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \rightarrow e_{i,b}}^n, \mathbf{case}(!)\ x\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \rightarrow e_{i,a}}^n\big) \\
\qquad\qquad\quad \mid (e_{1,b}, e_{1,a}) \in \mathit{splitBA}_f\ [\![e_1]\!], \dots, (e_{n,b}, e_{n,a}) \in \mathit{splitBA}_f\ [\![e_n]\!]\big]
\end{array}
$$

Fig. 4. Function splitting a *Core-Safe* expression into its parts executing before and after the last recursive call

costs to both the heap and the stack. Since it might be not possible to split a expression into a single pair with the base and recursive cases, we introduce expressions of the form $\sqcup\ e_i$, whose abstract interpretation is the least upper bound of the interpretations of the e_i. It will also be useful to define another function which splits a *Core-Safe* expression into those parts that execute before and including the last recursive call, and those executed after the last recursive call, In Fig. 4 we define such function, called $\mathit{splitBA}_f$. In Fig. 5 we show a *Full-Safe* definition for a function `split` splitting a list, and the result of applying *splitExp* and *splitBA* to its *Core-Safe* version.

6.2 Algorithm for computing Δ_f

If e_f is f's body, let $(e_r, e_b) = \mathit{splitExp}_f[\![e_f]\!]$ and $(e_{bef}, e_{aft}) = (\bigsqcup_i\ e^i_{bef}, \bigsqcup_i\ e^i_{aft})$, where $[\overline{(e^i_{bef}, e^i_{aft})}^n] = \mathit{splitBA}_f[\![e_r]\!]$. The idea here is to separately compute the charges to regions of the recursive and non-recursive parts of f's body, and then multiply these charges by respectively the number of internal and leaf nodes of f's call-tree.

1. Set $\Sigma\ f = ([\,]_f, 0, 0)$.
2. Let $(\Delta_r, \_, \_) = [\![e_r]\!]\ \Sigma\ \Gamma\ (n+m)$
3. Let $(\Delta_b, \_, \_) = [\![e_b]\!]\ \Sigma\ \Gamma\ (n+m)$
4. Then, $\Delta_f \stackrel{\text{def}}{=} \Delta_r\mid_{\rho \neq \rho_{self}} \times nr_f + \Delta_b\mid_{\rho \neq \rho_{self}} \times nb_f$.

Lemma 2. *If nr_f, nb_f, and all the size functions belong to $\mathbb{F}$, then all functions in Δ_f belong to $\mathbb{F}$.*

Lemma 3. *Δ_f is a correct abstract heap for f.*

Proof. This is a consequence of nr_f, nb_f, and all the size functions being upper bounds of their respective runtime figures, and of Δ_r, Δ_b being upper bounds of respectively the f's call-tree internal and leaf nodes heap charges. □

```
split 0 xs     = ([],xs)
split n []     = ([],[])
split n (x:xs) = (x:xs1,xs2)
   where (xs1,xs2) = split (n-1) xs

split n xs @ r1 r2 r3 =
 case n of
  _ -> case xs of
       (: y1 y2) ->
          let y3 = let x6 = - n 1 in
                   split x6 y2 @ r1 r2 r3 in #
```

(*Full-Safe* version, and *Core-Safe* up to the last call)

```
split n xs @ r1 r2 r3 =
 case n of
  0 -> let  x1 = [] @ r2 in
       let  x2 = (x1,xs) @ r3 in x2
  _ -> case xs of
        [] -> let x4 = [] @ r2 in
              let x3 = [] @ r1 in
              let x5 = (x4,x3) @ r3 in x5
```

(*Core-Safe* base cases)

```
split n xs @ r1 r2 r3 =
 case n of
  _ -> case xs of
       (: y1 y2) ->
          let y3  = let x6 = - n 1 in
                    split x6 y2 @ r1 r2 r3 in
          let xs1 = case y3 of (y4,y5) -> y4 in
          let xs2 = case y3 of (y6,y7) -> y7 in
          let x7  = (: y1 xs1) @ r2 in
          let x8  = (x7,xs2) @ r3 in x8
```

(*Core-Safe* recursive cases)

```
split n xs @ r1 r2 r3 =
 case n of
  _ -> case xs of
       (: y1 y2) ->
          let y3  = # in
          let xs1 = case y3 of (y4,y5) -> y4 in
          let xs2 = case y3 of (y6,y7) -> y7 in
          let x7  = (: y1 xs1) @ r2 in
          let x8  = (x7,xs2) @ r3 in x8
```

(*Core-Safe* after the last call)

Fig. 5. Splitting a *Core-Safe* definition

Let us call $\mathbb{I}_\Delta : \mathbb{D} \rightarrow \mathbb{D}$ to an iteration of the interpretation function, i.e. $\mathbb{I}_\Delta(\Delta_1) = \Delta_2$, being Δ_2 the abstract heap obtained by initially setting $\Sigma\ f = (\Delta_1, 0, 0)$, then computing $(\Delta, \_, \_) = [\![e_r]\!]\ \Sigma\ \Gamma\ (n+m)$, and then defining $\Delta_2 = \Delta\mid_{\rho \neq \rho_{self}}$.

Lemma 4. *For all n, $\mathbb{I}^n_\Delta(\Delta_f)$ is a correct abstract heap for f.*

Proof. This is a consequence of $\mathbb{D}$ being a complete lattice, $\mathbb{I}_\Delta$ being monotonic in $\mathbb{D}$, and $\mathbb{I}_\Delta(\Delta_f) \sqsubseteq \Delta_f$. As $\mathbb{I}_\Delta$ is reductive at Δ_f then, by Tarski's fixpoint theorem, $\mathbb{I}^n_\Delta(\Delta_f)$ is above the least fixpoint of $\mathbb{I}_\Delta$ for all n. □

As the algorithm for μ_f critically depends on how good is the result for Δ_f, it is advisable to spend some time iterating the interpretation $\mathbb{I}_\Delta$ in order to get better results for μ_f.

6.3 Algorithm for computing μ_f

First, we infer the part of μ_f due to space charges to the *self* region of f. Let us call it μ_f^{self}. As the *self* regions for f are stacked, this part only depends on the longest f's call chain, i.e. on the height of the call-tree.

1. Set $\Sigma\ f = ([\,]_f, 0, 0)$.
2. Let $([\rho_{self} \mapsto \mu_{bef}], \_, \_) = [\![e_{bef}]\!]\ \Sigma\ \Gamma\ (n+m)$, i.e. the charges to ρ_{self} made by the part of $f's$ body before (and including) the last recursive call.
3. Let $([\rho_{self} \mapsto \mu_{aft}], \_, \_) = [\![e_{aft}]\!]\ \Sigma\ \Gamma\ (n+m)$, i.e. the charges to ρ_{self} made by the part of $f's$ body after the last recursive call.
4. Let $([\rho_{self} \mapsto \mu_b], \_, \_) = [\![e_b]\!]\ \Sigma\ \Gamma\ (n+m)$, i.e. the charges to ρ_{self} made by the non-recursive part of $f's$ body.

5. Then, $\mu_f^{self} \stackrel{\text{def}}{=} \mu_{bef} \times (len_f - 1) + \sqcup\{\mu_b, \mu_{aft}\}$.

Now, the inference of μ_f is done by provisionally assuming a signature for f in which f's heap needs are at least those due to charges to *self*, plus those due to charges to other regions. The latter are recorded in Δ_f.

1. Let $\mu_{prov} \stackrel{\text{def}}{=} |\, \Delta_f \,| + \mu_f^{self}$
2. Set $\Sigma\ f = (\Delta_f, \mu_{prov}, 0)$.
3. Then, $(\_, \mu_f, \_) = [\![e_f]\!]\ \Sigma\ \Gamma\ (n+m)$.

Lemma 5. *If the functions in Δ_f, len_f, and the size functions belong to $\mathbb{F}$, then μ_f belongs to $\mathbb{F}$.*

Lemma 6. *μ_f is a safe upper bound for f's heap needs.*

Proof. This is a consequence of the correctness of the abstract interpretation rules, and of Δ_f, len_f, and the size functions being upper bounds of their respective runtime figures. □

As in the case of Δ_f, we can define an interpretation $\mathbb{I}_\mu$ taking any upper bound μ_1 as input, and producing a better one $\mu_2 = \mathbb{I}_\mu(\mu_1)$ as output.

Lemma 7. *For all n, $\mathbb{I}_\mu^n(\mu_f)$ is a safe upper bound for f's heap needs.*

Proof. This is a consequence of $\mathbb{F}$ being a complete lattice, $\mathbb{I}_\mu$ being monotonic in $\mathbb{F}$, and $\mathbb{I}_\mu$ being reductive at μ_f. □

6.4 Algorithm for computing σ_f

The algorithm for inferring μ_f traverses f's body from left to right because the abstract interpretation rules for μ need the charges to the previous heaps. For inferring σ_f we can do it better because its rules are symmetrical. The main idea is to count only once the stack needs due to calling to external functions.

1. Let $(\_, \_, \sigma_b) = [\![e_b]\!]\ \Sigma\ \Gamma\ (n+m)$.
2. Let $(\_, \_, \sigma_{bef}) = [\![e_{bef}]\!]\ \Sigma[f \mapsto (\_, \_, \sigma_b)]\ \Gamma\ (n+m)$, i.e. the stack needs before the last recursive call, assuming as f's stack needs those of the base case. This amounts to accumulating the cost of a leaf to the cost of an internal node of f's call tree.
3. Let $(\_, \_, \sigma_{aft}) = [\![e_{aft}]\!]\ \Sigma\ \Gamma\ (n+m)$.
4. We define the following function $\mathcal{S}$ returning a natural number. Intuitively it computes an upper bound to the difference in words between the initial stack in a call to f and the stack just before e_{bef} is about to jump to f again:

$$
\begin{array}{ll}
\mathcal{S}\ [\![\textbf{let}\ x_1 = e_1\ \textbf{in}\ \#]\!]\ td & = 2 + \mathcal{S}\ [\![e_1]\!]\ 0 \\
\mathcal{S}\ [\![\textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2]\!]\ td & = \begin{cases} 1 + \mathcal{S}\ [\![e_2]\!]\ (td+1) & \text{if } f \notin e_1 \\ \sqcup\{2 + \mathcal{S}\ [\![e_1]\!]\ 0, 1 + \mathcal{S}\ [\![e_2]\!]\ (td+1)\} & \text{if } f \in e_1 \end{cases} \\
\mathcal{S}\ [\![\textbf{case}\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^{n}]\!]\ td & = \bigsqcup_{r=1}^{n}(n_r + \mathcal{S}\ [\![e_r]\!]\ (td + n_r)) \\
\mathcal{S}\ [\![g\ \overline{a_i}^{p}\ @\ \overline{r_j}^{q}]\!]\ td & = p + q - td \\
\mathcal{S}\ [\![e]\!]\ td & = 0 \qquad \text{otherwise}
\end{array}
$$

```
length []     = 0
length (x:xs) = 1 + length xs
```

$split :: Int \to [a]@\rho_1 \to \rho_1 \to \rho_2 \to \rho_3 \to ([a]@\rho_2, [a]@\rho_1)@\rho_3$
$length :: [a]@\rho_1 \to Int$
$merge :: [a]@\rho_1 \to [a]@\rho_1 \to \rho_1 \to [a]@\rho_1$
$msort :: [a]@\rho_1 \to \rho_1 \to \rho_2 \to [a]@\rho_2$

```
merge []     ys = ys
merge (x:xs) [] = x : xs
merge (x:xs) (y:ys)
  | x <= y = x : merge xs (y:ys)
  | x > y  = y : merge (x:xs) ys
```

```
msort []      = []
msort (x:[])  = x:[]
msort xs      = merge (msort xs1) (msort xs2)
                  where (xs1,xs2) = split (length xs / 2) xs
```

Fig. 6. *Full-Safe* mergesort program

Function	Heap charges Δ	Heap needs μ	Stack needs σ
$length(x)$	$[\,]$	0	$5x - 4$
$split(n, x)$	$\begin{bmatrix} \rho_1 \mapsto 1 \\ \rho_2 \mapsto \min(n, x-1)+1 \\ \rho_3 \mapsto \min(n, x-1)+1 \end{bmatrix}$	$2\min(n, x-1)+3$	$9\min(n, x-1)+4$
$merge(x, y)$	$\begin{bmatrix} \rho_1 \mapsto \max(1, 2x+2y-5) \end{bmatrix}$	$\max(1, 2x+2y-5)$	$11(x+y-4)+20$
$msort^1(x)$	$\begin{bmatrix} \rho_1 \mapsto \frac{x^2}{2} - \frac{1}{2} \\ \rho_2 \mapsto 2x^2 - 3x + 3 \end{bmatrix}$	$0.31x^2 + 0.25x\log(x+1) + 14.3x$ $+ 0.75\log(x+1) + 10.3$	$\max(80, 13x - 10)$
$msort^2(x)$	$\begin{bmatrix} \rho_1 \mapsto \frac{x^2}{4} + x - \frac{1}{4} \\ \rho_2 \mapsto x^2 + x + 1 \end{bmatrix}$	$0.31x^2 + 8.38x + 13.31$	$\max(80, 11x - 25)$
$msort^3(x)$	$\begin{bmatrix} \rho_1 \mapsto \frac{x^2}{8} + \frac{7x}{4} + \frac{9}{8} \\ \rho_2 \mapsto \frac{x^2}{2} + 4x + \frac{1}{2} \end{bmatrix}$	$0.31x^2 + 8.38x + 13.31$	$\max(80, 11x - 25)$

Fig. 7. Cost results for the mergesort program

5. Then, $\sigma_f = (\mathcal{S}\ [\![e_{bef}]\!]\ (n+m)) * \sqcup\{0, len_f - 2\} + \sqcup\{\sigma_{bef}, \sigma_{aft}, \sigma_b\}$

Lemma 8. *If len_f, and all the size functions belong to $\mathbb{F}$, then σ_f belongs to $\mathbb{F}$.*

Lemma 9. *σ_f is a safe upper bound for f's stack needs.*

Proof. This is a consequence of the correctness of the abstract interpretation rules, and of len_f being an upper bound to f's call-tree height. $\square$

Also in this case, it makes sense iterating the interpretation as we did with Δ_f and μ_f, since it holds that $\mathbb{I}_\sigma(\sigma_f) \sqsubseteq \sigma_f$.

7 Case Studies

In Fig. 6 we show a *Full-Safe* version of the mergesort algorithm (the code for `split` was presented in Fig. 5) with the types inferred by the compiler. Region ρ_1 is used inside `msort` for the internal call `split n' xs @ r1 r1 self`, while region ρ_2 receives the charges made by `merge`. Notice that some charges to `msort`'s *self* region are made by `split`. In Fig. 7 we show the results of our interpretation for this program as functions of the argument sizes. Remember that the size of a list (the number of its cells) is the list length plus one. The functions shown have been simplified with the help of a computer algebra tool. We show the fixpoints framed in grey. The upper bounds obtained for `length`, `split`, and `merge` are

Function	**Heap needs μ**	**Stack needs σ**
$partition(p, x)$	$3x - 1$	$9x - 5$
$append(x, y)$	$x - 1$	$\max(8, 7x - 6)$
$quicksort(x)$	$3x^2 - 20x + 76$	$\max(40, 20x - 27)$
$insertD(e, x)$	1	$9x - 1$
$insertTD(x, t)$	2	$\frac{11}{2}t + \frac{7}{2}$
$fib(n)$	$2^n + 2^{n-3} + 2^{n-4} - 3$	$\max(10, 7n - 11)$
$sum(n)$	0	$3n + 6$
$sumT(a, n)$	0	5

Fig. 8. Cost results for miscellaneous *Safe* functions

```
sum 0 = 0
sum n = n + sum (n - 1)

sumT acc 0 = acc
sumT acc n = sumT (acc + n) (n - 1)
```

```
insertTD x Empty! = Node (Empty) x (Empty)
insertTD x (Node lt y rt)!
  | x == y = Node lt! y rt!
  | x > y  = Node lt! y (insertTD x rt)
  | x < y  = Node (insertTD x lt) y rt!
```

Fig. 9. Two summation functions and a destructive tree insertion function

exact and they are, as expected, fixpoints of the inference algorithm. For `msort` we show three iterations for Δ and σ, and another three for μ by using the last Δ. The upper bounds for Δ and μ are clearly over-approximated, since a term in x^2 arises which is beyond the actual space complexity class $O(x \log x)$ of this function. Let us note that the quadratic term's coefficient quickly decreases at each iteration in the inference of Δ. Also, μ and σ decrease in the second iteration but not in the third. This confirms the predictions of lemmas 4 and 7.

We have tried some more examples and the results inferred for μ and σ after a maximum of three iterations are shown in Fig. 8, where the fixpoints are also framed in grey. There is a `quicksort` function using two auxiliary functions `partition` and `append` respectively classifying the list elements into those lower (or equal) and greater than the pivot, and appending two lists. We also show the destructive `insertD` function of Sec. 2, and a destructive version of the insertion in a search tree (its code is shown in Fig. 9). Both consume constant heap space. The next one shown is the usual Fibonacci function with exponential time cost, and using a constructed integer in order to show that an exponential heap space is inferred. Finally, we show two simple summation functions (its code also appears in Fig. 9), the first one being non-tail recursive, and the second being tail-recursive. Our abstract machine consumes constant stack space in the second case (see [11]). It can be seen that our stack inference algorithm is able to detect this fact.

8 Related and Future Work

Hughes and Pareto developed in [7] a type system and a type-checking algorithm which guarantees safe memory upper bounds in a region-based first-order functional language. Unfortunately, the approach requires the programmer to provide detailed consumption annotations, and it is limited to linear bounds. Hofmann and Jost's work [6] presents a type system and a type inference algorithm which, in case of success, guarantees linear heap upper bounds for a first-order functional language, and it does not require programmer annotations.

The national project AHA [15] aims at inferring amortised costs for heap space by using a variant of sized-types [8] in which the annotations are polynomials of any degree. They address two novel problems: polynomials are not necessarily monotonic and they are *exact* bounds, as opposed to approximate upper bounds. Type-checking is undecidable in this system and in [16, 14] they propose an inference algorithm for a list-based functional language with severe restrictions in which a combination of testing and type-checking is done. The algorithm does not terminate if the input-output size relation is not polynomial.

In [2], the authors directly analyse Java bytecode and compute safe upper bounds for the heap allocation made by a program. The approach uses the results of [1], and consists of combining a code transformation to an intermediate representation, a cost relations inference step, and a cost relations solving step. The second one combines ranking functions inference and partial evaluation. The results are impressive and go far beyond linear bounds. The authors claim to deal with data structures such as lists and trees, as well as arrays. Two drawbacks compared to our results are that the second step performs a global program analysis (so, it lacks modularity), and that only the allocated memory (as opposed to the live memory) is analysed. Very recently [3] they have added an escape analysis to each method in order to infer live memory upper bounds. The new results are very promising.

The strengths of our approach can be summarised as follows: (a) It scales well to large programs as each *Safe* function is separately inferred. The relevant information about the called functions is recorded in the signature environment; (b) We can deal with any user-defined algebraic datatype. Most of other approaches are limited to lists; (c) We get upper bounds for the *live* memory, as the inference algorithms take into account the deallocation of dead regions made at function termination; (d) We can get bounds of virtually any complexity class; and (e) It is to our knowledge the only approach in which the upper bounds can be easily improved just by iterating the inference algorithm.

The weak points that still require more work are the restrictions we have imposed to our functions: they must be non-negative and monotonic. This exclude some interesting functions such as those that destroy more memory than they consume, or those whose output size decreases as the input size increases. Another limitation is that the arguments of recursive *Safe* functions related to heap or stack consumption must be non-increasing. This limitation could be removed in the future by an analysis similar to that done in [1] in which they maximise the argument sizes across a call-tree by using linear programming tools. Of course, this could only be done if the size relations are linear.

Another open problem is inferring *Safe* functions with region-polymorphic recursion. Our region inference algorithm [13] frequently infers such functions, where the regions used in an internal call may differ from those used in the external one. This feature is very convenient for maximising garbage (i.e. allocations to the *self* region) but it makes more difficult the attribution of costs to regions.

References

1. E. Albert, P. Arenas, S. Genaim, and G. Puebla. Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis. In *Static Analysis Symposium, SAS'08*, pages 221–237. LNCS 5079, Springer, 2008.
2. E. Albert, S. Genaim, and M. Gómez-Zamalloa. Heap Space Analysis for Java Bytecode. In *Proc. Int. Symp. on Memory Management, ISMM'07, Montreal, Canada*, pages 105–116. ACM, 2007.
3. E. Albert, S. Genaim, and M. Gómez-Zamalloa. Live Heap Space Analysis for Languages with Garbage Collection. In *Proc. Int. Symp. on Memory Management, ISMM'09, Dublin, Ireland*, pages 129–138. ACM, 2009.
4. J. de Dios and R. Peña. A Certified Implementation on top of the Java Virtual Machine. In *Formal Method in Industrial Critical Systems, FMICS'09, Eindhoven (The Netherlands)*, pages 1–16. LNCS (to appear), Springer, November 2009.
5. J. de Dios and R. Peña. Formal Certification of a Resource-Aware Language Implementation. In *Int. Conf. on Theorem Proving in Higher Order Logics, TPHOL'09, Munich (Germany)*, pages 1–15. LNCS 5674 (to appear), Springer, August 2009.
6. M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *Proc. 30th ACM Symp. on Principles of Programming Languages, POPL'03*, pages 185–197. ACM Press, 2003.
7. R. J. M. Hughes and L. Pareto. Recursion and Dynamic Data-Structures in Bounded Space; Towards Embedded ML Programming. In *Proc. Int. Conf. on Functional Programming, ICFP'99, Paris*, pages 70–81. ACM Press, Sept. 1999.
8. R. J. M. Hughes, L. Pareto, and A. Sabry. Proving the Correctness of Reactive Systems Using Sized Types. In *Conference Record of POPL '96: The* 23^{rd} *ACM SIGPLAN-SIGACT*, pages 410–423, 1996.
9. T. Lindholm and F. Yellin. *The Java Virtual Machine Sepecification Second Edition.* The Java Series. Addison-Wesley, 1999.
10. S. Lucas and R. Peña. Rewriting Techniques for Analysing Termination and Complexity Bounds of SAFE Programs. In *Proc. Logic-Based Program Synthesis and Transformation, LOPSTR'08, Valencia, Spain*, pages 43–57, July 2008.
11. M. Montenegro, R. Peña, and C. Segura. A Resource-Aware Semantics and Abstract Machine for a Functional Language with Explicit Deallocation. In *Workshop on Functional and (Constraint) Logic Programming, WFLP'08, Siena, Italy July, 2008 (to appear in ENTCS)*, pages 47–61, 2008.
12. M. Montenegro, R. Peña, and C. Segura. A Type System for Safe Memory Management and its Proof of Correctness. In *ACM Principles and Practice of Declarative Programming, PPDP'08, Valencia, Spain, July. 2008*, pages 152–162, 2008.
13. M. Montenegro, R. Peña, and C. Segura. A simple region inference algorithm for a first-order functional language. In S. Escobar, editor, *Int. Work. on Functional and (Constraint) Logic Programming, WFLP 2009, Brasilia*, pages 63–77, 2009.
14. Olha Shkaravska, Marko C. J. D. van Eekelen, and Ron van Kesteren. Polynomial size analysis of first-order shapely functions. *Logical Methods in Computer Science*, 5(2:10):1–35, 2009. Selected Papers from TLCA 2007.
15. M. van Eekelen, O. Shkaravska, R. van Kesteren, B. Jacobs, E. Poll, and S. Smetsers. AHA: Amortized Space Usage Analysis. In *Selected Papers Trends in Functional Programming, TFP'07, New York*, pages 36–53. Intellect, 2008.
16. R. van Kesteren, O. Shkaravska, and M. van Eekelen. Inferring static non-monotonically sized types through testing. In *Proc. Work. on Functional and (Constraint) Logic Programming, WFLP'07, Paris, France.* ENTCS, Elsevier, 2007.

Polynomials over the reals are safe for programs interpretations

Guillaume Bonfante[1], Florian Deloup[2] and Antoine Henrot[3]

1- Université de Nancy - LORIA, Nancy, France,
2- Université Paul Sabatier, Toulouse, France
3- Université de Nancy - IECN, Nancy, France

Abstract. In the field of implicit computational complexity, we are considering in this paper the fruitful branch of interpretation methods. Due to their good intensional properties, they have been widely developped. Among usual issues is the synthesis problem which has been solved by the use of Tarski's decision procedure, and consequently interpretations are usually chosen over the reals rather than over the integers. Doing so, one cannot use anymore the (good) properties of the natural (well-) ordering of **N** employed to bound the complexity of programs. We show that, actually, polynomials over the reals benefit from some properties that allows their safe use for complexity. We illustrate this by two characterizations, one of PTIME and one of PSPACE.

Among studies in rewriting are the noticeable work concerning termination. This is now a largely and thoroughly studied field, and very elegant methods have been proposed that cover a large spectrum of algorithms (see for instance [6]). These studies can be refined to get characterizations of the complexity of first order program. This has been one of the main successful approach of implicit computational complexity, see for instance the early works [2, 4].

To prove termination by interpretation over a well-founded ordering seems rather natural, and such interpretation methods have been introduced in the 70's (see [12, 11]). Lankford describes interpretations as monotone Σ-algebras with domain of interpretation being the natural numbers with their usual ordering. The fact that this ordering is well-founded gives immediately the well-foundedness of the rewriting relation. Based on Kruskal's Theorem, Dershowitz showed in [5] that the well-foundedness of the domain of interpretation is not necessary whenever the interpretations are chosen monotonic and have the subterm property.

One of the main interesting points about choosing real numbers rather than natural numbers is that we get (at least from a theoretical point of view) a procedure to verify the validity of an interpretation of a program by Tarski's decomposition procedure [20]. But furthermore, we can even give a semi-algorithm to compute interpretations: indeed, for a given choice of the degree of the interpretations, finding the coefficients of these polynomial becomes decidable. Actually, since the problem of the verification or the synthesis (up to some degree) can be stated by a first order formula with 2 alternations of quantifiers, following Roy et al. [17], the complexity of these algorithms is exponential with respect to the size of the program. To obtain a faster procedure, we have used in the CROCUS tool –developped by the first author– the (sufficient) criterion of Hong and Jakuš [10].

A second good point is that the use of reals versus integers enlarges the set of rewrite systems that have an interpretation as this has been shown recently by Lucas [13].

The counterpart is that the (usual) ordering on $\mathbf{R}$ is not well-founded. Consequently, the analysis of programs with interpretations over the reals has to be completely reconsidered. As a matter of fact, our contribution is to show that, if interpretations are restricted to polynomials[1], then, choosing integers or reals does not change the time complexity, neither the space complexity (up to some polynomials).

More precisely, due to Positivstellensatz, one recovers the two main features of interpretations, the bound on the derivation length of the rewriting relation and the size of computed terms. In some way, we follow the belief of Lucas in [14] who states that the field of algebraic geometry may give some new insight on issues about polynomial interpretations over the reals.

The use of interpretations to compute the size of terms is a widely used tool in implicit complexity theory. In particular, this has been considered by Shkaravska et al [18] to bound the size of computed terms.

Here, we propose the application of Stengle's Positivstellensatz [19]. This deep result deals with the algebraic structure of polynomials over the reals. We actually show that this structure has some fundamental consequences from the point of view of the complexity of programs.

One of our two characterization use dependency pairs. We mention here the work of Hirokawa and Moser [8] which is in the same spirit.

Due to lack of space, proofs are omitted. They can be found at `http://www.loria.fr/~bonfante/papers/fopara09.pdf`.

1 Preliminaries

We suppose that the reader has familiarity with first order rewriting. We briefly recall the context of the study, esssentially to fix the notations. Dershowitz and Jouannaud's survey [7] of rewriting is a good entry point for beginners.

1.1 Syntax of programs

All along, $\mathcal{X}$ will denote a set of variables, $\mathcal{C}$ a (finite) signature of constructor symbols and $\mathcal{F}$ a (finite) signature of function symbols.

[1] Potentially closed by the max function.

Definition 1. *The sets of terms and the rules are defined in the following way:*

(Constructor terms)	$\mathcal{T}(\mathcal{C}) \ni u$	$::=$	$\mathbf{c} \mid \mathbf{c}(u_1, \cdots, u_n)$
(terms)	$\mathcal{T}(\mathcal{C}, \mathcal{F}, \mathcal{X}) \ni t$	$::=$	$\mathbf{c} \mid x \mid \mathbf{c}(t_1, \cdots, t_n) \mid f(t_1, \cdots, t_n)$
(patterns)	$\mathcal{P} \ni p$	$::=$	$\mathbf{c} \mid x \mid \mathbf{c}(p_1, \cdots, p_n)$
(D-rules)	$\mathcal{D} \ni d$	$::=$	$f(p_1, \ldots, p_n) \to t$

where $x \in \mathcal{X}$, $f \in \mathcal{F}$, and $\mathbf{c} \in \mathcal{C}$. We shall use a type writer font for function symbols and a bold face font for constructors. Finally, we use $\boldsymbol{u}$ to denote a (finite) sequence of terms $u_1, \ldots, u_n$.

The size of a term is defined inductively as $\mathsf{S}(x) = 1$, $\mathsf{S}(\mathbf{c}) = 1$, $\mathsf{S}(\mathbf{c}(t_1, \cdots, t_)) = 1 + \sum_{i=1}^{n} \mathsf{S}(t_i)$ and $\mathsf{S}(\mathtt{f}(t_1, \cdots, t_)) = 1 + \sum_{i=1}^{n} \mathsf{S}(t_i)$.

A *context* is a term C with a particular variable $\Diamond$. If t is a term, $C[t]$ denotes the term C where the variable $\Diamond$ has been replaced by t.

Definition 2. *A program is a quadruplet $\mathtt{f} = \langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E} \rangle$ such that $\mathcal{E}$ is a finite set of $\mathcal{D}$-rules. Each variable in the right-hand side of a rule also appears in the left hand side of the same rule. We distinguish among $\mathcal{F}$ a main function symbol whose name is given by the program name $\mathtt{f}$. We denote by $\mathtt{F}$-program, the set of these programs.*

The set of rules induces a rewriting relation $\to$. The relation $\xrightarrow{*}$ is the reflexive and transitive closure of $\to$. All along, we suppose programs to be *confluent*, that is the rewriting relation is confluent.

We note $t_0 \to^n t_n$ the fact that $t_0 \to t_1 \cdots \to t_n$. One defines the derivation height for a term t as the maximal length of a derivation:

$$\mathtt{dh}(t) = \max\{n \in \mathbf{N} \mid \exists v : t \to^n v\}.$$

The domain of the computed functions is the constructor term algebra $\mathcal{T}(\mathcal{C})$. The program $\mathtt{f}$ computes a partial function $[\![\mathtt{f}]\!] : \mathcal{T}(\mathcal{C})^n \to \mathcal{T}(\mathcal{C})$ defined as follows. For every $u_1, \cdots, u_n \in \mathcal{T}(\mathcal{C})$, $[\![\mathtt{f}]\!](u_1, \cdots, u_n) = v$ iff $\mathtt{f}(u_1, \cdots, u_n) \xrightarrow{*} v$. Otherwise, it is undefined and $[\![\mathtt{f}]\!](u_1, \cdots, u_n) = \bot$.

Definition 3 (Call-tree). *Suppose we are given a program $\langle \mathcal{C}, \mathcal{F}, \mathcal{E} \rangle$. Let $\rightsquigarrow$ be the relation*

$$(f, t_1, \ldots, t_n) \rightsquigarrow g(u_1, \ldots, u_m) \Leftrightarrow f(t_1, \ldots, t_n) \to C[g(v_1, \ldots, v_m)] \xrightarrow{*} C[g(u_1, \ldots, u_m)]$$

where f and g are defined symbols, C is a context and $t_1, \ldots, t_n, u_1, \ldots, u_m$ are constructor terms. Given a term $f(t_1, \ldots, t_n)$, the relation $\rightsquigarrow$ defines a tree whose root is $(f, t_1, \ldots, t_n)$ and η' is a daughter of η iff $\eta \rightsquigarrow \eta'$. The size of a call-tree is the number of nodes it contains.

When we do not make a distinction between constructors and function symbols, we speak (improperly) of Term Rewriting System. We present them as a couple (Σ, R) where Σ is the signature and R is the set of rules.

1.2 Interpretations of programs

Given a signature Σ, a Σ-algebra on the domain A is a mapping $(\!|-|\!)$ which associates to every n-ary symbol $f \in \Sigma$ an n-ary function $(\!|f|\!) : A^n \to A$. A Σ-algebra can be extended to terms by

- $(\!|x|\!) = 1_A$, that is the identity on A, for $x \in \mathcal{X}$,
- $(\!|f(t_1, \dots, t_m)|\!) = \text{comp}((\!|f|\!), (\!|t_1|\!), \dots, (\!|t_m|\!))$ where comp is the composition of functions.

The interpretation $(\!|t|\!)$ of a term t with n variables is then a function $\mathcal{T}(\Sigma)^n \to \mathcal{T}(\Sigma)$.

An interpretation for a rewrite system (Σ, R) is an order-preserving mapping $(\!|-|\!)$ from $(\mathcal{T}(\Sigma), \xrightarrow{+})$ to some (partially) ordered set $(A, >)$. Actually, from now on, we restrict our attention to the case where $(A, >)$ is the set of non negative real numbers $\mathbf{R}^+$ with the usual ordering and $(\!|-|\!)$ has the structure of a Σ-algebra:

Definition 4. *A strict interpretation of a rewriting system (Σ, R) is given by a Σ-algebra $(\!|-|\!)$ such that:*

1. *for all symbol f, the interpretation $(\!|f|\!)$ is a monotonic function, that is if $x_i > x'_i$, then*
$$(\!|f|\!)(x_1, \dots, x_n) > (\!|f|\!)(x_1, \dots, x'_i, \dots, x_n),$$
2. *for all symbol f, the interpretation $(\!|f|\!)$ verifies the (weak) sub-term property, that is $(\!|f|\!)(x_1, \dots, x_n) \geq x_i$ with $i \in 1..n$,*
3. *for all rules $\ell \to r$, $(\!|\ell|\!) > (\!|r|\!)$.*

Example 1. Consider the system

$$\begin{pmatrix} \mathbf{A}(\mathbf{B}(x)) \to \mathbf{B}(\mathbf{B}(\mathbf{A}(x))) \\ \mathbf{c}(\mathbf{A}(x)) \to \mathbf{A}(\mathbf{A}(\mathbf{c}(x))) \end{pmatrix}$$

For this system, we define the interpretation $(\!|\mathbf{A}|\!)(x) = 3(x + 2)$, $(\!|\mathbf{B}|\!)(x) = x + 1$ and $(\!|\mathbf{c}|\!)(x) = x^2 + 1$.

From now on, we restrict interpretations to be *Max-Poly functions*, that is functions obtained by finite compositions of maximum, addition and multiplication. We note Max-Poly, the set of these functions. To stress this choice, we say that interpretations are chosen over Max-Poly.

Sup-interpretation have been introduced by Marion and Pechoux in [15]. We give a slight variant of their definition. In [15], the last inequality refers to the size of normal forms. We prefer to have a more uniform definition.

Definition 5. *A sup-interpretation of a rewriting system (Σ, R) is given by a Σ-algebra $(\!|-|\!)$ such that:*

1. *$(\!| f |\!)$ is a weakly monotonic function, that is if $x_i \geq x'_i$, then*

$$(\!| f |\!)(x_1, \ldots, x_n) \geq (\!| f |\!)(x_1, \ldots, x'_i, \ldots, x_n),$$

2. *for all constructor terms $t_1, \ldots, t_n$, we have the inequality $(\!| f(t_1, \ldots, t_n) |\!) \geq (\!| [\![f]\!](t_1, \ldots, t_n) |\!)$.*

Lemma 1. *Suppose that we are given a Σ-algebra for which interpretations of symbols f is bounded by some polynomials. Then, for all terms, $(\!| t |\!) \leq 2^{2^{O(|t|)}}$.*

Proof. By induction on terms. A proof (for natural numbers, but this has no consequences here) can be found in [2].

Definition 6. *The interpretation of a symbol f is said to be additive (resp. affine, resp multiplicative) if it has the shape $\sum_i x_i + c$ (resp. $\sum a_i x_i + c$, resp. any polynomial). A program with an interpretation is said to be additive(resp. affine, resp. multiplicative) when its* constructors *are additive (resp. affine, resp multiplicative).*

2 Positivstellensatz and applications

Given a strict interpretation for a TRS (Σ, R), it follows immediately that for all $s \to t$, $(\!| s |\!) > (\!| t |\!)$. If one takes the interpretation on natural numbers (as they were introduced by Lankford [12]), this can be used to give a bound on the derivation height. Thus, Hofbauer and Lautemann have shown in [9] that the derivation height is bounded by a double exponential. However, their argument uses deeply the fact that the interpretation of a term is itself a bound on the derivation height:

$$\mathtt{dh}(t) \leq (\!| t |\!). \tag{1}$$

Indeed, suppose $t_1 \to t_2 \to \cdots \to t_n$, then $(\!| t_1 |\!) > (\!| t_2 |\!) > \cdots > (\!| t_n |\!)$. On natural numbers, this means that $n \leq (\!| t_1 |\!)$, a conclusion that does not hold for interpretation over the reals.

Equation 1 comes from the fact that $(\!| t |\!) \geq (\!| u |\!) + 1$ for terms $t \to u$. This fact itself is due to a) $(\!| \ell |\!) > (\!| r |\!)$ implies that

$$(\!| \ell |\!) \geq (\!| r |\!) + 1 \tag{2}$$

and b) that for all $x_i > y_i$:

$$(\!| f |\!)(x_1, \ldots, x_i, \ldots, x_n) - (\!| f |\!)(x_1, \ldots, y_i, \ldots, x_n) \geq x_i - y_i. \tag{3}$$

The two inequalities 2, 3 do not hold in general for real interpretations.

In the context of termination proofs by dependency pairs, when using real interpretations instead of integer interpretations, people have enforced the hypotheses to recover the good properties holding with natural numbers. For instance [13, 16] suppose the existence of some real $\delta > 0$ such that $(\!| \ell |\!) \geq (\!| r |\!) + \delta$, recovering thus the well-foundedness of the order on natural numbers. Actually, we will show that this kind of extra hypothesis is unnecessary.

In this section, we introduce a deep mathematical result, positivstellensatz, and we show how it can be used to recover inequalities which can play the role of 2, 3. These new inequalities are the key features of Theorems 2 and 4.

Theorem 1 (Stengle [19]). *Suppose that we are given polynomials* $P_1, \ldots, P_m \in \mathbf{R}[x_1, \ldots, x_k]$*, the two following points are equivalent:*

1. $\{x_1, \ldots, x_k : P_1(x_1, \ldots, x_k) \geq 0 \wedge \cdots \wedge P_m(x_1, \ldots, x_k) \geq 0\} = \emptyset$
2. $\exists Q_1, \ldots, Q_m : -1 = \sum_{i \leq m} Q_i P_i$ *where the* Q_i *are sums of squares (and so positive and monotonic).*

We give a first application of the Theorem.

Proposition 1. *Suppose that a TRS* (Σ, R) *admits an interpretation* $(\!|-|\!)$ *over Max-Poly such that for all rules* $\ell \to r$*, we have* $(\!|\ell|\!) > (\!|r|\!)$*. There is a positive, monotonic polynomial* P *such that for all rules* $\ell \to r$*, we have* $(\!|\ell|\!)(x_1, \ldots, x_k) - (\!|r|\!)(x_1, \ldots, x_k) \geq \dfrac{1}{P(\max(x_1, \ldots, x_k))}$.

Proof. For all rules $\ell \to r$, we show below that we can build a monotonic polynomial $P_{\ell \to r}$ such that $(\!|\ell|\!)(x_1, \ldots, x_k) - (\!|r|\!)(x_1, \ldots, x_k) \geq \dfrac{1}{P_{\ell \to r}(x_1, \ldots, x_k)}$. Let us note $Q_{\ell \to r}(x) = P_{\ell \to r}(x, \ldots, x)$. It is then sufficient to take $P = \sum_{\ell \to r \in R} Q_{\ell \to r}$.

Let us consider one rule $\ell \to r$. Let $(\!|\ell|\!)(x_1, \ldots, x_k) = \max_{i \in I}(P_i(x_1, \ldots, x_k))$ and $(\!|r|\!)(x_1, \ldots, x_k) = \max_{j \in J}(Q_j(x_1, \ldots, x_k))$. So, for all j, we have the inequality $(\!|\ell|\!)(x_1, \ldots, x_k) > Q_j(x_1, \ldots, x_k)$. Suppose that we find for all these j a polynomial R_j such that

$$(\!|\ell|\!)(x_1, \ldots, x_k) \geq Q_j(x_1, \ldots, x_k) + \frac{1}{R_j(x_1, \ldots, x_k)}. \tag{4}$$

Then for all j, we have $(\!|\ell|\!)(x_1, \ldots, x_k) \geq Q_j(x_1, \ldots, x_k) + \dfrac{1}{\sum_{j \in J} R_j(x_1, \ldots, x_k)}$.

And so, $(\!|\ell|\!)(x_1, \ldots, x_k) \geq \max_{j \in J}(Q_j(x_1, \ldots, x_k)) + \dfrac{1}{\sum_{j \in J} R_j(x_1, \ldots, x_k)}$. Consequently, we define $P_{t,u}(x_1, \ldots, x_k) = \sum_{j \in J} R_j(x_1, \ldots, x_k)$.

So, it remains to find the polynomials R_j for all $j \in J$. Take one of these j. We define D_i to be the set $\{(x_1, \ldots, x_k) \in (\mathbf{R}^+)^k : \forall i \neq \ell : P_i(x_1, \ldots, x_k) \geq P_\ell(x_1, \ldots, x_k)\}$. On D_i, $(\!|\ell|\!)(x_1, \ldots, x_k) = P_i(x_1, \ldots, x_k)$. And so, on D_i, we have $P_i(x_1, \ldots, x_k) > Q_j(x_1, \ldots, x_k)$.

As a consequence, for all $i \in I$, the set

$$\left\{\begin{array}{l} x_1 \geq 0, \ldots, x_k \geq 0, \\ P_i(x_1, \ldots, x_k) - P_1(x_1, \ldots, x_k) \geq 0, \ldots, P_i(x_1, \ldots, x_k) - P_m(x_1, \ldots, x_k) \geq 0, \\ Q_j(x_1, \ldots, x_k) - P_i(x_1, \ldots, x_k) \geq 0 \end{array}\right\}$$

is empty. And then, by Positivstellensatz, we can state that there are positive polynomials $T_{1,i}, \dots, T_{k+m+1,i}$ such that

$$\begin{aligned}
&-1 = x_1 T_{1,i} + \cdots x_k T_{k,i} + \\
&(P_i(x_1, \dots, x_k) - P_1(x_1, \dots, x_k))T_{k+1,i} + \cdots + \\
&(P_i(x_1, \dots, x_k) - P_m(x_1, \dots, x_k))T_{k+m,i} + \\
&(Q_j(x_1, \dots, x_k) - P_i(x_1, \dots, x_k))T_{k+1+m,i}.
\end{aligned}$$

Observe that on D_i, we have

$$(P_i(x_1, \dots, x_k) - Q_j(x_1, \dots, x_k))T_{k+1+m,i} \geq 1.$$

So that we define $R_j = \sum_{1 \leq i \leq m} T_{k+1+m,i}$.

Proposition 1 has an important consequence. Since, in a derivation all terms have an interpretation bounded by the interpretation of the first term, there is a minimal decay at each step of the derivation.

Proposition 2. *Suppose that a TRS (Σ, R) admits a strict interpretation $(\!| - |\!)$ over Max-Poly. For all $A > 0$, the set of terms $\{t \in \mathcal{T}(\Sigma) \mid (\!| t |\!) < A\}$ is finite.*

Proof. For all symbols $f \in \Sigma$, we have $(\!| f |\!)(x_1, \dots, x_n) > x_i$ for all i. By 1, there is a polynomial P such that $(\!| f |\!)(x_1, \dots, x_n) \geq x_i + \dfrac{1}{P(x_1, \dots, x_n)}$. Take a term $f(t_1, \dots, t_n)$ such that $(\!| f(t_1, \dots, t_n) |\!) < A$.

$$\begin{aligned}
(\!| f(t_1, \dots, t_n) |\!) &\geq (\!| t_i |\!) + \frac{1}{P((\!| t_1 |\!), \dots, (\!| t_n |\!))} \\
&\geq (\!| t_i |\!) + \frac{1}{P(A, \dots, A)}
\end{aligned}$$

where the second inequality is due to the sub-term property together with the monotonicity of P. Consequently, the height of a term t with $(\!| t |\!) < A$ is bounded by $A \times P(A, \dots, A)$. There are only finitely many such terms.

Proposition 3. *Suppose that a TRS (Σ, R) admits a strict interpretation $(\!| - |\!)$ over Poly. There is a real $A > 0$ and a positive, monotonic polynomial P such that for all $x_1, \dots, x_n \geq 0$, if $x_{i_1}, \dots, x_{i_k} > A$, then for all symbol f, we have $(\!| f |\!)(x_1, \dots, x_n) \geq x_{i_1} + \cdots + x_{i_k} + \dfrac{1}{P((\!| f |\!)(x_1, \dots, x_n))}$.*

Proof. Due to Corollary 2, there is a bound A such that for all $x_1, \dots, x_n \geq 0$, if $x_{i_1}, \dots, x_{i_k} > A$, then for all symbol f, we have $(\!| f |\!)(x_1, \dots, x_n) > x_{i_1} + \cdots + x_{i_k}$. Applying Theorem 1, we get Q_f such that $(\!| f |\!)(x_1, \dots, x_n) \geq x_{i_1} + \cdots + x_{i_k} + \dfrac{1}{Q_f(x_1, \dots, x_n)} \geq x_{i_1} + \cdots + x_{i_k} + \dfrac{1}{Q_f((\!| f |\!)(x_1, \dots, x_n), \dots, (\!| f |\!)(x_1, \dots, x_n))}$. It is then routine to get a uniform polynomial wrt to all symbols.

3 The role of reals in complexity

We come back to two characterizations of implicit complexity, one of PTIME and one of PSPACE given respectively in [2] and [16]. Using the tools defined in the previous section, we show that the ordering over the real numbers can be used without restrictions in place of the ordering on natural numbers.

3.1 PTIME and strict interpretation

Theorem 2. *Functions computed by* F*-programs with an additive interpretation (over the reals) are exactly* PTIME *functions. For higher tiers, we get respectively* ETIME *for affine programs and* E_2TIME *for multiplicative programs.*

The rest of the section is devoted to the proof of the Theorem for additive programs. We let the reader adapt the proof for affine and multiplicative programs. From now on, we suppose we are given a F-program with a strict interpretation over polynomials. The key idea of the proof is to show that a), the number of steps in a computation remains polynomial in the size of the term rewritten, and b) the size of terms along the computation remains polynomial.

The main difficulty is that the bounds obtained at the last section (corollary 1 and 3) apply on terms with a sufficiently large interpretation. They introduce a value, next called A, above which we have a complete control on the step of computation. Lemma 2 shows that there are only a polynomial number of such steps. So, our approach is to normalize the term according to small 'step' (those below the value A) and then, to apply a 'big' step. Lemma 4 show that there are not so many such 'small' steps.

Lemma 2. *There is a polynomial P and a real $A > 0$ such that for all step $\ell\sigma \to r\sigma$ with $(\!| r\sigma |\!) > A$, then, for all context C, we have $(\!| C[\ell\sigma] |\!) \geq (\!| C[r\sigma] |\!) + \dfrac{1}{P((\!| \ell\sigma |\!))}$.*

Definition 7. *Given a real $A > 0$, we say that the A-size of a closed term t is the number of subterms u of t (including itself) such that $(\!| u |\!) > A$. We note $\mathsf{S}(t)_A$ the A-size of t.*

Lemma 3. *There is a constant A and a polynomial Q for which $\mathsf{S}(t)_A \leq Q((\!| t |\!))$ for all closed terms t. For all $B > A$, $\mathsf{S}(t)_B \leq \mathsf{S}(t)_A \leq Q((\!| t |\!))$.*

For $A > 0$, we say that $t = C[\ell\sigma] \to C[r\sigma] = u$ is an A-step whenever $(\!| r\sigma |\!) > A$. We note such a rewriting step $t \to_{>A} u$. Otherwise, it is an $\leq A$-step, and we note it $t \to_{\leq A} u$. We use the usual $*$ notation for transitive closure. In case we restrict the relation to the call by value strategy, we add "cbv" as a subscript. Take care that an $\to_{\leq A}$-normal form is not necessarily a normal form for $\to$.

Lemma 4. *There is a constant A and a polynomial P such that for all terms t, any call by value derivation $t \to^*_{\leq A, cbv} u$ has length less than $P((\!| t |\!))$.*

Lemma 5. *For constructor terms, we have $(\!| t |\!) \leq \Gamma \times \mathsf{S}(t)$ for some constant Γ.*

Lemma 6. *Let us suppose we are given an additive program with interpretation being polynomials. For a given function symbol f, there is a strategy such that for all constructor terms $t_1, \ldots, t_n$, the derivation length of $f(t_1, \ldots, t_n)$ is bounded by $Q(\max(\mathsf{S}(t_1), \ldots, \mathsf{S}(t_n)))$ where Q is a polynomial.*

Proof. Of Theorem 2 With the strategy defined above, we have seen that the derivation length of a term $f(t_1, \ldots, t_n)$ is polynomial wrt to $\max(\mathsf{S}(t_1), \ldots, \mathsf{S}(t_n))$. For the converse part, we refer the reader to [2] where a proof that PTIMEprograms can be computed by functional programs with strict interpretations over the integers. This proof can be safely used in the present context.

3.2 Dependency Pairs with polynomial interpretation over the reals

Termination by Dependency Pairs is a general method introduced by Arts and Giesl [1]. It puts into light recursive calls.

Suppose $f(t_1, \ldots, t_n) \to C[g(u_1, \ldots, u_n)]$ is a rule of the program. Then, $(F(t_1, \ldots, t_n), G(u_1, \ldots, u_n))$ is a dependency pair where F and G are new symbols associated to f and g respectively. $S(\mathcal{C}, \mathcal{F}, R)$ denotes the program thus obtained by adding these rules. The dependency graph links dependency pairs $(u, v) \to (u', v')$ if there is a substitution σ such that $\sigma(v) \overset{*}{\to} \sigma(u')$ and termination is obtained when there is no cycles in the graph. Since the definition of the graph involves the rewriting relation, its computation is undecidable. In practice, one gives an approximation of the graph which is bigger. Since this is not the issue here, we suppose that we have a procedure to compute this supergraph which we call the dependency graph.

Theorem 3. *[Arts,Giesl [1]] A TRS $(\mathcal{C}, \mathcal{F}, R)$ is terminating iff there exists a well-founded weakly monotonic quasi-ordering $\geq$, where both $\geq$ and $>$ are closed under substitution, such that*

- *$\ell \geq r$ for all rules $\ell \to r$,*
- *$s \geq t$ for all dependency pairs (s, t) on a cycle of the dependency graph and*
- *$s > t$ for at least one dependency pair on each cycle of the graph.*

It is natural to use the polynomial orderings presented above for the quasi-ordering and the ordering of terms. However, the ordering $>$ is not well-founded on $\mathbf{R}$, so that system may not terminate. Here is such an example.

Example 2. Consider the non terminating system:

$$\begin{pmatrix} f(0) \to 0 \\ f(x) \to f(\mathbf{s}(x)) \end{pmatrix}$$

Take $(\!| 0 |\!) = 1$, $(\!| \mathbf{s} |\!)(x) = x/2$. The system has a unique dependency pair $F(x) \to F(\mathbf{s}(x))$ for which we can give the interpretation $(\!| F |\!)(x) = x + 1$.[2]

[2] The interpretation is correct since for all terms t, $(\!| t |\!) > 0$.

One way to avoid these infinite descent is to force the inequalities over reals to be of the form $P(x_1, \dots, x_n) \geq Q(x_1, \dots, x_n) + \delta$ for some $\delta > 0$ (see for instance Lucas's work [14]). Doing so, one gets a well-founded ordering on reals. We propose an alternative approach to that problem, keeping the original ordering of $\mathbf{R}$.

Definition 8. *A weak polynomial*[3] *algebra for a signature Σ consists of monotone polynomials $(\!| f |\!)$ for all symbols in Σ.*

Definition 9. *A $\mathbf{R}$-DP-interpretation for a program $P = (\mathcal{C}, \mathcal{F}, R)$ is weak polynomial algebra $(\!| - |\!)$ for $S(P)$ such that*

1. *there is $\delta > 0$ such that for each n-ary constructor $\mathbf{c}$ with $n > 0$, for all $x_1, \dots, x_n \geq 0$, we have $(\!|\mathbf{c}|\!)(x_1, \dots, x_n) \geq \delta$,*
2. *$(\!|\ell|\!) \geq (\!|r|\!)$ for $\ell \to r \in R$,*
3. *$(\!|s|\!) \geq (\!|r|\!)$ for $(s, r) \in DP(R)$,*
4. *for each dependency pair (s, t) in a cycle, $(\!|s|\!) > (\!|r|\!)$ holds.*

The main difference with say [16] is that we do not ask for the existence of some δ such that $(\!|s|\!) \geq (\!|r|\!) + \delta$ in the last equation. To simplify the proof of Theorem 4, we took $(\!|s|\!) > (\!|t|\!)$ for all dependency pairs in a cycle, and not for only one. We make the conjecture that the theorem holds, even in the standard case: for each cycle, there is a dependency pair (s, t) such that $(\!|s|\!) > (\!|r|\!)$.

Lemma 7. *Suppose that a polynomial P is weakly monotonic in every argument on $(\mathbf{R}^+)^n$. Suppose that it is not constant wrt some variable. Then for any arbitrarily small $\delta > 0$, there is a polynomial P_δ such that for all $x_1 \geq \delta, \dots, x_n \geq \delta$, if $x_i \geq P_\delta(A)$ for some i, then $P(x_1, \dots, x_n) \geq A$.*

Theorem 4. *A program with a $\mathbf{R}$-DP-interpretation is strongly terminating. Moreover, its derivation height is bounded by $2^{2^{2^{O(n)}}}$ with n the size of the input. This bound is tight.*

Proof. Since, in the present terms, the hypothesis of the Theorem 3 do not hold, we come back to its proof and show that we have an extra-ingredient to get the termination property. Actually, we give a presentation of the proof by means of call-tree. This allows us a much more direct evaluation of the derivation height of terms.

Let us consider the call-tree of a term $f(t_1, \dots, t_n)$ for some constructor terms $t_1, \dots, t_n$. The size of the call tree is precisely the derivation length of the call-tree. Suppose that we prove that there is a polynomial P such that the depth of the call tree is polynomial wrt to the interpretation of $F(t_1, \dots, t_n)$. Since the branching of the call-tree is bounded by R the maximal size of the right hand side of rules, the size of the call-tree is bounded by $R^{P((\!|f(t_1,\dots,t_n)|\!))}$. But, since $(\!|F(t_1, \dots, t_n)|\!) \leq 2^{2^{O(|F(t_1,\dots,t_n)|)}}$ by Lemma 1, we have $P((\!|F(t_1, \dots, t_n)|\!)) \leq P(2^{2^{O(\sum_{i=1}^{n} |t_i|+1)}}) = 2^{2^{O(\max_{i=1}^{n} |t_i|)}}$. And, the conclusion follows.

[3] L-polynomial algebra in the terminology of Lucas [14].

So, it remains to prove that the existence of such a polynomial P. This is done by induction on the rank of symbols. But, first, consider a rule $f(\boldsymbol{p_i})\sigma \to C[g(\boldsymbol{e_i})]\sigma$ with f and g of same rank. Then, we have $(\!|F(\boldsymbol{p_i})|\!) > (\!|G(\boldsymbol{e_i})|\!)$. Notice that Proposition 1 applies in the present context for dependency pairs of same rank, so that we can state that $(\!|F(\boldsymbol{p_i})|\!) - (\!|G(\boldsymbol{e_i})|\!) > \dfrac{1}{P(x_1, \ldots, x_n)}$ where x_i are the variables of $F(\boldsymbol{p_i})$. Wlog, we can suppose (possibly by padding arguments) that P is common to all the dependency pairs of equal rank.

Base case Let us consider a symbol h of minimal rank and some constructor terms $t_1, \ldots, t_n$ and finally, let $P = f_1(\boldsymbol{u_1}), \ldots, f_k(\boldsymbol{u_k})), \ldots$ be a path in the call tree of $f(t_1, \ldots, t_n)$. From hypothesis (4) of Definition 9, we can state that $(\!|F_1(\boldsymbol{u_1})|\!) > (\!|F_2(\boldsymbol{u_2})|\!) > \cdots$.

Given a dependency pair given by $\ell = f(\boldsymbol{p_i}) \to C[g(\boldsymbol{e_i})] = r$ with f and g of equal rank, we extract the sequence $f_\psi = (f_{\psi(i)}(\boldsymbol{u_{\psi(i)}}))_{i\in\mathbf{N}}$ from P such that

- $f_{\psi(i)} = f$ and
- $\ell\sigma = f_{\psi(i)}(\boldsymbol{u_{\psi(i)}}) \to C[f_{\psi(i)+1}(\boldsymbol{u_{\psi(i)+1}})] = r$.

Wlog, we can suppose that $(\!|F(\boldsymbol{p_i})|\!)$ varies with $x_1, \ldots, x_m$ and is constant wrt $x_{m+1}, \ldots, x_n$. Let us consider a (possibly empty) set $\theta \subseteq \{1..m\}$. To simplify the readability of the proof, we suppose $\theta = \{k..m\}$.

We extract the sequence $f_\varphi = (f_{\varphi(i)}(\boldsymbol{u_{\varphi(i)}}))_{i\in\mathbf{N}}$ from f_ψ such that

1. for all $j \leq k$, $(\!|\sigma(x_j)|\!) \neq 0$,
2. for all $j \geq k$, $(\!|\sigma(x_j)|\!) = 0$,

where σ is the substitution mention in the construction of f_ψ.

Observe that $(\!|F(\boldsymbol{p_i})|\!)$ varies with $x_1, \ldots, x_k$. And that $(\!|\sigma(x_i)|\!) \geq \delta$ for all $i \leq k$. Consequently, from Lemma 7, we get a polynomial Q such that $(\!|\sigma(x_i)|\!) \leq Q((\!|f_i(\boldsymbol{u_i})|\!))$. But, then, $(\!|F(\boldsymbol{p_i})|\!) \geq (\!|G(\boldsymbol{e_i})|\!) + \dfrac{1}{P(Q((\!|F_i(\boldsymbol{u_i})|\!)), \ldots, Q((\!|F_i(\boldsymbol{u_i})|\!)))} \geq (\!|G(\boldsymbol{e_i})|\!) + \dfrac{1}{P(Q((\!|F_1(\boldsymbol{u_1})|\!)), \ldots, Q((\!|F_1(\boldsymbol{u_1})|\!)))}$.

As a conclusion, we have a polynomial bound on the initial subsequence restricted to the redex of a chosen rule and a particular choice of variable whose interpretation is 0. Since the number of dependency pairs is finite, since the number of choice for the variables is finite (bounded by 2^D where D is the maximal number of variables in a rule), since only (finitely many) constants have interpretation equal to 0, we can say that the length of the sequence P is bounded by $N \times 2^D \times |\mathcal{C}| \times P(Q((\!|F_1(\boldsymbol{u_1})|\!)), \ldots, Q((\!|F_1(\boldsymbol{u_1})|\!)))$.

induction step Suppose f has a higher rank. A path of a call-tree can be decomposed in a sub-path of nodes with function of rank of f, and symbols of smaller rank. The depth of symbols of rank of f can be treated as it has been done in the base case. For symbols of lower rank, one employs the induction. The depth of symbols of different ranks sums, and we get a call-tree of polynomial depth in the interpretation of the initial term.

The bound is tight as shown by the next example.

Example 3. The Quantified Boolean Formula (QBF) problem is PSPACE complete. It consists in determining the validity of a boolean formula with quantifiers over propositional variables. Without loss of generality, we restrict formulae to $\neg, \vee, \exists$. QBF problem is solved by the following program.

$$\begin{array}{rl}
\mathtt{not}(\mathbf{tt}) \to \mathbf{ff} & \mathtt{not}(\mathbf{ff}) \to \mathbf{tt} \\
\mathtt{or}(\mathbf{tt}, x) \to \mathbf{tt} & \mathtt{or}(\mathbf{ff}, x) \to x \\
\mathbf{0} = \mathbf{0} \to \mathbf{tt} & \mathbf{s}(x) = \mathbf{0} \to \mathbf{ff} \\
\mathbf{0} = \mathbf{s}(y) \to \mathbf{ff} & \mathbf{s}(x) = \mathbf{s}(y) \to x = y \\
\mathtt{in}(x, \varepsilon) \to \mathbf{ff} & \mathtt{in}(x, \mathbf{cons}(a, l)) \to \mathtt{or}(x = a, \mathtt{in}(x, l))
\end{array}
\qquad
\begin{array}{r}
(\!| 0 |\!) = (\!| \varepsilon |\!) = 1 \\
(\!| \mathbf{s} |\!)(x) = (\!| \mathbf{Var} |\!)(x) = x + 1 \\
(\!| \mathbf{Not} |\!)(x) = (\!| \mathbf{Or} |\!)(x) = x + 1 \\
(\!| \mathbf{Exists} |\!)(x) = x + 2
\end{array}$$

$$\begin{aligned}
\mathtt{verify}(\mathbf{Var}(x), t) &\to \mathtt{in}(x, t) \\
\mathtt{verify}(\mathbf{Not}(\varphi), t) &\to \mathtt{not}(\mathtt{verify}(\varphi, t)) \\
\mathtt{verify}(\mathbf{Or}(\varphi_1, \varphi_2), t) &\to \mathtt{or}(\mathtt{verify}(\varphi_1, t), \mathtt{verify}(\varphi_2, t)) \\
\mathtt{verify}(\mathbf{Exists}(n, \varphi), t) &\to \mathtt{or}(\mathtt{verify}(\varphi, \mathbf{cons}(n, t)), \mathtt{verify}(\varphi, t)) \\
\mathtt{qbf}(\varphi) &\to \mathtt{verify}(\varphi, \varepsilon)
\end{aligned}$$

They admit the following interpretation :

$$\begin{aligned}
(\!| \mathtt{or} |\!)(x) = (\!| \mathtt{not} |\!)(x) = (\!| \mathtt{qbf} |\!)(x) &= 1 \\
(\!| = |\!)(x, y) = (\!| \mathtt{in} |\!)(x, y) = (\!| \mathtt{verify} |\!)(x, y) &= 1 \\
(\!| \mathrm{NOT} |\!)(x) &= x \\
(\!| \mathrm{OR} |\!)(x, y) = (\!| \mathrm{EQ} |\!)(x, y) &= \max(x, y) \\
(\!| \mathrm{IN} |\!)(x, y) &= x + y \\
(\!| \mathrm{VERIFY} |\!)(x, y) &= 2 \times x + y + 1 \\
(\!| \mathrm{QBF} |\!)(x) &= x + 2
\end{aligned}$$

It is well known that the derivation height of the QBF is exponential wrt the number of nested **Exists** symbols. Since the following program builds an input of double exponential depth, we get the triple exponential bound on the derivation height.

$$\begin{array}{rl}
\mathtt{add}(\mathbf{0}, y) \to y & \mathtt{add}(\mathbf{s}(x), y) \to \mathbf{s}(\mathtt{add}(x, y)) \\
\mathtt{mult}(\mathbf{0}, y) \to \mathbf{0} & \mathtt{mult}(\mathbf{s}(x), y) \to \mathtt{add}(y, \mathtt{mult}(x, y)) \\
\mathtt{dexp}(\mathbf{0}) \to \mathbf{s}(\mathbf{s}(\mathbf{0})) & \mathtt{dexp}(\mathbf{s}''(x)) \to \mathtt{mult}(\mathtt{dexp}(x), \mathtt{dexp}(x)) \\
\mathrm{e}(0) \to \mathbf{tt} & \mathrm{e}(\mathbf{s}(n)) \to \mathbf{Exists}(n, \mathrm{e}(n)) \\
\mathrm{main}(x) \to \mathtt{qbf}(\mathrm{e}(x)) &
\end{array}$$

with interpretations:

$$
\begin{aligned}
(\!|\mathbf{0}|\!) &= 0 \\
(\!|\mathbf{s}|\!)(x) &= x+1 \\
(\!|e|\!)(x) &= 3x \quad (\!|\mathrm{E}|\!)(x) = x \\
(\!|\mathrm{main}|\!)(x) &= (\!|\mathtt{qbf}(\mathrm{e}(x))|\!) \quad (\!|\mathrm{MAIN}|\!)(x) = x \\
(\!|\mathtt{add}|\!)(x,y) &= x+y \quad (\!|\mathrm{ADD}|\!)(x,y) = x+y \\
(\!|\mathtt{mult}|\!)(x,y) &= x \times y \quad (\!|\mathrm{MUL}|\!)(x,y) = x+y \\
(\!|\mathtt{dexp}|\!)(x) &= x+2 \quad (\!|\mathrm{DEXP}|\!)(x) = x+y
\end{aligned}
$$

We observe that the term $\mathrm{main}(\underbrace{\mathbf{s}''\cdots\mathbf{s}''}_{n \text{ times } \mathtt{q}}\mathbf{0})$ rewrites in normal form to the term $\mathbf{Exists}(s^{2^n}(\mathbf{0},\ldots,\mathbf{Exists}(\mathbf{0},\mathbf{tt})\cdots)$ gives the triple exponential derivation height lower bound.

It is not clear whether there are some programs with **R**-DP-interpretation which do not admit interpretations in the sense of Lucas [14]. Nevertheless, there is at least one good point for **R**-DP-interpretation: the logical formulation of the synthesis of **R**-DP-interpretation involves less alternation of quantifiers. Suppose that we are given a program and a fixed degree, the logical formula corresponding to $(\!|\ell|\!) > (\!|r|\!)$ is $\forall x_1,\ldots,x_n > 0 : (\!|\ell|\!)(x_1,\ldots,x_n) > (\!|r|\!)(x_1,\ldots,x_n)$. While for expressing $>_\delta$, we have to write $\exists\delta > 0 : \forall x_1,\ldots,x_n > 0 : (\!|\ell|\!)(x_1,\ldots,x_n) > (\!|r|\!)(x_1,\ldots,x_n)+\delta$. Since the complexity of the QED procedure depends drastically on the number of alternation, we may hope to get thus a better procedure.

With respect to complexity, the proof of Theorem 4 gives us some insight on the cost of a computation. Consider programs with a **R**-DP-interpretation such that any constructor $\mathbf{c}$ has an interpretation of the form $k_{\mathbf{c}} + \sum_i x_i$, in other words, an additive **R**-DP-interpretation in [16]'s terminology. There is a constant K such that for all contructor terms t, $(\!|t|\!) \leq K.|t|$, so that $(\!|f(t_1,\ldots,t_n)|\!)$ is polynomially bounded wrt the size of the (constructor) terms t_i.[4] The proof shows then that the nesting of function calls is itself bounded polynomially. If, furthermore the interpretation of capital functions verify the sub-term property, we can state that the size of arguments remain polynomial. This is an other formulation (and a slightly more general one) of the hypothesis of "bounded recursion call" which can be found in [16]. So that computations can be performed within PSPACE. Since polynomial time can be done with such systems (cf. [2]), and QBF can be simulated, it is then clear that the following Theorem holds:

Theorem 5. *Functions computed by programs*

- *with additive* **R***-DP-interpretations*
- *the interpretation of capital symbols F has the sub-term property*

[4] See [3] for a proof.

are exactly PSPACE *computable functions.*

Proof. The QBF program showed above together with the fact that programs with an **R**-DP-interpretations are closed under polynomial reduction show that such programs correspond exactly to PSPACE.

Acknowledgement The authors would like thank the anonymous referees for their valuable help.

References

1. T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.
2. Guillaume Bonfante, Adam Cichon, Jean-Yves Marion, and Hélène Touzet. Algorithms with polynomial interpretation termination proof. *J. Funct. Program.*, 11(1):33–53, 2001.
3. Guillaume Bonfante, Jean-Yves Marion, and Jean-Yves Moyen. Quasi-interpretations: a way to control resources. *Theoretical Computer Science*, 2009. to appear.
4. A. Cichon and J.-Y. Marion. The light lexicographic path ordering. Technical report, Loria, 2000. Workshop Rule.
5. Nachum Dershowitz. A note on simplification orderings. *Information Processing Letters*, pages 212–215, 1979.
6. Nachum Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, pages 69–115, 1987.
7. Nachum Dershowitz and Jean-Pierre Jouannaud. *Handbook of Theoretical Computer Science vol.B*, chapter Rewrite systems, pages 243–320. 1990.
8. Nao Hirokawa and Georg Moser. Automated complexity analysis based on the dependency pair method. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *IJCAR*, volume 5195 of *Lecture Notes in Computer Science*, pages 364–379. Springer, 2008.
9. Dieter Hofbauer and Clemens Lautemann. Termination proofs and the length of derivations. *Lecture Notes in Computer Science*, 355:167–177, 1988.
10. Hoon Hong and Dalibor Jakus. Testing positiveness of polynomials. *J. Autom. Reasoning*, 21(1):23–38, 1998.
11. G. Huet and D. Oppen. Equations and rewrite rules: A survey. In R. V. Book, editor, *Formal Language Theory: Perspectives and Open Problems*, pages 349–405. AP, 1980.
12. D.S. Lankford. On proving term rewriting systems are noetherien. Technical report, 1979.
13. Salvador Lucas. On the relative power of polynomials with real, rational, and integer coefficients in proofs of termination of rewriting. *Appl. Algebra Eng., Commun. Comput.*, 17(1):49–73, 2006.
14. Salvador Lucas. Practical use of polynomials over the reals in proofs of termination. In *PPDP '07: Proceedings of the 9th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 39–50, New York, NY, USA, 2007. ACM.
15. Jean-Yves Marion and Romain Péchoux. Resource analysis by sup-interpretation. In *FLOPS*, volume 3945 of *Lecture Notes in Computer Science*, pages 163–176. Springer-Verlag, 2006.

16. Jean-Yves Marion and Romain Péchoux. Characterizations of polynomial complexity classes with a better intensionality. In Sergio Antoy and Elvira Albert, editors, *PPDP*, pages 79–88. ACM, 2008.
17. M.-F. Roy S. Basu, R. Pollack. *Algorithms in real algebraic geometry.* Springer, Berlin Heidelberg New York, 2003.
18. Olha Shkaravska, Marko C. J. D. van Eekelen, and Ron van Kesteren. Polynomial size analysis of first-order shapely functions. *CoRR*, abs/0902.2073, 2009.
19. G. Stengle. A nullstellensatz and a positivstellensatz in semialgebraic geometry. *Mathematische Annalen*, 207(2):87–97, 1973.
20. Alfred Tarski. *A Decision Method for Elementary Algebra and Geometry.* University of California Press, 1951. 2nd edition.

A Some preliminary result about polynomials over the reals

Definition 10. *A polynomial $P \in \mathbf{R}[x, y]$ is said over-homogeneous of degree d whenever there is a real $A \geq 0$ and a real $B \geq 1$ such that $\forall x, y \geq A, \forall \lambda > B$: $P(\lambda x, \lambda y) \geq \lambda^d P(x, y)$. We say that P is A, B-over-homogeneous when we want to put the focus on A and B.*

Lemma 8. *Given $P \in \mathbf{R}[x, y]$ of tota degree greater than 2, we suppose that there is some C such that for all $k \geq 0$, all $x, y > C$, we have $P(x, y) \geq 0$. Then, $P_{\geq k}$ is homogeneous of degree k for $k \geq 1$ and $P_{\geq k}(x, y) \geq 0$ with $x, y > C'$ for some C'.*

Theorem 6. *Given a polynomial $P \in \mathbf{R}[x_1, \dots, x_n]$ such that*

(i) $\forall x_1, \dots, x_n \geq 0 : P(x_1, \dots, x_n) > \max(x_1, \dots, x_n)$,
(ii) $\forall x'_i > x_i, x_1, \dots, x_n \geq 0 : P(x_1, \dots, x'_i, x_{i+1}, \dots, x_n) > P(x_1, \dots, x_n)$,

then, there exist $A \geq 0$ such that $P(x_1, \dots, x_n) > \sum_{i=1}^{n} x_i$ whenever for all $1 \leq i \leq n$, we have $x_i > A$.

Using the compactness of $[0..A]^n$, the following hold.

Corollary 1. *Let $P(x_1, \dots, x_n)$ be a polynomial with the hypothesis of Theorem 6. For all subsets $I \subseteq \{1..n\}$, there is a constant A_I such that for all $x_1, \dots, x_n \geq 0$ with $x_i \geq A_I$ for $i \in I$, we have $P(x_1, \dots, x_n) > \sum_{i \in I} x_i$.*

And, as a corollary of the corollary,

Corollary 2. *There is a constant $A' \geq 0$ such that for all $x_1, \dots, x_n \geq 0$ with $x_{i_1}, \dots, x_{i_k} > A'$, we have $P(x_1, \dots, x_n) > x_{i_1} + \dots + x_{i_k}$.*

Notice that compared to the latter corollary 1, the bound is uniform with respect to I.

Theorem 7. *Given a polynomial $P \in \mathbf{R}[x_1, \dots, x_n]$ such that* *(i)* $\forall x_1 \geq 0, \ldots, x_n \geq 0 : P(x_1, \ldots, x_n) > \max(x_1, \ldots, x_n)$,

(ii) $\forall x_1 \geq 0, \ldots, x_n \geq 0 : \dfrac{\partial P}{\partial x_i}(x_1, \ldots, x_n) > 0$ *for all* $i \leq n$,

then, there exist $A > 0$ *such that for all* $\Delta > 0$*, we have* $P(x_1, \ldots, x_i + \Delta, \ldots, x_n) > P(x_1, \ldots, x_n) + \Delta$ *whenever* $x_1, \ldots, x_n > A$.

Corollary 3. *With the hypothesis of the theorem above, we can say that there is a bound* B *such that for all* $x_1, \ldots, x_n \geq 0$*, if* $x_i > B$*, then,* $P(x_1, \ldots, x_i + \Delta, \ldots, x_n) > P(x_1, \ldots, x_n) + \Delta$.

Static Complexity Analysis of Higher Order Programs

James Avery[1], Lars Kristiansen[2], and Jean-Yves Moyen[3⋆]

[1] DIKU, University of Copenhagen
[2] Department of Mathematics, University of Oslo
[3] LIPN - UMR 7030 – CNRS - Université Paris 13 – F-93430 Villetaneuse, France

Abstract. The overall goal of the research presented in this paper is to find automatic methods for static complexity analysis of higher order programs.

1 Introduction and Related Work

In the first part of the paper we consider a first order imperative language. Our method for dealing with this language is reminiscent of the methods of Ben-Amram, Jones and Kristiansen [JK09,KJ05,BAJK08] or of Niggl and Wünderlich [NW06]. The method is also, although the link is not obvious, related to quasi-interpretation and sup-interpretation; see e.g. Bonfante, Marion and Moyen [BMM07] and Marion and Péchoux [MP09].

In the second part of the paper, we lift the method developed in the first part to allow analysis of higher order programs written in a language that is an imperative version of Gödel's T. There have not been much research on automatic complexity analysis of higher order programs. One exception is Benzinger [B04], however, the semi-automatic approach taken in [B04] seems to be very different from our approach. We believe the work presented in this paper suggests how a number of methods, including the powerful method in [BAJK08], can be generalised to deal with higher order programs.

The paper presents work in progress. The main result is still conjecture, and there is much room for simplification, but several examples are given that illustrate why and how the methods should work.

2 Vectors and Matrices over Semirings

Recall that a *semiring* is a set $\mathcal{S}$ together with two internal operations $+$ and $\cdot$, called *addition* and *multiplication*, such that: $+$ is associative, commutative and has a neutral element 0; $\cdot$ is associative and has a neutral element 1; 0 annihilates $\cdot$; $\cdot$ is distributive over $+$.

⋆ This work was partially supported by a grant from Agence Nationale de la Recherche, ref. ANR-08-BLANC-0211-01 (COMPLICE project)

We consider vectors over a semiring $\mathcal{S}$. To make working with sparse representations easier, we associate an index set $\mathcal{I}$ and define a vector as a mapping $V : \mathcal{I} \to \mathcal{S}$. This representation is equivalent to the standard one, in which vectors are represented by $\mathcal{S}$-tuples denoting the coefficients in a linear expansion in a basis. $\mathcal{I}$ might be infinite, but in our analysis, any vector will have only finitely many nonzero entries.

If $i \in \mathcal{I}$ is an index and V is a vector, $V[i]$ denotes the i^{th} component of the vector (that is, V_i in usual notation). Addition and scalar multiplication are canonically lifted from scalars to vectors point-wise: $(V + W)[i] = V[i] + W[i]$ and $(a \cdot V)[i] = a \cdot (V[i])$.

We use the sparse representation $\left(\begin{smallmatrix} x_1 & \cdots & x_n \\ a_1 & \cdots & a_n \end{smallmatrix}\right)$ to denote the vector V, where $V[x_i] = a_i$ for $\vec{a} \in \mathcal{S}$ and $\vec{x} \in \mathcal{I}$, and where $V[y] = 0$ for any $y \in \mathcal{I} \setminus \{x_1, \ldots, x_n\}$. Notice that $()$ then is the zero vector and that

$$\begin{pmatrix} x_1 & \cdots & x_i \\ a_1 & & a_i \end{pmatrix} \oplus \begin{pmatrix} x_{i+1} & \cdots & x_n \\ a_{i+1} & & a_n \end{pmatrix} = \begin{pmatrix} x_1 & \cdots & x_n \\ a_1 & & a_n \end{pmatrix}$$

when $\{x_1, \ldots, x_i\}$ and $\{x_{i+1}, \ldots, x_n\}$ are disjoint. Let V be a vector. We define

$$\begin{pmatrix} x_1 & \cdots & V & \cdots & x_n \\ a_1 & & a_i & & a_n \end{pmatrix} = a_i V \oplus \begin{pmatrix} x_1 & \cdots & x_{i-1} & x_{i+1} & \cdots & x_n \\ a_1 & & a_{i-1} & a_{i+1} & & a_n \end{pmatrix} \qquad (*)$$

This notation will turn out to be convenient. We will use $e\,[x \setminus t]$ to denote the expression e where the free occurrences of x have been replaced by t. By (*),

$$\begin{pmatrix} x_1 & \cdots & x_i & \cdots & x_n \\ a_1 & & a_i & & a_n \end{pmatrix} [x_i \setminus V] = a_i V \oplus \begin{pmatrix} x_1 & \cdots & x_{i-1} & x_{i+1} & \cdots & x_n \\ a_1 & & a_{i-1} & a_{i+1} & & a_n \end{pmatrix} \qquad (**)$$

We will also see that the equality (**) provides a convenient way of defining the product of higher order matrices by certain substitutions.

A matrix over scalars $\mathcal{S}$ and index set $\mathcal{I}$ is seen as a mapping from the indices to vectors (over $\mathcal{S}$ and $\mathcal{I}$). These vectors are the usual column vectors of algebra textbooks. We use $M, A, B, C, \ldots$ to denote matrices, and M_j to denote the vector that the matrix M assigns to the index j. Thus, $M_j[i]$ is a scalar.

We have chosen to use the nonstandard notation $M_j[i]$ in place of the standard notation M_{ij}, the reason being that this notation works much better for higher order. The reader should note that M_j denotes the j^{th} column vector of the matrix M, i.e. M_j^T in standard row-major notation.

Sums and *products* of matrices are defined as usual: $(A+B)_j[i] = A_j[i]+B_j[i]$ and $(A \cdot B)_j[i] = \sum_{k \in \mathcal{I}} A_k[i] \cdot B_j[k]$.

If $\mathcal{S}$ is non-negative, the matrix M is an *upper bound* of the matrix A, and we write $M \geq A$, iff there exists a matrix B such that $M = A + B$. Thus we have a partial ordering of matrices. The ordering symbols $\geq, \leq, >, <$ have their standard meaning with respect to this ordering, and we will use standard terminology, that is, we say that A is greater than B when $A \geq B$, that A is strictly less than B when $A < B$, et cetera.

The *zero matrix* is denoted by $\mathbf{0}$, and $M = \mathbf{0}$ iff $M_j = ()$ for all $j \in \mathcal{I}$. We have $\mathbf{0} + M = M + \mathbf{0} = M$ for any matrix M. The *identity matrix* is denoted by $\mathbf{1}$, and $\mathbf{1}$ by $M = \mathbf{1}$ iff $M_j[i] = 1$ for $i = j$, $M_j[i] = 0$ for $i \neq j$. We have $\mathbf{1} \cdot M = M \cdot \mathbf{1} = M$ for any matrix M. Furthermore, let $M^0 = \mathbf{1}$ and $M^{n+1} = M \cdot M^n$. We define the *closure* of the matrix M, written M^*, by the infinite sum $M^* = \mathbf{1} + M + M^2 + M^3 + \ldots$.

Let $\mathcal{M}_n$ denote the set of $n \times n$ matrices over a semiring. Then the algebraic structure $(\mathcal{M}_n, +, \cdot, \mathbf{0}, \mathbf{1})$ is a semiring. The closure M^* may not exist for every matrix $M \in \mathcal{M}_n$. However, if M^* exists, the identity $M^* = \mathbf{1} + (M \cdot M^*)$ holds.

3 Analysis of a First-Order Programming Language

3.1 The programming language

Syntax and Semantics We consider deterministic imperative programs that manipulate natural numbers held in variables. Each variable stores a single natural number. Our language is an extension of the well-know LOOP language studied in Meyer & Ritchie [MR76] and in several other places. In this language, a function is computable if and only if it is primitive recursive.

Expressions and *sequences* are defined by the following grammar:

$$\begin{array}{lll}
\text{(Variables)} & & ::= \mathtt{X} \\
\text{(Constants)} & & ::= k_n \text{ for each } n \in \mathbb{N} \\
\text{(Operators)} & & ::= \mathtt{op} \\
\text{(Expressions)} & \ni e & ::= \mathtt{op}(e_1, e_2) \mid \mathtt{X} \mid k_n \\
\text{(Sequences)} & \ni s & ::= \varepsilon \mid s_1 \mathtt{;} s_2 \mid \mathtt{X:=}e \mid \mathtt{loop}\ \mathtt{X}\ \{\, s \,\}
\end{array}$$

Sequences will also be called *programs*.

A program is executed as expected from its syntax, so we omit a detailed formalisation. The semantics at first order is the restriction of the high-order semantics depicted in Figure 1. At any execution step, each variable $\mathtt{X}_i$ holds a natural number x_i, and the expressions are evaluated straightforwardly without side effects. There is an unspecified set of operators that are all computable in constant time. Typical operators include `add`, `mul`, `sub` and `max`, whose semantics are respectively addition, multiplication, modified subtraction[4] and maximum. The constant k_n denotes the integer $n \in \mathbb{N}$. The program $\mathtt{loop}\ \mathtt{X}\ \{\, s \,\}$ executes the sequence s in its body x times in a row, where x is the value stored in `X` when the loop starts. `X` may not appear in the body of the loop. The sequence $s_1 \mathtt{;} s_2$ executes first the sequence s_1 followed by the sequence s_2. Programs of the form $\mathtt{X:=}e$ are ordinary assignment statements, and the command ε does nothing. In all, the semantics of the language is straightforward.

Feasible Programs Let s be a program with variables $\{\mathtt{X}_1, \ldots, \mathtt{X}_n\}$. The program execution relation $a_1, \ldots, a_n[s]b_1, \ldots, b_n$ holds iff the variables $\mathtt{X}_1, \ldots, \mathtt{X}_n$ respectively hold the numbers $a_1, \ldots, a_n$ when the execution of s starts and respectively the numbers $b_1, \ldots, b_n$ when the execution terminates. We say that a program s is *feasible* if for any subprogram s' there exist polynomials $p_1, \ldots, p_n$ such that $a_1, \ldots, a_n[s']b_1, \ldots, b_n \ \Rightarrow\ b_i \leq p_i(a_1, \ldots, a_n)$.

[4] That is, $\mathtt{sub}(\mathtt{X}, \mathtt{Y})$ returns 0 if `Y` is greater than `X`.

3.2 Abstract Interpretation

A Particular Semiring We will analyse programs by interpreting expressions as vectors and sequences as matrices. These vectors and matrices will be over

- the index set $k, \mathtt{X}_0, \mathtt{X}_1, \mathtt{X}_2, \ldots$, that is, the set of program variables extended by the index k,
- and the semiring $(\mathbb{N}, \oplus, \otimes, 0, 1)$ where $\oplus$ is the maximum operator, i.e. $a \oplus b = \max(a, b)$, and $\otimes$ is standard multiplication of natural numbers.

Example In this semiring, the closure M^* of a matrix M might not exists. Let A and B be the following matrices over $(\mathbb{N}, \oplus, \otimes, 0, 1)$:

$$A = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 0 & 2 & 1 \end{pmatrix} \qquad B = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 2 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Then we have

- $A^2 = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 2 & 2 & 1 \end{pmatrix}$, $A^3 = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 4 & 2 & 1 \end{pmatrix}$, and $A^4 = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 4 & 2 & 1 \end{pmatrix}$
- $B^2 = \begin{pmatrix} 1 & 0 & 0 \\ 4 & 4 & 0 \\ 0 & 0 & 1 \end{pmatrix}$, $B^3 = \begin{pmatrix} 1 & 0 & 0 \\ 8 & 8 & 0 \\ 0 & 0 & 1 \end{pmatrix}$, and $B^3 = \begin{pmatrix} 1 & 0 & 0 \\ 16 & 16 & 0 \\ 0 & 0 & 1 \end{pmatrix}$

Hence, we see that A^* exists with the value $A^* = \mathbf{1} \oplus A \oplus A^2 \oplus A^3 \oplus \ldots = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 4 & 2 & 1 \end{pmatrix}$ whereas $B^* = \begin{pmatrix} 1 & 0 & 0 \\ \infty & \infty & 0 \\ 0 & 0 & 1 \end{pmatrix}$ is not a matrix over $\mathbb{N}$.

Some Intuition Our goal is to decide whether or not a given program is feasible by analysing the syntax of the program. This feasibility problem is of course undecidable, but we will present a sound computable method in the sense that if the method certifies a program as feasible, the program will indeed be feasible.

Our abstract interpretation $[\![s]\!]$ of a program s will either be undefined or a matrix

$$[\![s]\!] \quad = \quad [\mathtt{X}_1 \mapsto V_1,\ \ldots, \mathtt{X}_n \mapsto V_n]$$

where $\mathtt{X}_1, \ldots, \mathtt{X}_n$ are the variables occurring in s and $V_1, \ldots, V_n$ are vectors over $\mathbb{N}$ and the index set $\{\mathtt{X}_1, \ldots, \mathtt{X}_n\}$.[5] The program s will be feasible if $[\![s]\!]$ is defined, and the vector V_i will contain information about how the output value of $\mathtt{X}_i$ depends on the input values of $\mathtt{X}_1, \ldots, \mathtt{X}_n$. Let $x_1, \ldots, x_n$ be the input values, and let $x'_1, \ldots, x'_n$ be the output values, of respectively $\mathtt{X}_1, \ldots, \mathtt{X}_n$. If $V_i[\mathtt{X}_j] = 0$, then there exists a polynomial independent of x_j such that $x'_i < p$. If $V_i[\mathtt{X}_j] = 1$, then there exists a polynomial q independent of x_j such that $x'_i \leq x_j + q$. Finally, if $V_i[\mathtt{X}_j] > 1$, then there exists a polynomial p dependent of x_j such that $x'_i \leq p$. The exact value of $V_i[\mathtt{X}_j]$ will say something about the degree of the variable x_j in this polynomial: if $V_i[\mathtt{X}_j] = k$, then the degree of x_j will be less or equal to k. If $[\![s]\!]$ is undefined, s might not be a feasible program.

Let us study an example where s is `loop U { X:=add(X,Y); Z:=X }`. Our interpretation of s will be

$$[\![s]\!] = \left[\mathtt{X} \mapsto \begin{pmatrix} \mathtt{XYZU} \\ 1202 \end{pmatrix},\ \mathtt{Y} \mapsto \begin{pmatrix} \mathtt{XYZU} \\ 0100 \end{pmatrix},\ \mathtt{Z} \mapsto \begin{pmatrix} \mathtt{XYZU} \\ 1212 \end{pmatrix},\ \mathtt{U} \mapsto \begin{pmatrix} \mathtt{XYZU} \\ 0001 \end{pmatrix}\right].$$

[5] For simplicity, assume that no constants occurs in s.

This interpretation tells something about the program s. The entry $\mathtt{X} \mapsto \binom{\mathtt{XYZU}}{1202}$ ensures that there exists a polynomial $p(y,u)$ such that the output value of $\mathtt{X}$ is bounded by $x + p(y,u)$, where x, y, u are the input values of respectively $\mathtt{X}, \mathtt{Y}, \mathtt{U}$. We cannot read off an exact polynomial, but we can conclude that such a polynomial exists. Furthermore, we can conclude that the polynomial may be *dependent* of the input values of $\mathtt{Y}$ and $\mathtt{U}$ and is *independent* of the input values of $\mathtt{X}$ and $\mathtt{Z}$. By inspecting the program s, we can see that $p(y,u)$ might be the polynomial $y \times u$, but we cannot deduce this information from the interpretation $[\![s]\!]$. However, we know from $[\![s]\!]$ that there exists *some* polynomial $p(y,u)$ where the degrees of y and u are less or equal to 2. The entry $\mathtt{Y} \mapsto \binom{\mathtt{XYZU}}{0100}$ tells us that the output value of $\mathtt{Y}$ is bounded by the input value of $\mathtt{Y}$ and independent of the input values of the remaining program variables. The interpretation $[\![s]\!]$ yields similar information about the output values of $\mathtt{Z}$ and $\mathtt{U}$.

The interpretation $[\![s]\!]$ will be undefined for any infeasible program s. E.g., the interpretation of the program $\mathtt{loop\ X}\ \{\ \mathtt{Y{:=}mul(Y,Y)}\ \}$ will not be defined as the output value of $\mathtt{Y}$ will not be bounded by a polynomial in the input values of $\mathtt{X}$ and $\mathtt{Y}$. We have

$$\begin{aligned}
[\![\mathtt{Y{:=}mul(Y,Y)}]\!] &= [\mathtt{X} \mapsto \tbinom{\mathtt{XY}}{10}, \mathtt{Y} \mapsto \tbinom{\mathtt{XY}}{02}] \\
[\![\mathtt{Y{:=}mul(Y,Y);\ Y{:=}mul(Y,Y)}]\!] &= [\mathtt{X} \mapsto \tbinom{\mathtt{XY}}{10}, \mathtt{Y} \mapsto \tbinom{\mathtt{XY}}{04}] \\
[\![\mathtt{Y{:=}mul(Y,Y);\ Y{:=}mul(Y,Y);\ Y{:=}mul(Y,Y)}]\!] &= [\mathtt{X} \mapsto \tbinom{\mathtt{XY}}{10}, \mathtt{Y} \mapsto \tbinom{\mathtt{XY}}{08}]
\end{aligned}$$

and so on, but $[\![\mathtt{loop\ X}\ \{\ \mathtt{Y{:=}mul(Y,Y)}\ \}]\!]$ is undefined.

Loop Correction In addition to the standard operators on vectors and matrices defined in Section 2, we need the operation of *loop correction*: For each $\mathtt{X} \in \mathcal{I}$, we define a unary operator $M^{\downarrow\mathtt{X}}$ on a matrix M, by

$$M_j^{\downarrow\mathtt{X}} = \begin{cases} M_j \oplus \binom{\mathtt{X}}{a} & \text{if } a > 1 \\ M_j & \text{otherwise} \end{cases}$$

where $a = \sum_{i \in \mathcal{I}} M_j[i]$. Note that both the closure and the loop correction operator are monotonous, i.e. $M \leq M^*$ (if M^* exists) and $M \leq M^{\downarrow\mathtt{X}}$ for any matrix M and any $\mathtt{X} \in \mathcal{I}$.

The Interpretation Operator We define the interpretation operator $[\![\cdot]\!]$ mapping expressions to vectors and sequences to matrices.

– Interpretations of expressions:
  - for any program variable $\mathtt{X}$, let $[\![\mathtt{X}]\!] = \binom{\mathtt{X}}{1}$
  - for any constant $\mathtt{k}_n$, let $[\![\mathtt{k}_n]\!] = \binom{k}{1}$
  - $[\![\mathtt{op}(e_1, e_2)]\!] = [\![\mathtt{op}]\!]([\![e_1]\!], [\![e_2]\!])$, i.e. the function $[\![\mathtt{op}]\!]$ applied to the arguments $[\![e_1]\!]$ and $[\![e_2]\!]$
  - $[\![\mathtt{add}]\!] = \lambda x \lambda y \binom{x\ y}{1\ 2}$ and $[\![\mathtt{mul}]\!] = \lambda x \lambda y \binom{x\ y}{2\ 2}$ and $[\![\mathtt{sub}]\!] = \lambda x \lambda y \binom{x}{1}$ and $[\![\mathtt{max}]\!] = \lambda x \lambda y \binom{x\ y}{1\ 1}$

– Interpretations of sequences:
  - $[\![s_1\,\mathtt{;}\,s_2]\!] = [\![s_1]\!] \otimes [\![s_2]\!]$
  - $[\![\mathtt{X}\,\mathtt{:=}\,e]\!] = \mathbf{1}^{\mathtt{X}}_{[\![e]\!]}$ where $\mathbf{1}^{\mathtt{X}}_{[\![e]\!]}$ denotes the identity matrix $\mathbf{1}$ where the vector indexed by $\mathtt{X}$ is replaced by the vector $[\![e]\!]$
  - $[\![\varepsilon]\!] = \mathbf{1}$ (the identity matrix)
  - $[\![\mathtt{loop}\ \mathtt{X}\ \{\ s\ \}]\!] = \begin{cases} ([\![s]\!]^*)^{\downarrow \mathtt{X}} & \text{if } [\![s]\!]^* \text{ exists} \\ \text{undefined} & \text{otherwise} \end{cases}$

Example We will compute the the interpretation of the program expression $\mathtt{max}(\mathtt{X}_1, \mathtt{X}_2)$. Recall definition (*) from Section 2, that is, for $\vec{x} \in \mathcal{I}$ and $\vec{a} \in \mathcal{S}$ and any vector V, we have $\begin{pmatrix} x_1 & \cdots & V & \cdots & x_n \\ a_1 & \cdots & a_i & \cdots & a_n \end{pmatrix} = a_i V \oplus \begin{pmatrix} x_1 & \cdots & x_{i-1} & x_{i+1} & \cdots & x_n \\ a_1 & \cdots & a_{i-1} & a_{i+1} & \cdots & a_n \end{pmatrix}$.

$$\begin{aligned}
[\![\mathtt{max}(\mathtt{X}_1, \mathtt{X}_2)]\!] &= \lambda x \lambda y \begin{pmatrix} x & y \\ 1 & 1 \end{pmatrix} \begin{pmatrix} \mathtt{X}_1 \\ 1 \end{pmatrix} \begin{pmatrix} \mathtt{X}_2 \\ 1 \end{pmatrix} = \lambda y \begin{pmatrix} \begin{pmatrix} \mathtt{X}_1 \\ 1 \end{pmatrix} & y \\ 1 & 1 \end{pmatrix} \begin{pmatrix} \mathtt{X}_2 \\ 1 \end{pmatrix} \\
&= \lambda y \left(1 \begin{pmatrix} \mathtt{X}_1 \\ 1 \end{pmatrix} \oplus \begin{pmatrix} y \\ 1 \end{pmatrix}\right) \begin{pmatrix} \mathtt{X}_2 \\ 1 \end{pmatrix} = \lambda y \begin{pmatrix} \mathtt{X}_1 & y \\ 1 & 1 \end{pmatrix} \begin{pmatrix} \mathtt{X}_2 \\ 1 \end{pmatrix} = \begin{pmatrix} \mathtt{X}_1 & \begin{pmatrix} \mathtt{X}_2 \\ 1 \end{pmatrix} \\ 1 & 1 \end{pmatrix} \\
&= 1 \begin{pmatrix} \mathtt{X}_2 \\ 1 \end{pmatrix} \oplus \begin{pmatrix} \mathtt{X}_1 \\ 1 \end{pmatrix} = \begin{pmatrix} \mathtt{X}_1 & \mathtt{X}_2 \\ 1 & 1 \end{pmatrix}
\end{aligned}$$

Example Next we compute the the interpretation of the program expression $\mathtt{mul}(\mathtt{sub}(\mathtt{X}_1, \mathtt{X}_2), \mathtt{X}_3)$. We have $[\![\mathtt{mul}(\mathtt{sub}(\mathtt{X}_1, \mathtt{X}_2), \mathtt{X}_3)]\!] =$

$$\lambda x \lambda y \begin{pmatrix} x & y \\ 2 & 2 \end{pmatrix} ([\![\mathtt{sub}(\mathtt{X}_1, \mathtt{X}_2)]\!], [\![\mathtt{X}_3]\!]) = \lambda x \lambda y \begin{pmatrix} x & y \\ 2 & 2 \end{pmatrix} \left(\lambda x \lambda y \begin{pmatrix} x \\ 1 \end{pmatrix} ([\![\mathtt{X}_1]\!], [\![\mathtt{X}_2]\!]), [\![\mathtt{X}_3]\!]\right) =$$

$$\lambda x \lambda y \begin{pmatrix} x & y \\ 2 & 2 \end{pmatrix} \left(\begin{pmatrix} [\![\mathtt{X}_1]\!] \\ 1 \end{pmatrix}, [\![\mathtt{X}_3]\!]\right) = \lambda x \lambda y \begin{pmatrix} x & y \\ 2 & 2 \end{pmatrix} \left(\begin{pmatrix} \begin{pmatrix} \mathtt{X}_1 \\ 1 \end{pmatrix} \\ 1 \end{pmatrix}, \begin{pmatrix} \mathtt{X}_3 \\ 1 \end{pmatrix}\right) =$$

$$\lambda x \lambda y \begin{pmatrix} x & y \\ 2 & 2 \end{pmatrix} \left(\begin{pmatrix} \mathtt{X}_1 \\ 1 \end{pmatrix}, \begin{pmatrix} \mathtt{X}_3 \\ 1 \end{pmatrix}\right) = \begin{pmatrix} \begin{pmatrix} \mathtt{X}_1 \\ 1 \end{pmatrix} & \begin{pmatrix} \mathtt{X}_3 \\ 1 \end{pmatrix} \\ 2 & 2 \end{pmatrix} = \begin{pmatrix} \mathtt{X}_1 & \mathtt{X}_3 \\ 2 & 2 \end{pmatrix}$$

The computation shows that the expression $\mathtt{mul}(\mathtt{sub}(\mathtt{X}_1, \mathtt{X}_2), \mathtt{X}_3)$ is interpreted as the vector $\begin{pmatrix} \mathtt{X}_1 & \mathtt{X}_2 \\ 2 & 2 \end{pmatrix}$. Hence, there exists a polynomial p such that the value of the expression is bounded by $p(\mathtt{X}_1, \mathtt{X}_3)$. Note that the value of the expression indeed also depends on $\mathtt{X}_2$, however, there exists a polynomial bound on this value that does not depend on $\mathtt{X}_2$, e.g. the polynomial $\mathtt{X}_1 \times \mathtt{X}_3$.

Example We will now show how to compute the interpretation of the program $\mathtt{X}_3\,\mathtt{:=}\,\mathtt{mul}(\mathtt{sub}(\mathtt{X}_1, \mathtt{X}_2), \mathtt{X}_3)\,\mathtt{;}\,\mathtt{X}_2\,\mathtt{:=}\,\mathtt{max}(\mathtt{X}_2, \mathtt{X}_3)$. The interpretations of the expressions occurring in the program are computed in the previous examples, and we have

$$\begin{aligned}
&[\![\mathtt{X}_3\,\mathtt{:=}\,\mathtt{mul}(\mathtt{sub}(\mathtt{X}_1, \mathtt{X}_2), \mathtt{X}_3)\,\mathtt{;}\,\mathtt{X}_2\,\mathtt{:=}\,\mathtt{max}(\mathtt{X}_2, \mathtt{X}_3)]\!] = \\
&\qquad [\![\mathtt{X}_3\,\mathtt{:=}\,\mathtt{mul}(\mathtt{sub}(\mathtt{X}_1, \mathtt{X}_2), \mathtt{X}_3)]\!] \otimes [\![\mathtt{X}_2\,\mathtt{:=}\,\mathtt{max}(\mathtt{X}_2, \mathtt{X}_3)]\!] = \\
&\qquad\qquad \mathbf{1}^{\mathtt{X}_3}_{[\![\mathtt{mul}(\mathtt{sub}(\mathtt{X}_1, \mathtt{X}_2), \mathtt{X}_3)]\!]} \otimes \mathbf{1}^{\mathtt{X}_2}_{[\![\mathtt{max}(\mathtt{X}_2, \mathtt{X}_3)]\!]} = \mathbf{1}^{\mathtt{X}_3}_{\begin{pmatrix} \mathtt{X}_1 & \mathtt{X}_3 \\ 2 & 2 \end{pmatrix}} \otimes \mathbf{1}^{\mathtt{X}_2}_{\begin{pmatrix} \mathtt{X}_2 & \mathtt{X}_3 \\ 1 & 1 \end{pmatrix}}
\end{aligned}$$

The computation shows that the interpretation of the program is the product of the the two matrices $\mathbf{1}^{\mathtt{X}_3}_{\left(\begin{smallmatrix}\mathtt{X}_1 & \mathtt{X}_3\\ 2 & 2\end{smallmatrix}\right)}$ and $\mathbf{1}^{\mathtt{X}_2}_{\left(\begin{smallmatrix}\mathtt{X}_1 & \mathtt{X}_3\\ 1 & 1\end{smallmatrix}\right)}$. Recall that $\mathbf{1}^{\mathtt{X}_3}_{\left(\begin{smallmatrix}\mathtt{X}_1 & \mathtt{X}_3\\ 2 & 2\end{smallmatrix}\right)}$ is the identity matrix where the column vector indexed by $\mathtt{X}_3$ is replaced by the vector $\left(\begin{smallmatrix}\mathtt{X}_1 & \mathtt{X}_3\\ 2 & 2\end{smallmatrix}\right)$, that is

$$\mathbf{1}^{\mathtt{X}_3}_{\left(\begin{smallmatrix}\mathtt{X}_1 & \mathtt{X}_3\\ 2 & 2\end{smallmatrix}\right)} = \left(\begin{array}{cccc} & \mathtt{X}_1 & \mathtt{X}_2 & \mathtt{X}_3 \\ \mathtt{X}_1 & 1 & 0 & 2 \\ \mathtt{X}_2 & 0 & 1 & 0 \\ \mathtt{X}_3 & 0 & 0 & 2 \end{array}\right) \quad \text{and} \quad \mathbf{1}^{\mathtt{X}_2}_{\left(\begin{smallmatrix}\mathtt{X}_1 & \mathtt{X}_3\\ 1 & 1\end{smallmatrix}\right)} = \left(\begin{array}{cccc} & \mathtt{X}_1 & \mathtt{X}_2 & \mathtt{X}_3 \\ \mathtt{X}_1 & 1 & 0 & 0 \\ \mathtt{X}_2 & 0 & 1 & 0 \\ \mathtt{X}_3 & 0 & 1 & 1 \end{array}\right)$$

and the product is $\left(\begin{array}{cccc} & \mathtt{X}_1 & \mathtt{X}_2 & \mathtt{X}_3 \\ \mathtt{X}_1 & 1 & 0 & 2 \\ \mathtt{X}_2 & 0 & 1 & 0 \\ \mathtt{X}_3 & 0 & 0 & 2 \end{array}\right) \otimes \left(\begin{array}{cccc} & \mathtt{X}_1 & \mathtt{X}_2 & \mathtt{X}_3 \\ \mathtt{X}_1 & 1 & 0 & 0 \\ \mathtt{X}_2 & 0 & 1 & 0 \\ \mathtt{X}_3 & 0 & 1 & 1 \end{array}\right) = \left(\begin{array}{cccc} & \mathtt{X}_1 & \mathtt{X}_2 & \mathtt{X}_3 \\ \mathtt{X}_1 & 1 & 2 & 2 \\ \mathtt{X}_2 & 0 & 1 & 0 \\ \mathtt{X}_3 & 0 & 2 & 2 \end{array}\right)$. Hence, we have

$$[\![\mathtt{X}_3 \mathtt{:=mul(sub(}\mathtt{X}_1, \mathtt{X}_2\mathtt{)}, \mathtt{X}_3\mathtt{)}\mathtt{;}\ \mathtt{X}_2 \mathtt{:=max(}\mathtt{X}_2, \mathtt{X}_3\mathtt{)}]\!] = \left(\begin{array}{cccc} & \mathtt{X}_1 & \mathtt{X}_2 & \mathtt{X}_3 \\ \mathtt{X}_1 & 1 & 2 & 2 \\ \mathtt{X}_2 & 0 & 1 & 0 \\ \mathtt{X}_3 & 0 & 2 & 2 \end{array}\right)$$

What does this matrix tell us? Let x_1, x_2, x_3 be the numbers stored in $\mathtt{X}_1, \mathtt{X}_2, \mathtt{X}_3$ when the execution of the sequence begins, and let x'_1, x'_2, x'_3 be the values stored in $\mathtt{X}_1$, $\mathtt{X}_2, \mathtt{X}_3$ at execution end. The certificate obtained above implies: (i) $x'_1 \leq x_1$. (ii) $x'_2 \leq x_2 + p(x_1, x_3)$ for some polynomial p; in this case $x'_2 \leq x_2 + (x_1 \times x_3)$. (iii) $x'_3 \leq q(x_1, x_3)$ for some polynomial q; in this case $x'_3 \leq x_1 \times x_3$.

Example The bound on a value computed inside a loop may or may not depend on how many times the loop's body is executed. The loop correction operator adds the appropriate dependence on the iteration variable to the value bounds.

Consider the program $\mathtt{loop\ Z}\ \{\ \mathtt{X:=add(X,Y)}\ \}$. Assume the input value of $\mathtt{Z}$ is z. Any bound on the output value of $\mathtt{X}$ depends on z, as the value held by $\mathtt{X}$ will be increased z times by the content of $\mathtt{Y}$. There exists a bound on the output value of $\mathtt{Y}$ that does not depend on z as $\mathtt{Y}$ is not modified inside the loop. The loop correction operator ensures that this is reflected in the interpretation of $\mathtt{loop\ Z}\ \{\ \mathtt{X:=add(X,Y)}\ \}$. We invite the reader to check that

$$[\![\mathtt{loop\ Z}\ \{\ \mathtt{X:=add(X,Y)}\ \}]\!] = ([\![\mathtt{X:=add(X,Y)}]\!]^*)^{\downarrow \mathtt{Z}} =$$
$$([\mathtt{X} \mapsto \left(\begin{smallmatrix}\mathtt{XY}\\ 12\end{smallmatrix}\right), \mathtt{Y} \mapsto \left(\begin{smallmatrix}\mathtt{Y}\\ 1\end{smallmatrix}\right)]^*)^{\downarrow \mathtt{Z}} = [\mathtt{X} \mapsto \left(\begin{smallmatrix}\mathtt{XY}\\ 12\end{smallmatrix}\right), \mathtt{Y} \mapsto \left(\begin{smallmatrix}\mathtt{Y}\\ 1\end{smallmatrix}\right)]^{\downarrow \mathtt{Z}} = [\mathtt{X} \mapsto \left(\begin{smallmatrix}\mathtt{XYZ}\\ 122\end{smallmatrix}\right), \mathtt{Y} \mapsto \left(\begin{smallmatrix}\mathtt{Y}\\ 1\end{smallmatrix}\right)]$$

Example We will study the sequence $\mathtt{loop\ Y}\ \{\ s\ \}$ where s is the sequence $\mathtt{X}_1 \mathtt{:=add(}\mathtt{X}_1, \mathtt{X}_2\mathtt{)}\mathtt{;}\ \mathtt{X}_2 \mathtt{:=add(}\mathtt{X}_2, \mathtt{X}_3\mathtt{)}$. It is left to the reader to verify that $[\![s]\!] = \left(\begin{array}{ccccc} & \mathtt{X}_1 & \mathtt{X}_2 & \mathtt{X}_3 & \mathtt{Y} \\ \mathtt{X}_1 & 1 & 0 & 0 & 0 \\ \mathtt{X}_2 & 2 & 1 & 0 & 0 \\ \mathtt{X}_3 & 0 & 2 & 1 & 0 \\ \mathtt{Y} & 0 & 0 & 0 & 1 \end{array}\right)$. Now, $[\![s]\!]^*$ exists with $[\![s]\!]^* = \left(\begin{array}{ccccc} & \mathtt{X}_1 & \mathtt{X}_2 & \mathtt{X}_3 & \mathtt{Y} \\ \mathtt{X}_1 & 1 & 0 & 0 & 0 \\ \mathtt{X}_2 & 2 & 1 & 0 & 0 \\ \mathtt{X}_3 & 4 & 2 & 1 & 0 \\ \mathtt{Y} & 0 & 0 & 0 & 1 \end{array}\right)$. Hence we obtain

$$[\![\mathtt{loop\ Y}\ \{\ s\ \}]\!] = [\![s]\!]^{*\downarrow \mathtt{Y}} = \left(\begin{array}{ccccc} & \mathtt{X}_1 & \mathtt{X}_2 & \mathtt{X}_3 & \mathtt{Y} \\ \mathtt{X}_1 & 1 & 0 & 0 & 0 \\ \mathtt{X}_2 & 2 & 1 & 0 & 0 \\ \mathtt{X}_3 & 4 & 2 & 1 & 0 \\ \mathtt{Y} & 0 & 0 & 0 & 1 \end{array}\right)^{\downarrow \mathtt{Y}} = \left(\begin{array}{ccccc} & \mathtt{X}_1 & \mathtt{X}_2 & \mathtt{X}_3 & \mathtt{Y} \\ \mathtt{X}_1 & 1 & 0 & 0 & 0 \\ \mathtt{X}_2 & 2 & 1 & 0 & 0 \\ \mathtt{X}_3 & 4 & 2 & 1 & 0 \\ \mathtt{Y} & 4 & 2 & 0 & 1 \end{array}\right)$$

Now, let x_1, x_2, x_3, y be the numbers stored in respectively $\mathtt{X}_1, \mathtt{X}_2, \mathtt{X}_3, \mathtt{Y}$ when the execution of $\mathtt{loop\ Y}\ \{\ s\ \}$ starts; and let x'_1, x'_2, x'_3, y' be the numbers stored in

$\mathtt{X}_1, \mathtt{X}_2, \mathtt{X}_3, \mathtt{Y}$ when the execution ends. The computed certificate guarantees that (i) $x_1' \leq x_1 + p(x_2, x_3, y)$ for some polynomial p, e.g. $x_1' \leq x_1 + y^3 \times x_2 \times x_3$, (ii) $x_2' \leq x_2 + q(x_3, y)$ for some polynomial q, e.g. $x_2' \leq x_2 + y \times x_3$, and (iii) $x_3' \leq x_3$ and $y' \leq y$.

Example We shall study the sequence `loop Y` $\{ s \}$ where s is the sequence $\mathtt{X}_1\mathtt{:=add(X}_1\mathtt{,X}_2\mathtt{);\ X}_2\mathtt{:=add(X}_2\mathtt{,X}_2\mathtt{)}$. One can easily calculate $[\![s]\!] = \left(\begin{array}{cccc} & \mathtt{X}_1 & \mathtt{X}_2 & \mathtt{Y} \\ \mathtt{X}_1 & 1 & 0 & 0 \\ \mathtt{X}_2 & 2 & 2 & 0 \\ \mathtt{Y} & 0 & 0 & 1 \end{array}\right)$. This is the matrix B from Example 1, for which the closure B^* does not exist. Hence, $[\![\mathtt{loop\ Y}\ \{ s \}]\!]$ is undefined, and we cannot certify the program. This is as desired, since some of the output values of the program are not polynomially bounded in the input values. Namely $x_2' = x_2 \times 2^y$.

Theorem 1. *Assume that $[\![s]\!] = M$ for some matrix M. Then, for $i = 1, \ldots, n$, there exists a polynomial p_i such that $\vec{a}[s]\vec{b} \Rightarrow b_i \leq \max(\vec{u}) + p_i(\vec{v})$ where a_j is in the list $\vec{u}$ iff $M_{\mathtt{X}_i}[\mathtt{X}_j] = 1$; and a_j is in the list $\vec{v}$ iff $M_{\mathtt{X}_i}[\mathtt{X}_j] > 1$.*

This is an adaptation of the result of Jones & Kristiansen [JK09].

Corollary 1. *If $[\![s]\!]$ is defined, then s is feasible.*

3.3 Comments and Comparisons with Related Work

The *mwp*-analysis presented in [JK09], can be seen as an abstract interpretation method where first-order imperative programs are interpreted as matrices over a finite semiring. The method presented above builds on the insights of *mwp*-analysis and is, in certain respects, an improvement of *mwp*-analysis:

– By interpreting operators as λ-expressions over vectors, we can easily include any operator in our programming language. We have included `add`, `mul`, `sub` and `max`, and it is straightforward to extend this list, e.g. we can include e.g. `div` (integer division) by the interpretation $[\![\mathtt{div}]\!] = \lambda x \lambda y \left(\begin{smallmatrix} x \\ 1 \end{smallmatrix} \right)$. In contrast, the original *mwp*-analysis only admits the operators `add` and `mul`.
– We provide a technique for dealing with constants. The *mwp*-analysis in [JK09] assumes there are no constants in the programs.
– We interpret programs as matrices over an infinite semiring, and our matrices contains more information than the finite ones in [JK09]. When an *mwp*-matrix is assigned to a program, we know that there exists polynomial upper bounds on the output values of the program; the present analysis provides bounds on the degrees these polynomial bounds.

These improvements of *mwp*-analysis are side effects of our effort to construct an interpretation method which is suitable for lifting to a higher order setting. The reader should also be aware that our method in certain respects is weaker than *mwp*-analysis, which again is weaker than the method introduced in [BAJK08]. This is acceptable since our motivation is not to capture as many first-order algorithms as possible. Rather, it is to develop a theory and book-keeping framework that can easily be lifted to higher orders.

Hopefully, lifting the interpretation presented in this section to higher orders will be a useful and educating exercise. Eventually, we might embark on more difficult projects, like e.g. lifting the analysis in [BAJK08].

4 Analysis of a Higher-Order Programming Language

4.1 The Higher-Order Programming Language

Types We will now extend our first order programming language to a higher order language. This language has variables of any type; variables of type ι hold natural numbers and variables of type σ, where $\sigma \neq \iota$, hold compound types. Expressions are typed in a straightforward way by a classical typing system. We leave out the typing rules here as they are quite obvious. Product types are used for typing pairs, arrow types for procedures, and so forth. The notation $\sigma_1, \sigma_2, \ldots, \sigma_n \to \tau$ is shorthand for $\sigma_1 \to (\sigma_2 \to (\ldots(\sigma_n \to \tau)\ldots))$. E.g., a function from $\mathbb{N}^3$ into $\mathbb{N}$ will be of the type $\iota, \iota, \iota \to \iota$ which is shorthand for the type $\iota \to (\iota \to (\iota \to \iota))$. When needed, we will use superscript to denote the type of an expression e, that is, e^σ is an expression of type σ.

Syntax *Types*, *Expressions* and *sequences* are defined by the following grammar:

$$\begin{array}{lrl}
\text{(Types)} & \ni \sigma, \tau & ::= \iota | \sigma \otimes \tau | \sigma \to \tau \\
\text{(Variables)} & & ::= \mathtt{X} \\
\text{(Constants)} & & ::= k_n \text{ for each } k \in \mathbb{N} \\
\text{(Operators)} & & ::= \mathtt{op} \\
\text{(Expressions)} \ni & e & ::= \mathtt{X} | k_n | \mathtt{op}(e_1, e_2) | \mathtt{pair}(e_1, e_2) | \mathtt{fst}(e) | \mathtt{snd}(e) | \\
 & & \quad \mathtt{app}(e_1, e_2) \mid \{s\}_{\mathtt{X}} \mid \mathtt{proc(X)}\ e \\
\text{(Sequences)} \quad \ni & s & ::= \varepsilon \mid s_1 \mathtt{;} s_2 \mid \mathtt{X} \mathtt{:=} e \mid \mathtt{loop}\ \mathtt{X}^\iota\ \{\ s\ \}
\end{array}$$

As before, the set of operators is unspecified, but apply only to first order (i.e. with expressions of type ι). The expression $\{s\}_{\mathtt{X}}$ returns the value of X after executing the sequence s, while all other variables are left unmodified (i.e. local copies are created and discarded as needed). It is mostly used as the body of procedure definitions, thus leading to a call by value mechanism.

The high-order language extends the first order LOOP language by using high-order variables. It is close to the extension done by Crolard & al. [CPV09]. It can be verified that the resulting language is an imperative variant of Gödel's System T. Our transformation is easier than the one by Crolard & al., because we do not require the lock-step simulation of System T.

Semantics Programs are expressions of type ι whose only free variables are of type ι. They are evaluated by a standard call by value semantics depicted on Figure 1.

An *environment* env maps variables to expressions. Updating an environment is written env$\{\mathtt{X} \mapsto e\}$, env$\{\mathtt{X} \mapsto \emptyset\}$ is used to unbind X when building λ-abstraction for procedures. Operators are evaluated via a given semantics

$$\frac{}{\texttt{X}, \mathrm{env} \vdash \mathrm{env}(\texttt{X})}\ (\textsc{Var}) \qquad \frac{}{k_n, \mathrm{env} \vdash k_n}\ (\textsc{Cons}) \qquad \frac{e_1, \mathrm{env} \vdash^* e_1' \quad e_2, \mathrm{env} \vdash^* e_2'}{\texttt{op}(e_1, e_2), \mathrm{env} \vdash \overline{\texttt{op}}(e_1', e_2')}\ (\textsc{Op})$$

$$\frac{e, \mathrm{env}\{\texttt{X} \mapsto \emptyset\} \vdash^* e'}{\texttt{proc(X)}\ e, \mathrm{env} \vdash \texttt{proc(X)}\ e'}\ (\textsc{Proc}) \qquad \frac{s, \mathrm{env} \Vdash \mathrm{env}'}{\{s\}_{\texttt{X}}, \mathrm{env} \vdash \mathrm{env}'(\texttt{X})}\ (\textsc{Return})$$

$$\frac{e_1, \mathrm{env} \vdash^* \texttt{proc(X)}\ e_1' \quad e_2, \mathrm{env} \vdash^* e_2' \quad e_1'\mathrm{env}\{\texttt{X} \mapsto e_2'\} \vdash^* e'}{\texttt{app}(e_1, e_2), \mathrm{env} \vdash e'}\ (\textsc{App})$$

$$\frac{}{\varepsilon, \mathrm{env} \Vdash \mathrm{env}}\ (\textsc{Empty}) \qquad \frac{s_1, \mathrm{env} \Vdash \mathrm{env}' \quad s_2, \mathrm{env}' \Vdash \mathrm{env}''}{s_1\,\texttt{;}\,s_2, \mathrm{env} \Vdash \mathrm{env}''}\ (\textsc{Comp})$$

$$\frac{s\,\texttt{;}\ldots\texttt{;}\,s, \mathrm{env} \Vdash \mathrm{env}'}{\texttt{loop X}\ \{\ s\ \}, \mathrm{env} \Vdash \mathrm{env}'}\ (\textsc{Loop}) \qquad \frac{e, \mathrm{env} \vdash^* e'}{\texttt{X} := e, \mathrm{env} \Vdash \mathrm{env}\{\texttt{X} \mapsto e'\}}\ (\textsc{Assign})$$

Fig. 1. Call by value semantics

function $\overline{\texttt{op}}^{\iota,\iota\to\iota}$. $\vdash^*$ is the reflexive transitive closure of $\vdash$. Rule (Proc) reduces inside the procedure's body, thus leading to a static binding of variables. Rule (Loop) expands the loop to env(X) copies of its body.

4.2 Higher Order Vectors and Matrices

Vectors In order to separate the programs and program's expressions from their interpretation (as matrices, vectors, and algebraic expressions), we consider a base type κ for algebraic expressions.

We have first order indices $x_0^\kappa, x_1^\kappa, x_2^\kappa, \ldots$, and $\mathcal{I}^\kappa$ denotes the set of first order indices. Let $\mathcal{S}$ be any semiring. We define the arrow types $\alpha \to \beta$ and product types $\alpha \otimes \beta$ over the base type κ in the standard way. Letters early in the Greek alphabet $\alpha, \beta, \gamma \ldots$ t denote types over the base type κ whereas $\sigma, \tau, \ldots$ denote types over the base type ι. For any type α, we have a set of indices $\mathcal{I}^\alpha = \{x_0^\alpha, x_1^\alpha, x_2^\alpha, \ldots\}$. $\mathcal{I}$ denotes the set of all index sets.

Next, we lift the standard addition operator on vectors to an operator on algebraic expressions of higher types. We define the addition operator $+_\alpha$ over the type α recursively over the structure of the type α.

- $a +_\kappa b = a + b$ (standard addition of vectors over $\mathcal{S}$ and $\mathcal{I}^\kappa$)
- $c = a +_{\alpha\to\beta} b$ iff $c(i) = a(i) +_\beta b(i)$ for every $i \in \alpha$
- $(a_1, a_2)\ +_{\alpha\otimes\beta}\ (b_1, b_2) = (a_1 +_\alpha b_1,\ a_2 +_\beta b_2)$

We define the *higher order vector expressions* over $\mathcal{S}$ and the index set $\mathcal{I}$ as the set containing the first order vectors (indexed by $\mathcal{I}^\kappa$), variables (i.e. indexes of $\mathcal{I}$) and which is closed by addition, λ-abstraction, application and pairing. If e^β is an expression and x^α is a free index in e, then $e[x \setminus t^\alpha]$ denotes the expression resulting from replacing all free occurrences of x in e by t. To improve the readability, we will write $e[x_1 \setminus t_1, \ldots, x_n \setminus t_n]$ in place of $e[x_1 \setminus t_1] \ldots [x_n \setminus t_n]$.

Matrices A *higher order matrix* over $\mathcal{S}$ and $\mathcal{I}$ is a mapping from $\mathcal{I}$ into the higher order vector expressions over $\mathcal{S}$ and $\mathcal{I}$ such that each index of type α is mapped to an expression of type α. We use $M, A, B, C, \ldots$ to denote higher order matrices, and M_x denotes the expression to which M maps the index x. The *unity matrix* matrix $\mathbf{1}$ is the matrix such that $\mathbf{1}_x = \left(\begin{smallmatrix} x \\ 1 \end{smallmatrix}\right)$ when $x \in \mathcal{I}^\kappa$ is an index of the base type κ; and $\mathbf{1}_x = x$ when $x \in \mathcal{I} \setminus \mathcal{I}^\kappa$. The *sum* $A{+}B$ of the matrices A and B is defined by

$$M = A{+}B \quad \Leftrightarrow \quad M_x = A_x +_\sigma B_x$$

for any index x of type σ. The matrix A is an *upper bound* of the matrix B, written $A \geq B$, if there exists a matrix C such that $A = B{+}C$. Thus we have a partial ordering of the higher order matrices. The ordering symbols $\geq, \leq, >, <$ have their standard meaning with respect to this ordering. The *product* $A{\cdot}B$ of the matrices A and B is defined by

$$M = A{\cdot}B \quad \Leftrightarrow \quad M_x = B_x[y_1 \setminus A_{y_1}, \ldots, y_n \setminus A_{y_n}]$$

where $y_1, \ldots, y_n$ are the free indices in B_x.

It is worth noticing that this multiplication extends the usual matrix multiplication in the sense that the product, as defined above, of two first-order matrices (i.e. whose high order indexes are mapped to 0) is the same as the usual matrix product.

We can now lift the closure operator defined in Section 2 to higher order matrices. We define the closure M^* of the higher order matrix M by the infinite sum $M^* = \mathbf{1} + M^1 + M^2 + M^3 + \ldots$ where $M^1 = M$ and $M^{n+1} = M{\cdot}M^n$.

We will use the sparse representation $\left(\begin{smallmatrix} X_1 \mapsto e_1 \\ \vdots \\ X_n \mapsto e_n \end{smallmatrix}\right)$ to denote the higher order matrix where the index X_1 maps to the vector expression e_1, the index X_2 maps to the vector expression e_2, and so on. Furthermore, any index Y not occurring in the list $X_1, \ldots, X_n$ maps to Y.

4.3 Interpretation of Programs

Let $\mathcal{S}$ be the semiring $(\mathbb{N}, \oplus, \otimes, 0, 1)$ defined in Section 3.2. We map the types over ι to the types over κ by $\overline{\iota} = \kappa$, $\overline{\sigma \rightarrow \tau} = \overline{\sigma} \rightarrow \overline{\tau}$, and $\overline{\sigma \otimes \tau} = \overline{\sigma} \otimes \overline{\tau}$. Let the index set $\mathcal{I}$ be the set of program variables, any variable $\mathtt{X} \in \mathcal{I}^\sigma$ will serve as an index of type $\overline{\sigma}$. We will interpret program expressions and program sequences as respectively vector expressions and matrices over $\mathcal{S}$ and $\mathcal{I}$. Before we can define the interpretation operator $[\![\cdot]\!]$, we need to lift the loop correction operator from Section 3.2 to higher order matrices: the function $\Phi^\alpha : \mathcal{I}^\kappa \times \alpha \longrightarrow \alpha$ is given by

- $\Phi^\kappa(x,V) \;=\; \begin{cases} V \oplus \binom{x}{a} \text{ if } a > 1 & \text{where } a = \sum_{y \in \mathcal{I}^\kappa} V[y] \\ V \quad\quad \text{otherwise} \end{cases}$
- $\Phi^{\alpha\to\beta}(x,W) \;=\; \lambda U^\alpha \Phi^\beta(x, W(U))$
- $\Phi^{\alpha\otimes\beta}(x,W) \;=\; \langle \Phi^\alpha(x,W_0), \Phi^\beta(x,W_1)\rangle$, where W_0 and W_1 respectively denote the left and the right component of the pair W

When M is a high order matrix, then $M^{\downarrow x}$ is the matrix A where $A_y = \Phi^\alpha(x, M_y)$ for each $y^\alpha \in \mathcal{I}$. We say that the loop correction $M^{\downarrow x}$ takes place with respect to the first order index x. Next, we define the interpretation operator $[\![\cdot]\!]$.

- Interpretations of expressions:
  - for any program variable $\mathtt{X}^\iota$ let $[\![\mathtt{X}]\!] = \binom{\mathtt{X}}{1}$
  - for any program variable $\mathtt{X}^\sigma$ where $\sigma \neq \iota$ let $[\![\mathtt{X}]\!] = x^{\overline{\sigma}}$
  - $[\![\mathtt{proc(X)}\, e]\!] = \lambda x [\![e]\!]$
  - $[\![\mathtt{app}(e_1,e_2)]\!] = [\![e_1]\!]([\![e_2]\!])$
  - $[\![\mathtt{pair}(e_1,e_2)]\!] = \langle [\![e_1]\!], [\![e_1]\!]\rangle$
  - $[\![\mathtt{fst}(e)]\!] = \ell$ where ℓ is the first component of the pair $[\![e]\!]$
  - $[\![\mathtt{snd}(e)]\!] = \ell$ where ℓ is the second component of the pair $[\![e]\!]$
  - $[\![\mathtt{add}]\!] = \lambda x^\kappa \lambda y^\kappa \left(\begin{smallmatrix} x & y \\ 1 & 2 \end{smallmatrix}\right)$ $\quad [\![\mathtt{mul}]\!] = \lambda x \lambda y \left(\begin{smallmatrix} x & y \\ 2 & 2 \end{smallmatrix}\right)$
  - $[\![\mathtt{sub}]\!] = \lambda x \lambda y \left(\begin{smallmatrix} x \\ 1 \end{smallmatrix}\right)$ $\quad [\![\mathtt{max}]\!] = \lambda x \lambda y \left(\begin{smallmatrix} x & y \\ 1 & 1 \end{smallmatrix}\right)$
  - $[\![\{s\}_{\mathtt{X}}]\!] = [\![s]\!]_{\mathtt{X}}$
- Interpretations of sequences:
  - $[\![s_1\,\mathtt{;}\,s_2]\!] = [\![s_1]\!]\cdot[\![s_2]\!]$
  - $[\![x\,\mathtt{:=}\,e]\!] \;=\; (\mathbf{1}^x_{[\![e]\!]})$ where $\mathbf{1}^x_{[\![e]\!]}$ denotes the identity matrix $\mathbf{1}$ where the vector indexed by x is replaced by the vector $[\![e]\!]$
  - $[\![\varepsilon]\!] = \mathbf{1}$ (the identity matrix)
  - $[\![\mathtt{loop\ X}\ \{\, s \,\}]\!] \;=\; \begin{cases} ([\![s]\!]^*)^{\downarrow \mathtt{X}} & \text{if } [\![s]\!]^* \text{ exists} \\ \text{undefined otherwise} \end{cases}$

5 Conjectures

A sequence s is a *program* iff, for some $n > 0$, the first order variables $\mathtt{X}_1^\iota, \ldots, \mathtt{X}_n^\iota$ are the only free variables occurring in s.

The program execution relation $a_1,\ldots,a_n[s]b_1,\ldots,b_n$ holds if and only if the variables $\mathtt{X}_1,\ldots,\mathtt{X}_n$ respectively hold the numbers $a_1,\ldots,a_n$ when the execution of s starts and the numbers $b_1,\ldots,b_n$ when the execution terminates; that is, the program execution relation is defined exactly as for first-order programs.

Conjecture 1. Let $2^x_0 = x$ and $2^x_{\ell+1} = 2^{2^x_\ell}$, and let s be a program. Assume that there exists a matrix M such that $[\![s]\!] = M$. Then, for $i = 1,\ldots,n$, there exists $\ell \in \mathbb{N}$ such that

$$\vec{a}[s]\vec{b} \;\Rightarrow b_i \;\leq \max(\vec{u}) + 2^{\max(\vec{v},1)}_\ell$$

where a_j is in the list $\vec{u}$ iff $M_{\mathtt{X}_i}[\mathtt{X}_j] = 1$; and a_j is in the list $\vec{v}$ iff $M_{\mathtt{X}_i}[\mathtt{X}_j] \geq 2$.

Conjecture 2. The existence of M with $[\![s]\!] = M$ is decidable.

We will now give examples that justify the conjectures. However, no proofs are provided.

6 Justifying the Conjectures

In our higher order language we can write a program p of the form

$$\texttt{F:=proc(f}^{\iota\to\iota}\texttt{) proc(Z}^{\iota}\texttt{) \{loop Z \{ f:=proc(U}^{\iota}\texttt{) ff(U) \}\}}_{\texttt{f}}\texttt{; X}^{\iota}\texttt{:=F(g,Y)(Z)}$$

where g is some expression of type $\iota \to \iota$, that is, a function $\mathbb{N} \to \mathbb{N}$. To improve the readability, we will write `f(X)` in place of `app(f,X)`, `F(g,Y)` in place of `app(app(F,g),Y)`, et cetera. The program assigns a functional to `F` such that $\mathtt{F}(\mathtt{g}, n)(\mathtt{X}) = \mathtt{g}^{2^n}(\mathtt{X})$, that is, the expression $\mathtt{F}(\mathtt{g}, k_n)$ executes the function `g` repeatedly 2^n times on its input argument and returns the result. Thus, the last command of the program assigns to the numeric variable `X` the result of applying a function over and over again, $2^{\mathtt{Y}}$ times, to the value held by `Z`. We have

$$[\![p]\!] \;=\; \lambda f \lambda z(([f \mapsto \lambda u f f \left(\begin{smallmatrix} u \\ 1 \end{smallmatrix}\right)]^*)^{\downarrow z})_f$$

Note that we used `typewriter` font for the program variables while we now use normal font for the high-order algebraic expression variables that interpret them. Now we have:

$$\lambda u f f \left(\begin{smallmatrix} u \\ 1 \end{smallmatrix}\right) [f \setminus \lambda u f f \left(\begin{smallmatrix} u \\ 1 \end{smallmatrix}\right)] \;=\; \lambda u(\lambda u f f \left(\begin{smallmatrix} u \\ 1 \end{smallmatrix}\right) (\lambda u f f \left(\begin{smallmatrix} u \\ 1 \end{smallmatrix}\right) \left(\begin{smallmatrix} u \\ 1 \end{smallmatrix}\right))) \;=\; \lambda u f f f f \left(\begin{smallmatrix} u \\ 1 \end{smallmatrix}\right)$$

Hence,

$$[f \mapsto \lambda u f f \left(\begin{smallmatrix} u \\ 1 \end{smallmatrix}\right)]^2 \;=\; [f \mapsto \lambda u f f \left(\begin{smallmatrix} u \\ 1 \end{smallmatrix}\right)] \cdot [f \mapsto \lambda u f f \left(\begin{smallmatrix} u \\ 1 \end{smallmatrix}\right)] \;=\; [f \mapsto \lambda u f f f f \left(\begin{smallmatrix} u \\ 1 \end{smallmatrix}\right)]$$

By the same token we have $[f \mapsto \lambda u f f \left(\begin{smallmatrix} u \\ 1 \end{smallmatrix}\right)]^3 = [f \mapsto \lambda u f f f f f f f f \left(\begin{smallmatrix} u \\ 1 \end{smallmatrix}\right)]$ and $[f \mapsto \lambda u f f \left(\begin{smallmatrix} u \\ 1 \end{smallmatrix}\right)]^4 = [f \mapsto \lambda u f f f f f f f f f f f f f f f f \left(\begin{smallmatrix} u \\ 1 \end{smallmatrix}\right)]$, etc., and thus

$$\begin{aligned}[f \mapsto \lambda u f f \left(\begin{smallmatrix} u \\ 1 \end{smallmatrix}\right)]^* \;=\; \mathbf{1} +_\sigma [f \mapsto \lambda u f f \left(\begin{smallmatrix} u \\ 1 \end{smallmatrix}\right)] +_\sigma [f \mapsto \lambda u f f \left(\begin{smallmatrix} u \\ 1 \end{smallmatrix}\right)]^2 +_\sigma \ldots \;=\; \\ [f \mapsto \mathbf{1} +_\sigma \lambda u f f \left(\begin{smallmatrix} u \\ 1 \end{smallmatrix}\right) +_\sigma \lambda u f f f f \left(\begin{smallmatrix} u \\ 1 \end{smallmatrix}\right) +_\sigma \ldots]\end{aligned}$$

where $\sigma \equiv \kappa \to \kappa$. If the infinite sum on the right hand side converges to the value A, then

$$[f \mapsto \lambda u f f \left(\begin{smallmatrix} u \\ 1 \end{smallmatrix}\right)]^* \;=\; [f \mapsto A]$$

Now, A is an expression of type $\kappa \to \kappa$, and

$$[f \mapsto \lambda u f f \left(\begin{smallmatrix} u \\ 1 \end{smallmatrix}\right)]^{*\downarrow z} \;=\; [f \mapsto A]^{\downarrow z} \;=\; [f \mapsto \Phi^{\kappa\to\kappa}(z, A)] \;=\; [f \mapsto \lambda u \Phi^{\kappa}(z, Au)]$$

The two last equalities hold by the definition of the loop correction operator. Notice that we are not yet able to compute the actual value of this interpretation. Indeed, we first need to instantiate the λ-abstraction within it before computing. So it is not possible, at this point, to decide, e.g., whether the closure exists. This is expected, as it is the interpretation of the functional `F`, whose behaviour (with respect to the bound on the computed value) depends not only on the first order values, but also on its high-order arguments (namely `f`, being interpreted as the abstracted variable f). Now, we see that

$$[\![p]\!] \;=\; \lambda f \lambda z(([f \mapsto \lambda u f f \left(\begin{smallmatrix} u \\ 1 \end{smallmatrix}\right)]^*)^{\downarrow z})_f \;=\; \lambda f \lambda z(\lambda u \Phi^{\kappa}(z, Au)) \qquad (*)$$

where $A \equiv 1_\sigma +_\sigma \lambda u f f \left(\begin{smallmatrix} u \\ 1 \end{smallmatrix}\right) +_\sigma \lambda u f f f f \left(\begin{smallmatrix} u \\ 1 \end{smallmatrix}\right) +_\sigma \dots$ and

$$\Phi^\kappa(z, Au) \;=\; \begin{cases} Au \oplus \left(\begin{smallmatrix} z \\ a \end{smallmatrix}\right) \text{ if } a > 1 \text{ where } a = \sum_{y \in \mathcal{I}} (Au)[y] \\ Au \quad\quad\quad \text{otherwise} \end{cases}$$

What happens now, if the `g` in our program is the identity function, and what happens if `g` is the doubling function? First, assume that `g` is the identity function, i.e. we have the program

$$s_0 \;\equiv\; \begin{array}{l} \mathtt{F\!:=\!proc(f^{\iota\to\iota})\,proc(Z^\iota)\{loop\;Z\;\{\,f\!:=\!proc(U^\iota)\,ff(U)\,\}\}_f;} \\ \mathtt{X^\iota\!:=\!F(proc(X)X,Y)(Z)} \end{array}$$

Then,

$$[\![s_0]\!] \;=\; \begin{pmatrix} \mathtt{F} \mapsto \lambda f \lambda z(([f \mapsto \lambda u f f \left(\begin{smallmatrix} u \\ 1 \end{smallmatrix}\right)]^*)^{\downarrow z})_f \\ \mathtt{X} \mapsto \lambda f \lambda z(([f \mapsto \lambda u f f \left(\begin{smallmatrix} u \\ 1 \end{smallmatrix}\right)]^*)^{\downarrow z})_f([\![\lambda x.x]\!],[\![y]\!])[\![z]\!] \end{pmatrix}$$

If f is the identity function, that is the function $\lambda x^\kappa \left(\begin{smallmatrix} x \\ 1 \end{smallmatrix}\right)$, then get

$$\begin{aligned} A \equiv 1_\sigma +_\sigma \lambda u f f \left(\begin{smallmatrix} u \\ 1 \end{smallmatrix}\right) +_\sigma \lambda u f f f f \left(\begin{smallmatrix} u \\ 1 \end{smallmatrix}\right) +_\sigma \dots \;=\; \lambda u \lambda x^\kappa \left(\begin{smallmatrix} x \\ 1 \end{smallmatrix}\right)(\lambda x^\kappa \left(\begin{smallmatrix} x \\ 1 \end{smallmatrix}\right)\left(\begin{smallmatrix} u \\ 1 \end{smallmatrix}\right)) \;+_\sigma \\ \lambda u \lambda x^\kappa \left(\begin{smallmatrix} x \\ 1 \end{smallmatrix}\right)(\lambda x^\kappa \left(\begin{smallmatrix} x \\ 1 \end{smallmatrix}\right)(\lambda x^\kappa \left(\begin{smallmatrix} x \\ 1 \end{smallmatrix}\right)(\lambda x^\kappa \left(\begin{smallmatrix} x \\ 1 \end{smallmatrix}\right)\left(\begin{smallmatrix} u \\ 1 \end{smallmatrix}\right)))) \;+_\sigma\; \dots \;=\; \lambda u \left(\begin{smallmatrix} u \\ 1 \end{smallmatrix}\right)\;. \end{aligned}$$

Hence, by (*) we obtain

$$\begin{aligned} &\lambda f \lambda z(([f \mapsto \lambda u f f \left(\begin{smallmatrix} u \\ 1 \end{smallmatrix}\right)]^*)^{\downarrow z})_f([\![\mathtt{proc(X)\;X}]\!],[\![\mathtt{Y}]\!]) \;= \\ &\quad \lambda f \lambda z(\lambda u \Phi^\kappa(z, Au)(\lambda x^\kappa \left(\begin{smallmatrix} x \\ 1 \end{smallmatrix}\right), \left(\begin{smallmatrix} y \\ 1 \end{smallmatrix}\right)) \;=\; \lambda z(\lambda u \Phi^\kappa(z, \lambda u \left(\begin{smallmatrix} u \\ 1 \end{smallmatrix}\right) u)) \left(\begin{smallmatrix} y \\ 1 \end{smallmatrix}\right) \;= \\ &\quad \lambda u \Phi^\kappa(\left(\begin{smallmatrix} y \\ 1 \end{smallmatrix}\right), \left(\begin{smallmatrix} u \\ 1 \end{smallmatrix}\right)) \;=\; \lambda u \begin{cases} \left(\begin{smallmatrix} u \\ 1 \end{smallmatrix}\right) \oplus \left(\begin{smallmatrix} y \\ a \end{smallmatrix}\right) & \text{if } a > 1 \text{ where } a = \sum_{i \in \mathcal{I}} \left(\begin{smallmatrix} u \\ 1 \end{smallmatrix}\right)[i] \\ \left(\begin{smallmatrix} u \\ 1 \end{smallmatrix}\right) & \text{otherwise} \end{cases} \\ &\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad =\; \lambda u \left(\begin{smallmatrix} u \\ 1 \end{smallmatrix}\right) \end{aligned}$$

Finally, we get

$$[\![s_0]\!]_{\mathtt{X}} \;=\; ((\lambda f \lambda z(\lambda u \Phi^\kappa(z, Au))\lambda x^\kappa \left(\begin{smallmatrix} x \\ 1 \end{smallmatrix}\right))\left(\begin{smallmatrix} \mathtt{Y} \\ 1 \end{smallmatrix}\right)\left(\begin{smallmatrix} \mathtt{Z} \\ 1 \end{smallmatrix}\right) \;=\; \lambda u \left(\begin{smallmatrix} u \\ 1 \end{smallmatrix}\right)\left(\begin{smallmatrix} \mathtt{Z} \\ 1 \end{smallmatrix}\right) \;=\; \begin{pmatrix} \mathtt{Z} \\ 1 \end{pmatrix}$$

Thus, the value assigned to the first order index `X` by the interpretation $[\![s_0]\!]$ is the vector $\left(\begin{smallmatrix} \mathtt{Z} \\ 1 \end{smallmatrix}\right)$. This is good. If we inspect the program, we see that if a is held by `Z` when the execution of s_0 starts, and b is held by `X` when the execution terminates, then $b \leq a$.

What, then, will the interpretation of our program look like if we call the functional `F` with the doubling function instead of the identity function? That is, if we have the program

$$s_1 \;\equiv\; \begin{array}{l} \mathtt{F\!:=\!proc(f^{\iota\to\iota})\,proc(Z^\iota)\{loop\;Z\;\{\,f\!:=\!proc(U^\iota)\,ff(U)\,\}\}_f;} \\ \mathtt{X^\iota\!:=\!F(add,Y)(Z)} \end{array}$$

The answer is: The interpretation $[\![s_1]\!]$ does not exist.

Note that $[\![\texttt{proc(X) \{X:=add(X,X)\}}_{\texttt{X}}]\!] = \lambda x \left(\begin{smallmatrix} x \\ 2 \end{smallmatrix}\right)$. Thus, computing $[\![s_1]\!]$ involves computing the infinite sum

$$1_\sigma +_\sigma \lambda u f f \left(\begin{smallmatrix} u \\ 1 \end{smallmatrix}\right) +_\sigma \lambda u f f f f \left(\begin{smallmatrix} u \\ 1 \end{smallmatrix}\right) +_\sigma \lambda u f f f f f f f f \left(\begin{smallmatrix} u \\ 1 \end{smallmatrix}\right) +_\sigma \dots$$

where f is the function $\lambda x \left(\begin{smallmatrix} x \\ 2 \end{smallmatrix}\right)$. We get

$$\begin{aligned} 1_\sigma \ +_\sigma \ & \lambda u \lambda x \left(\begin{smallmatrix} x \\ 2 \end{smallmatrix}\right) (\lambda x \left(\begin{smallmatrix} x \\ 2 \end{smallmatrix}\right) \left(\begin{smallmatrix} u \\ 1 \end{smallmatrix}\right)) +_\sigma \\ & \lambda u \lambda x \left(\begin{smallmatrix} x \\ 2 \end{smallmatrix}\right) (\lambda x \left(\begin{smallmatrix} x \\ 2 \end{smallmatrix}\right) (\lambda x \left(\begin{smallmatrix} x \\ 2 \end{smallmatrix}\right) (\lambda x \left(\begin{smallmatrix} x \\ 2 \end{smallmatrix}\right) \left(\begin{smallmatrix} u \\ 1 \end{smallmatrix}\right)))) \ +_\sigma \ \dots \ = \\ & \lambda u \left(\begin{smallmatrix} u \\ 4 \end{smallmatrix}\right) \ +_\sigma \ \lambda u \left(\begin{smallmatrix} u \\ 16 \end{smallmatrix}\right) \ +_\sigma \ \lambda u \left(\begin{smallmatrix} u \\ 256 \end{smallmatrix}\right) \ +_\sigma \ \dots \end{aligned}$$

This sum does not converge, and hence, the interpretation of program s_1 is undefined.

Now, let us show both the use of the constants k_n and an example where the program is certified, even if the computed values are not polynomial. Let `g` in the program above be the successor function. That is, we have the program s_2 where

$$s_2 \ \equiv \ \begin{array}{l} \texttt{F:=proc(f}^{\iota\to\iota}\texttt{) proc(Z}^{\iota}\texttt{)\{loop Z \{ f:=proc(U}^{\iota}\texttt{) ff(U) \}\}}_{\texttt{f}}\texttt{;} \\ S^{\iota\to\iota}\texttt{:=proc(X)add(X,}k_1\texttt{); X}^{\iota}\texttt{:=F(}S\texttt{,Y)(Z)} \end{array}$$

This program computes the value $\texttt{Z} + 2^{\texttt{Y}}$ and stores it in `X`, and the program has a certificate. Notice first that $[\![S]\!] = \lambda x \left(\begin{smallmatrix} x & k \\ 1 & 2 \end{smallmatrix}\right)$, where k is the special index dedicated to the constants. Computing $[\![s_2]\!]$ involves computing the infinite sum

$$1_\sigma +_\sigma \lambda u f f \left(\begin{smallmatrix} u \\ 1 \end{smallmatrix}\right) +_\sigma \lambda u f f f f \left(\begin{smallmatrix} u \\ 1 \end{smallmatrix}\right) +_\sigma \lambda u f f f f f f f f \left(\begin{smallmatrix} u \\ 1 \end{smallmatrix}\right) +_\sigma \dots$$

where f is the function $\lambda x \left(\begin{smallmatrix} x & k \\ 1 & 2 \end{smallmatrix}\right)$. This sum converges and equals $\lambda u \left(\begin{smallmatrix} u & k \\ 1 & 2 \end{smallmatrix}\right)$. This entails that the program has an interpretation. This certificate has $x \mapsto \left(\begin{smallmatrix} z & y & k \\ 1 & 2 & 2 \end{smallmatrix}\right)$.

Acknowledgements The authors thank Amir Ben-Amram for helpful comments.

References

[BAJK08] A.M Ben-Amram, N.D. Jones, and L. Kristiansen. Linear, polynomial or exponential? Complexity inference in polynomial time. In *CiE'08:Logic and Theory of Algorithms*, volume 5028 of *LNCS*, pages 67–76. Springer, 2008.

[BMM07] G. Bonfante, J.-Y. Marion, and J.-Y. Moyen. Quasi-interpretations – a way to control resources. *Theoretical Computer Science*, 2007. To appear.

[MP09] J.-Y. Marion, and R. Pechoux. Sup-interpretations, a semantic method for static analysis of program resources. *AMS Transactions on Coputational Logic* 10, 2009.

[B04] R. Benzinger. Automated higher-order complexity analysis. *Theoretical Computer Science* 318 (2009), 79-103.

[CPV09] T. Crolard, E. Polonowski, and P. Valarcher. Extending the loop language with higher-order procedural variables. *Special issue of ACM TOCL on Implicit Computational Complexity*, 10(4):1–36, 2009.

[JK09] N. D. Jones and L. Kristiansen. A flow calculus of mwp-bounds for complexity analysis. *ACM Transactions of Computational Logic*, 10, 2009.

[KJ05] L. Kristiansen and N.D. Jones. The flow of data and the complexity of algorithms. In Torenvliet (eds.) Cooper, Lwe, editor, *CiE'05:New Computational Paradigms*, volume 3526 of *LNCS*, pages 263–274. Springer, 2005.

[MR76] A. R. Meyer and D. M. Ritchie. The complexity of loop programs. In *Proc. ACM Nat. Meeting*, 1976.

[NW06] K.-H. Niggl and H. Wunderlich. Certifying polynomial time and linear/polynomial space for imperative programs. *SIAM Journal on Computing*, 35(5):1122–1147, March 2006. published electronically.

Implementing a practical polynomial programming language

Michael J. Burrell[1], Robin Cockett[2], and Brian F. Redmond[2]

[1] mburrel@uwo.ca. Computer Science Department, University of Western Ontario, London, Ontario, Canada, N6A 5B7.

[2] bredmond@ucalgary.ca,robin@cpsc.ucalgary.ca. Department of Computer Science, University of Calgary, Calgary, Alberta, Canada, T2N 1N4.

Abstract. We describe a programming language, Pola, in which the evaluation of every well-typed program halts in time polynomial with respect to the size of its input. It is a functional style language intended as a practical language for writing real-time or embedded system applications in which time and space resources are critical. Pola supports "polynomial" inductive data types, such as trees and lists, but also supports coinductive data types which are lazily evaluated. In addition to a discussion of these data types, the syntax and operational semantics of Pola are provided with particular emphasis on type inference. It is, of course, the type system which enforces the polynomial-time property of the language. Finally, a system for automatically inferring run-time bounds is presented.

1 Introduction

Pola [2] is a functional language, first introduced in [3], wherein every well-typed program halts in polynomial time with respect to the size of its input. It is also polynomial time complete: any polynomial-time Turing machine can be simulated in Pola. While the type discipline required to ensure Pola programs are polynomial time necessarily affects the expressiveness of the language, Pola is designed to allow a natural style of programming which retains as much expressiveness for the programmer as is possible.

The type discipline of Pola allows the compiler to automatically infer run-time and size bounds for programs. This positions Pola well for developing critical real-time or embedded software. Pola can provide absolute guarantees on resource consumption beyond what testing can offer.

Pola grew from the interpretation of the safe recursion of Bellantoni and Cook [1] as a proof theory of a polarized logic [4]. Polarized logics were designed to model games and some of this terminology, such as the "player" and "opponent" worlds (cf. "safe" and "normal"), has been retained in Pola.

Bellantoni and Cook's system of safe recursion, which only considered binary natural numbers, was a simplification of an earlier system developed by Leivant [8] which had general inductive data types. Both these systems allowed duplication in the safe world. Pola, however, uses a crucial idea introduced by

Hofmann [6]: Pola's player (or safe) world is affine and is inhabited by purely constant time computations and it is by "iterating" these that one obtains polynomial time.

In order to model Bellantoni and Cook's system, Hofmann found it necessary to allow the duplication of certain types (such as binary numbers) into his safe computations. Furthermore, in [7], Hofmann highlighted various expressivity problems with systems based on predicative recursion. Pola resolves these issues with a novel new programming construct called "peeking" while retaining an affine "player" world wherein every computation is constant time. Not only can one simulate Bellantoni and Cook's system but also Pola offers the ability to perform safe recursion on general inductive data (including Leivant's data) while ensuring every computation halts in polynomial time. In addition, it avoids many of the performance problems described by Colson [5] and offers coinductive data, which provide laziness and further higher-order capabilities.

Pola aims to be foremost a practical language, offering general inductive and coinductive data in a setting with the theoretical benefits of guaranteed termination and run-time inference.

A more theoretical foundation to Pola is provided in [3]; here we focus on practical concerns except where a theoretical exposition is required for understanding.

2 Data types

To introduce Pola it is first necessary to discuss data types: all the computational power of Pola is delivered by recursion over data types.

2.1 Tuple types

Pola offers two tuple types: tensors, denoted $(\alpha_1, \ldots, \alpha_n)$ over types α_i; and products, denoted $(\alpha_1; \cdots; \alpha_n)$. The two sorts of tuples share the same unit, (). The only distinction between the two constructs, made more precise in section 4.2, is in the typing restrictions.

2.2 Inductive data types

The general form for defining a new inductive data type, A, optionally parameterized over type variables $b_1, \ldots, b_k$, is as follows:

$$\begin{aligned} \mathsf{data}\ \mathsf{A}(b_1, \ldots, b_k) \to c = {} & \mathsf{C}_1 : \mathcal{F}_1(c, b_1, \ldots, b_k) \to c \\ & \mid \qquad \vdots \\ & \mid \mathsf{C}_n : \mathcal{F}_n(c, b_1, \ldots, b_k) \to c; \end{aligned}$$

The $\mathcal{F}_i$ are *modalities*: that is, type expressions, built from the type variables $c, b_1, \ldots, b_k$, constant types, and tuple types. The declaration introduces a new inductive data type A together with constructors $\mathsf{C}_1, \ldots, \mathsf{C}_n$. Examples of simple

$$
\begin{array}{l|l}
\begin{aligned}\mathsf{data\ Bool} \to c = {} & \mathsf{False} : \ \to c \\ & \mid \mathsf{True} : \ \to c;\end{aligned} &
\begin{aligned}\mathsf{data\ Nat} \to c = {} & \mathsf{Zero} : \ \to c \\ & \mid \mathsf{Succ} : c \to c;\end{aligned} \\
\begin{aligned}\mathsf{data\ List}(a) \to c = {} & \mathsf{Nil} : \ \to c \\ & \mid \mathsf{Cons} : a, c \to c;\end{aligned} &
\begin{aligned}\mathsf{data\ BNat} \to c = {} & \mathsf{BEnd} : \ \to c \\ & \mid \mathsf{B0} : c \to c; \\ & \mid \mathsf{B1} : c \to c;\end{aligned} \\
\begin{aligned}\mathsf{data\ Tree}(a, b) \to c = {} & \mathsf{Leaf} : a \to c \\ & \mid \mathsf{Node} : c, b, c \to c;\end{aligned} &
\begin{aligned}\mathsf{data\ LTree}(a, b) \to c = {} & \mathsf{LLeaf} : a \to c \\ & \mid \mathsf{LNode} : c; b; c \to c;\end{aligned}
\end{array}
$$

Fig. 1. Examples of common inductive data types.

inductive data types are given in figure 1. Note the use of the type variable c to stand recursively for the type being defined.

Mutually recursive inductive data types can be defined as well. For instance, rose trees (trees in which each node has zero or more children) can be defined as follows:

$$
\begin{aligned}
\mathsf{data\ RoseTree}(a) \to c = {} & \mathsf{Rose} : a, d \to c; \\
\mathsf{and\ RoseList}(a) \to d = {} & \mathsf{RNil} : \ \to d \\
& \mid \mathsf{RCons} : c, d \to d;
\end{aligned}
$$

Inductive data types are sufficient to describe polynomial-time computation, but alone they do not offer an adequate level of expressiveness.

2.3 Coinductive data types

A good intuition for coinductive data is to think of them as objects (in the object-oriented programming sense) where each "destructor" (in Pola's terminology) can be thought of as an object-oriented "method". Coinductive data types capture a collection of computations waiting to happen and thus provide a general framework for describing laziness and closures.

The general form for defining a coinductive data type, A, optionally parameterized over type variables $b_1, \ldots, b_k$, with destructors $\mathsf{D}_1, \ldots, \mathsf{D}_n$, is as follows:

$$
\begin{aligned}
\mathsf{data}\ c \to \mathsf{A}(b_1, \ldots, b_k) = {} & \mathsf{D}_1 : c, \mathcal{F}_1(c, b_1, \ldots, b_k) \to \mathcal{G}_1(c, b_1 \ldots, b_k) \\
& \mid \qquad\qquad\qquad \vdots \\
& \mid \mathsf{D}_n : c, \mathcal{F}_n(c, b_1, \ldots, b_k) \to \mathcal{G}_n(c, b_1 \ldots, b_k);
\end{aligned}
$$

Here, both $\mathcal{F}_i$ and $\mathcal{G}_i$ are modalities. Coinductive data types can be mutually recursive in the same style as inductive data types.

A common usage for coinductive data types is to represent a lazily evaluated infinite list. There are two simple descriptions for an infinite list, as follows:

$$
\begin{aligned}
\mathsf{data}\ c \to \mathsf{InfList}_1(a) = {} & \mathsf{Head} : c \to a \\
& \mid \mathsf{Tail} : c \to c; \\
\mathsf{data}\ c \to \mathsf{InfList}_2(a) = {} & \mathsf{Next} : c \to (a, c);
\end{aligned}
$$

$\mathsf{InfList}_2$ has a single destructor which returns both its head and tail as a tensor pair (allowing both the head and the tail to be subsequently used).

Another common usage for coinductive data types is to provide closure-like objects. This is possible by providing a destructor with more than one argument (analogous to a method with an argument). For instance, the data type of (player) functions with domain a and codomain b can be described as follows:

$$\mathsf{data}\ c \to \mathsf{Fn}(a, b) = \mathsf{Eval} : c, a \to b;$$

Usage and operation of both inductive and coinductive data types will be dealt with in later sections.

3 Abstract syntax

Pola is a functional language and aims to keep its syntax as close to a typical functional language as possible. The unique parts of the syntax pertain to the distinction between the opponent and player worlds, described in section 4.1, the fold and unfold constructs to provide safe recursion and syntax relating to coinductive data types. The syntax is given in figure 2.

At the top level, we have a series of data type or function declarations. Function declarations consist of a list of parameters (split into "opponent" parameters and "player" parameters) followed by a term. Since Pola relies on type inference, described in section 4, there is no need for any explicit type annotations in function declarations, though the programmer is certainly allowed to provide type constraints.

The peek construct is syntactically and semantically the same as a case construct; the difference is in the typing rules. It will be described further in section 4. Branches of case, peek and fold constructs all perform pattern-matching and optionally introduce new variables into scope.

Records create (non-recursive) coinductive objects. For instance, to create a record of type $\mathsf{Fn}(\mathsf{Nat}, \mathsf{Nat})$ (function from the natural numbers to the natural numbers) which adds one to its argument, we could write ($\mathsf{Eval} : x.\mathsf{Succ}(x)$). Other coinductive data types, for instance infinite lists, necessarily need to be described recursively and for that we use the unfold construct. An unfold is syntactically just like a record except that it introduces a function which can be called recursively in a controlled way. The restrictions on this recursive function are dealt with in section 4. Consider the infinite list of all natural numbers:

```
unfold g(x) as (
 Head : x;
 Tail : g(Succ(x))
) in g(Zero)
```

The result of this unfold construct, like all coinductive objects, is lazy. No computation takes place until we destruct it. For instance, if we assign the variable *nats* to the infinite list of natural numbers above, $\mathsf{Head}(nats)$ yields the value Zero and $\mathsf{Head}(\mathsf{Tail}(nats))$ yields the value $\mathsf{Succ}(\mathsf{Zero})$.

$\mathit{Declaration} := \mathsf{data}\ \mathsf{D}(a_1,\dots,a_m) \to c = \{\mathit{Constructor}^+\}$	Inductive data type
$\mathit{Declaration} := \mathsf{data}\ c \to \mathsf{D}(a_1,\dots,a_m) = \{\mathit{Destructor}^+\}$	Coinductive data type
$\mathit{Declaration} := f = x_1,\dots,x_m \mid y_1,\dots,y_n.\mathit{Term}$	Function declaration
$\mathit{Constructor} := \mathsf{C} : \mathit{Modality} \to c$	
$\mathit{Destructor} := \mathsf{D} : c, \mathit{Modality} \to \mathit{Modality}$	
$\mathit{Modality} := x$	Single type variable
$\mathit{Modality} := \mathit{Modality},\dots,\mathit{Modality}$	Tensor
$\mathit{Modality} := \mathit{Modality};\cdots;\mathit{Modality}$	Product
$\mathit{Term} := x$	Variable
$\mathit{Term} := (\mathit{Term},\dots,\mathit{Term})$	Tensor
$\mathit{Term} := (\mathit{Term};\cdots;\mathit{Term})$	Product
$\mathit{Term} := f(\mathit{Term})$	Function call
$\mathit{Term} := \mathsf{C}(\mathit{Term})$	Construction
$\mathit{Term} := \mathsf{D}(\mathit{Term})$	Destruction
$\mathit{Term} := \mathsf{case}\ \mathit{Term}\ \mathsf{of}\ \{\ \mathit{Branch}^+\ \}$	Case
$\mathit{Term} := \mathsf{peek}\ \mathit{Term}\ \mathsf{of}\ \{\ \mathit{Branch}^+\ \}$	Peek
$\mathit{Term} := (\ \mathit{Cobranch}^+\ )$	Record
$\mathit{Term} := \mathsf{fold}\ f(x, \mathit{Modality})\ \mathsf{as}\ \{\ \mathit{Branch}^+\ \}\ \mathsf{in}\ \mathit{Term}$	Fold
$\mathit{Term} := \mathsf{unfold}\ f(\mathit{Modality})\ \mathsf{as}\ (\ \mathit{Cobranch}^+\ )\ \mathsf{in}\ \mathit{Term}$	Unfold
$\mathit{Branch} := \mathsf{C}(\mathit{VarPatterns}).\mathit{Term}$	Constructor pattern
$\mathit{Branch} := (\mathit{VarPatterns}).\mathit{Term}$	Tuple projection
$\mathit{Cobranch} := \mathsf{D} : \mathit{VarPatterns}.\mathit{Term}$	Destructor
$\mathit{VarPatterns} := x$	Single Variable
$\mathit{VarPatterns} := \mathit{VarPatterns},\dots,\mathit{VarPatterns}$	Tensor projection
$\mathit{VarPatterns} := \mathit{VarPatterns};\cdots;\mathit{VarPatterns}$	Product projection

Fig. 2. Abstract syntax of Pola.

General recursion is not allowed in Pola, instead recursion is controlled by the fold construct. A fold is a case construct in the context of a recursive function in the same way that an unfold construct is a record construct in the context of a recursive function – though the typing restrictions are different. For example, consider the fold expression which adds two natural numbers, m and n:

$$\begin{array}{l} \mathit{add} = n \mid m.\mathsf{fold}\ f(x,y)\ \{ \\ \ \mathsf{Zero}.y; \\ \ \mathsf{Succ}(z).\mathsf{Succ}(f(z,y)) \\ \}\ \mathsf{in}\ f(n,m); \end{array}$$

This fold construct is a case, performing pattern matching, on the variable x in the context of the recursive function f. The reason for separating the parameters of the function, n and m, by a vertical bar is described in section 4.1.

4 Type inference

Proper type inference is key to the operation of Pola since it is correct typing that ensures every function runs in polynomial time.

4.1 Opponent and player worlds

Variables in Pola live in one of two worlds: the opponent world and the player world. For both types and patterns, a vertical bar separates variables of the two worlds. For instance, the *add* function defined in section 3 would have type $add :: \mathsf{Nat} \mid \mathsf{Nat} \mapsto \mathsf{Nat}$ to read that it takes one parameter (n) in the opponent world and one parameter (m) in the player world.

The opponent world has no restriction on how many times variables can be referenced. Also, a variable can be brought from the opponent world to the player world, but not the other way around. The first argument to a fold construct (n in the *add* function)—the argument which drives the recursion—must be in the opponent world.

Variables in the player world may not be duplicated. That is, once introduced, they may only be referenced once. An apparent exception to this is the peek construct, which will be explained later.

Usually we shall refer to the opponent environment as Γ and the player environment as Δ or Σ.

4.2 Sequent notation

Global symbols, such as constructors, destructors or previously defined functions, are added to the opponent context, Γ. A type inference sequent is of the form $\Gamma \mid \Delta \mapsto t : \alpha$, read as the inferring of Pola term t, assigned type variable α, in context $(\Gamma \mid \Delta)$. Γ is an opponent context (mapping from symbols to types), Δ is a player context, t is a term and α is a type. The opponent context is a set of symbols paired with types; ordering is unimportant. The player context is a bunched context, described below. Premises to a rule can either be sequents or of the form $\Gamma \Rightarrow (f : \beta)$, signifying that a look-up of f is done in the global environment Γ, but that no inference is being done and that β is not a type variable which needs to be unified. Note that constructors will be generally of type $(\mathsf{C} : \quad \mid \alpha \mapsto \beta)$ to say that they have no opponent parameters. A rule of the following form stands to mean, given premises $A_1, \ldots, A_k$, sequent B can be proved if type equations $\zeta_1, \ldots, \zeta_m$ are satisfied:

$$\frac{A_1 \quad \cdots \quad A_k}{B} \; \zeta_1, \ldots, \zeta_m$$

The player context is a bunched context, i.e., represented as a tree ([11, 10]). In BNF form, the player context Δ is defined as follows: $\Delta ::= \emptyset \mid x : \alpha \mid \Delta, \Delta \mid \Delta; \Delta$. I.e., it is either the empty context, a single mapping between variable and type, a tensor (separated by commas) or product (separated by semicolons). Both tensors and products are commutative and associative. When constructing a tensor, player contexts must be disjoint; when projecting from a tensor, all elements of the tensor may be used. Conversely, player contexts can be shared when constructing a product, but elements must be kept disjoint when projecting from a product.

We use $\Sigma[\Delta]$ to signify that Δ appears somewhere in the bunched context Σ. Note that in some cases, such as in the inference rule for tensor, we consider a player context to be split into two or more subcontexts. For brevity and simplicity we state in the rules here that the player context is split correctly, as if non-deterministically. For non-bunched contexts an efficient solution, passing around a context and removing variables as they are used[3], is only a minor modification. For bunched logics a mechanism for efficiently splitting contexts is less clear and is elided here.

4.3 Simple terms

Figure 3 defines rules of type inference for simple terms in Pola.

Type equations in Pola can be quantified either existentially or universally. Existentially quantified type equations are analogous to type equations in other Hindley-Milner type inference type systems [12], where $\exists\alpha.\alpha = \beta$ is to be read as finding the most general type for β satisfying the equation in the context of some fresh type variable α. They should not be confused with existentially quantified types as described in other typing systems. Universally quantified types differ from usual type variables and are used in sections 4.4 and 4.5.

4.4 Terms on inductive data types

Rules of inference for terms on inductive data types are given in figure 4. The rules given here gloss over the issues involved with polymorphic types. A constructor, Cons, for instance, would reasonably have type $\Lambda\alpha.\ \ \mid \alpha, \mathsf{List}(\alpha) \mapsto \mathsf{List}(\alpha)$ and thus each time it is used it needs to be instantiated with new type variables. To avoid complicating what are otherwise already crowded rules of inference, we assume we already have Cons specific to whatever type we are interested in, e.g., $\mathsf{Cons} :\ \ \mid \mathsf{Nat}, \mathsf{List}(\mathsf{Nat}) \mapsto \mathsf{List}(\mathsf{Nat})$. Treatment of polymorphism is elided since it is not the focus of these rules.

The simplest term on inductive data types is the construction, in which case we look up the constructor C in the global environment and then perform inference on the arguments to the constructor.

The case construct does pattern matching on terms strictly in the opponent world. This is enforced by ensuring that inference on the subject of the case, t, happens with an empty player environment. Variables $\tilde{x}_i$ introduced by pattern-matching in the case are therefore introduced into the opponent environment. Note that the $\tilde{x}_i$ variables may be a product or tensor, or some combination thereof, but this multiplicative structure is ignored when brought into the "flat" opponent environment since the opponent environment does not need to maintain any affineness.

[3] In practice it is better to mark them as "already used" rather than have them removed entirely, in the interest of the compiler providing useful error messages.

$$\frac{}{x : \beta, \Gamma \mid \Delta \mapsto x : \alpha}\ \alpha = \beta \qquad \text{Opponent variable}$$

$$\frac{}{\Gamma \mid \Sigma[x : \beta] \mapsto x : \alpha}\ \alpha = \beta \qquad \text{Player variable}$$

$$\frac{[\Gamma \mid \Delta_i \mapsto t_i : \beta_i]_{i=1,\ldots,n}}{\Gamma \mid \Delta_1, \ldots, \Delta_n \mapsto (t_1, \ldots, t_n) : \alpha}\ \exists \beta_1, \ldots, \beta_n . \alpha = (\beta_1, \ldots, \beta_n) \qquad \text{Tensor}$$

$$\frac{[\Gamma \mid \Delta \mapsto t_i : \beta_i]_{i=1,\ldots,n}}{\Gamma \mid \Delta \mapsto (t_1; \cdots; t_n) : \alpha}\ \exists \beta_1, \ldots, \beta_n . \alpha = (\beta_1; \cdots; \beta_n) \qquad \text{Product}$$

$$\frac{\begin{array}{l}\Gamma \Rightarrow (f : \beta \mid \gamma \mapsto \eta) \\ \Gamma \mid \ \ \mapsto t : \beta' \\ \Gamma \mid \Delta \mapsto u : \gamma'\end{array}}{\Gamma \mid \Delta \mapsto f(t|u) : \alpha}\ \exists \beta', \gamma' . \beta = \beta', \gamma = \gamma', \alpha = \eta \qquad \begin{array}{l}\text{Function calls} \\ (t \text{ opponent}, u \text{ player})\end{array}$$

$$\frac{\begin{array}{l}\Gamma \mid \Delta \mapsto t : \beta \\ \Gamma \mid \Sigma[(x_1 : \gamma_1, \ldots, x_n : \gamma_n); \Delta] \mapsto u : \alpha\end{array}}{\Gamma \mid \Sigma[\Delta] \mapsto \mathsf{peek}\ t\ \mathsf{of}\ (x_1, \ldots, x_n).u : \alpha}\ \exists \beta, \gamma_1, \ldots, \gamma_n . \beta = (\gamma_1, \ldots, \gamma_n) \qquad \text{Tensor projection}$$

$$\frac{\begin{array}{l}\Gamma \mid \Delta \mapsto t : \beta \\ \Gamma \mid \Sigma[x_1 : \gamma_1; \cdots; x_n : \gamma_n; \Delta] \mapsto u : \alpha\end{array}}{\Gamma \mid \Sigma[\Delta] \mapsto \mathsf{peek}\ t\ \mathsf{of}\ (x_1; \cdots; x_n).u : \alpha}\ \exists \beta, \gamma_1, \ldots, \gamma_n . \beta = (\gamma_1; \cdots; \gamma_n) \qquad \text{Product projection}$$

$$\frac{\Gamma \mid \Delta \mapsto u : \beta \quad \Gamma \mid \Sigma[x : \beta; \Delta] \mapsto t : \alpha}{\Gamma \mid \Sigma[\Delta] \mapsto t\ \mathsf{where}\ x := u : \alpha}\ \exists \beta. \qquad \text{Player where}$$

$$\frac{\Gamma \mid \ \ \mapsto u : \beta \quad x : \beta, \Gamma \mid \Delta \mapsto t : \alpha}{\Gamma \mid \Delta \mapsto t\ \mathsf{where}\ x = u : \alpha}\ \exists \beta. \qquad \text{Opponent where}$$

Fig. 3. Rules of type inference for simple Pola terms.

The peek construct is similar to the case with the most notable exception being that the subject, t, can be in the player world[4] and accordingly the variables $\tilde{x}_i$ are introduced in the player world. However, there is one other complication aimed at allowing the programmer more expressiveness: if the variables introduced are not used, the original variables used by t may be reused. Intuitively this corresponds to the idea of "peeking" at the result of t without actually using the result of t and thus t can still not be duplicated in general. We use product $((\cdots); \Delta)$ to indicate that Δ may be reused if and only if the other variables introduced are not. There is a further restriction not stated in the rule in that

[4] By "can be in the player world" we mean that type inference on the term happens in the context of the player environment. In contrast, the subject, t, of a case is "in the opponent world", i.e., without any player environment.

$$\frac{\mathit{\Gamma} \Rightarrow \beta = [(\mathsf{C}_i :\ \mid \tilde{\gamma}_i \mapsto c)]_{i=1,\ldots,n} \quad \mathit{\Gamma} \mid \mathit{\Delta} \mapsto t : \tilde{\gamma}'_j}{\mathit{\Gamma} \mid \mathit{\Delta} \mapsto \mathsf{C}_j(t) : \alpha}\ \exists \tilde{\gamma}'_j . \tilde{\gamma}'_j = \tilde{\gamma}_j, \alpha = \beta \qquad \text{Construction}$$

$$\frac{\begin{array}{l} \mathit{\Gamma} \mid \ \mapsto t : \beta \\ \mathit{\Gamma} \Rightarrow \delta = [(\mathsf{C}_i :\ \mid \tilde{\gamma}_i \mapsto c)]_{i=1,\ldots,n} \\ [\tilde{x}_i : \tilde{\gamma}_i[\delta/c], \mathit{\Gamma} \mid \mathit{\Delta} \mapsto u_i : \alpha]_{i=1,\ldots,n} \end{array}}{\mathit{\Gamma} \mid \mathit{\Delta} \mapsto \mathsf{case}\ t\ \mathsf{of}\ \{\cdots \mathsf{C}_i(\tilde{x}_i).u_i; \cdots\} : \alpha}\ \exists \beta . \beta = \delta \qquad \text{Case}$$

$$\frac{\begin{array}{l} \mathit{\Gamma} \mid \mathit{\Delta} \mapsto t : \beta \\ \mathit{\Gamma} \Rightarrow \delta = [(\mathsf{C}_i :\ \mid \tilde{\gamma}_i \mapsto c)]_{i=1,\ldots,n} \\ [\mathit{\Gamma} \mid \mathit{\Sigma}[\tilde{x}_i : \tilde{\gamma}_i[\delta/c]; \mathit{\Delta}] \mapsto u_i : \alpha]_{i=1,\ldots,n} \end{array}}{\mathit{\Gamma} \mid \mathit{\Sigma}[\mathit{\Delta}] \mapsto \mathsf{peek}\ t\ \mathsf{of}\ \{\cdots \mathsf{C}_i(\tilde{x}_i).u_i; \cdots\} : \alpha}\ \exists \beta . \beta = \delta \qquad \text{Peek}$$

$$\frac{\begin{array}{l} (f : \beta \mid \tilde{\eta} \mapsto \alpha), \mathit{\Gamma} \mid \mathit{\Delta} \mapsto v : \alpha \\ \mathit{\Gamma} \Rightarrow \delta = [(\mathsf{C}_i :\ \mid \tilde{\gamma}_i \mapsto c)]_{i=1,\ldots,n} \\ \left[\begin{array}{c|c} (f :\ \mid \zeta, \tilde{\eta} \mapsto \alpha), & \tilde{x}_i : \tilde{\gamma}_i[\zeta/c], \\ \tilde{x}'_i : \tilde{\gamma}_i[\delta/c], \mathit{\Gamma} & \tilde{y} : \tilde{\eta} \end{array} \mapsto u_i : \alpha\right]_{i=1,\ldots,n} \end{array}}{\mathit{\Gamma} \mid \mathit{\Delta} \mapsto \mathsf{fold}\ f(x, \tilde{y})\ \mathsf{as}\ \{\cdots \mathsf{C}_i(\tilde{x}'_i | \tilde{x}_i).u_i; \cdots\}\ \mathsf{in}\ v : \alpha}\ \begin{array}{c} \exists \beta, \tilde{\eta}. \forall \zeta. \\ \beta = \delta \end{array} \qquad \text{Fold}$$

Fig. 4. Rules of inference for Pola terms on inductive types.

the variables of t may only be reused if t does not contain a destruction or a recursive function call; otherwise, super-polynomial time evaluations can arise[5].

Note that a data type may introduce the $\tilde{x}_i$ variables as tensor, product, or any combination thereof. Tensor variables are introduced into $\mathit{\Delta}$ as tensors and product variables are introduced as products, as expected.

The fold construct is, in essence, a case construct in the context of a recursive function, f, though it requires more care in typing. The fold is the only construct, other than the unfold dealt with in section 4.5, which makes use of universally quantified types. A universal type variable *cannot* be unified with any other type. The mechanics of this are described in section 4.6. In the opponent environment, $\tilde{x}'_i$ variables are introduced with the concrete type of the subject of the fold δ substituted for c. In the player context, $\tilde{x}_i$ variables are introduced with the universal type ζ substituted for c. For instance, in the branch $\mathsf{Cons}(h'|h, t'|t).u$ where the subject of the fold is of type $\mathsf{List}(\mathsf{Nat})$, $h' : \mathsf{Nat}$, $h : \mathsf{Nat}$, $t' : \mathsf{List}(\mathsf{Nat})$ and $t : \zeta$. Since the first parameter of the recursive function f is of type ζ and ζ cannot be unified with any other type, this means that *only* variables introduced by pattern matching of the appropriate type can be used to make a recursive call.

[5] In fact, relaxing this restriction does not affect the polynomial space bound and permits the encoding of QBF, a well-known PSPACE-complete problem [2]. It would be interesting to compare this system with Leivant and Marion's system for polyspace given in [9].

Further, since the $x_{i,j}$ variables are introduced in the player context, they may not be duplicated, and hence at most one recursive call per variable is allowed.

To more clearly see how typing works in fold constructions, consider the following example of a fold over a list of booleans, which returns the second last element of the list l:

```
fold f(x, y) as {
 Nil.y;
 Cons(z|−, zs′|zs).f(zs, case zs′ of {
  Nil.y;
  Cons(−, −).z })
} in f(l, True)
```

The type of the entire expression, and the type of each branch, is $\alpha = \mathsf{Bool}$. The type being folded over is $\beta = \mathsf{List}(\mathsf{Bool})$. The type of the only argument to the fold is $\eta_1 = \mathsf{Bool}$, corresponding to the variable y. No variables are introduced in the Nil branch, but in the Cons branch we have $z : \mathsf{Bool}$ and $zs' : \mathsf{List}(\mathsf{Bool})$ being introduced to the opponent context and $zs : \zeta$ added to the player context. Since zs is the only variable in context of type ζ, it is the only variable that can be used as the first argument in a recursive call to f.

4.5 Terms on coinductive data types

Rules of inference relating to coinductive types are given in figure 5.

$$\frac{\begin{array}{l}\mathit{\Gamma} \Rightarrow \beta = [(\mathsf{D}_i : \ \mid c, \tilde{\gamma}_i \mapsto \delta_i)]_{i=1,\dots,n} \\ [\mathit{\Gamma} \mid \tilde{x}_i : \tilde{\gamma}_i, \mathit{\Delta} \mapsto t_i : \delta'_i]_{i=1,\dots,n}\end{array}}{\mathit{\Gamma} \mid \mathit{\Delta} \mapsto (\cdots \mathsf{D}_i : \tilde{x}_i.t_i; \cdots) : \alpha} \ \begin{array}{c}\exists \delta'_i . \delta'_i = \delta_i[\beta/c], \\ \alpha = \beta\end{array} \qquad \text{Record}$$

$$\frac{\begin{array}{l}(g : \ \mid \tilde{\eta} \mapsto \alpha), \mathit{\Gamma} \mid \mathit{\Delta}_1 \mapsto u : \alpha \\ \mathit{\Gamma} \Rightarrow \beta = [(\mathsf{D}_i : c, \tilde{\gamma}_i \mapsto \delta_i)]_{i=1,\dots,n} \\ \big[(g : \ \mid \tilde{\eta} \mapsto \zeta), \mathit{\Gamma} \ \big| \ \tilde{y} : \tilde{\eta}, \tilde{x}_i : \tilde{\gamma}_i, \mathit{\Delta}_2 \mapsto t_i : \delta_i[\zeta/c]\big]_{i=1,\dots,n}\end{array}}{\mathit{\Gamma} \mid \mathit{\Delta}_1, \mathit{\Delta}_2 \mapsto \mathsf{unfold}\ g(\tilde{y})\ \mathsf{as}\ (\cdots \mathsf{D}_i : \tilde{x}_i : t_i; \cdots)\ \mathsf{in}\ u : \alpha} \ \begin{array}{c}\exists \tilde{\eta}. \forall \zeta . \\ \alpha = \beta\end{array} \ \text{Unfold}$$

$$\frac{\begin{array}{l}\mathit{\Gamma} \Rightarrow \beta = [(\mathsf{D}_i : \ c, \tilde{\gamma}_i \mapsto \delta_i)]_{i=1,\dots,n} \\ \mathit{\Gamma} \mid \mathit{\Delta}_1 \mapsto r : \beta' \\ \mathit{\Gamma} \mid \mathit{\Delta}_2 \mapsto t : \tilde{\gamma'}_j\end{array}}{\mathit{\Gamma} \mid \mathit{\Delta}_1, \mathit{\Delta}_2 \mapsto \mathsf{D}_j(r, t) : \alpha} \ \exists \beta', \tilde{\gamma}'_j . \begin{array}{c}\beta' = \beta, \tilde{\gamma}'_j = \tilde{\gamma}_j, \\ \alpha = \delta_j\end{array} \qquad \text{Destruction}$$

Fig. 5. Rules of inference for Pola terms on coinductive objects.

The return type of the recursive function, g, in an unfold is universal in type ζ in the same way that the first parameter of a recursive function for a fold is universal.

4.6 Unification

Unification of type equations generated by type inference works slightly differently in Pola than in other languages due to the requirement that universally quantified type variables never be allowed to unify with any other types. From unification we get a single equation with likely many quantifiers, for instance $\exists\alpha,\beta.\forall\gamma.\beta = \mathsf{Bool}, \exists\delta.\delta = \alpha, \alpha = \beta$. Unification works from right-to-left, eliminating quantifiers until only concrete types or free variables remain.

We use capital letters (e.g., A, B) to stand for strings of equations (where B does not contain a quantifier) and lowercase Greek letters (e.g., α, β) to stand for types or type variables. There are only four rewriting rules:

$$\begin{array}{rl}
A, \exists\alpha.B \Rightarrow A, B & \text{If there is no mention of } \alpha \text{ in } B \\
A, \exists\alpha.\alpha = \beta, B \Rightarrow A, B[\beta/\alpha] & \\
A, \forall\alpha.B \Rightarrow A, B & \text{If there is no mention of } \alpha \text{ in } B \\
A, \Box\alpha.\beta(\gamma) = \beta(\eta), B \Rightarrow A, \Box\alpha.\gamma = \eta, B & \begin{array}{l}\text{Breaking up structure.} \\ \Box \in \{\exists, \forall\}\end{array}
\end{array}$$

$B[\beta/\alpha]$ stands for B with α substituted by β, performing an occurs check to ensure β does not contain α. It is a type error if there is a universal quantifier where α *does* occur in B. Ordering within a list of equations not containing a quantifier is unimportant.

As an example, consider the following unification process:

$$\begin{array}{l}
\exists\alpha,\beta.\forall\gamma.\beta = \mathsf{Bool}, \exists\delta.\delta = \alpha, \alpha = \beta \\
\Rightarrow \exists\alpha,\beta.\forall\gamma.\beta = \mathsf{Bool}, \alpha = \beta \\
\Rightarrow \exists\alpha,\beta.\beta = \mathsf{Bool}, \alpha = \beta \\
\Rightarrow \exists\alpha.\alpha = \mathsf{Bool}
\end{array}$$

5 Operational semantics

We describe the semantics of the language in big step style, shown in figure 6. A sequent is given in the notation $\Gamma \vdash t \Rightarrow u$ to mean that, in the environment of Γ which maps symbols to values, term t reduces to value u. Note that since affineness and distinctions between player and opponent player contexts only have significance during type inference, we only have a flat context, between symbols and values, when evaluating via the operational semantics. Likewise, the difference between product and tensor, between case and peek and the difference between player where and opponent where are no longer considered.

Coinductive values store the environment from when they were "recorded", similarly to closures in other languages. They also store the bodies of all destructors. The notation $\left\langle \Phi \middle| \begin{array}{c} \mathsf{D}_1 : \lambda x_1, \ldots, x_m.t_1 \\ \vdots \\ \mathsf{D}_n : \lambda y_1, \ldots, y_k.t_n \end{array} \right\rangle$ is used to denote a coinductive "record" with stored environment Φ and destructors D_1 through D_n. Note in the semantics for a destruction that we switch to the stored environment, Φ, when evaluating the destructor.

$$\frac{}{x = u, \mathit{\Gamma} \vdash x \Rightarrow u} \quad \text{Variable reference}$$

$$\frac{\mathit{\Gamma} \vdash t_1 \Rightarrow u_1 \quad \cdots \quad \mathit{\Gamma} \vdash t_n \Rightarrow u_n}{\mathit{\Gamma} \vdash (t_1, \ldots, t_n) \Rightarrow (u_1, \ldots, u_n)} \quad \text{Tensor/product}$$

$$\frac{[f = \cdots, \mathit{\Gamma} \vdash t_i \Rightarrow u_i]_{1 \leq i \leq n} \quad x_1 = u_1, \ldots, x_n = u_n, f = \cdots, \mathit{\Gamma} \vdash t \Rightarrow u}{f = \lambda x_1 \ldots x_n . t, \mathit{\Gamma} \vdash f(t_1, \ldots, t_n) \Rightarrow u} \quad \text{Function call}$$

$$\frac{[\mathit{\Gamma} \vdash t_i \Rightarrow u_i]_{1 \leq i \leq n}}{\mathit{\Gamma} \vdash \mathsf{C}(t_1, \ldots, t_n) \Rightarrow \mathsf{C}(u_1, \ldots, u_n)} \quad \text{Construction}$$

$$\frac{\mathit{\Gamma} \vdash r \Rightarrow \left\langle \mathit{\Phi} \left| \begin{matrix} \mathsf{D}_i : \lambda y_1 \ldots y_k . t \\ \vdots \end{matrix} \right. \right\rangle \quad \mathit{\Gamma} \vdash t_i \Rightarrow u_i \quad y_1 = u_1, \ldots, y_k = u_k, \mathit{\Phi} \vdash t \Rightarrow u}{\mathit{\Gamma} \vdash \mathsf{D}_i(r, t_1, \ldots, t_k) \Rightarrow u} \quad \text{Destruction}$$

$$\frac{\mathit{\Gamma} \vdash t \Rightarrow \mathsf{C}_i(s_1, \ldots, s_k) \quad x_1 = s_1, \ldots, x_k = s_k, \mathit{\Gamma} \vdash t_i \Rightarrow u}{\mathit{\Gamma} \vdash \mathsf{peek}\ t\ \mathsf{of}\ \{\cdots \mathsf{C}_i(x_1, \ldots, x_k).t_i; \cdots\} \Rightarrow u} \quad \text{Case/peek}$$

$$\frac{}{\mathit{\Gamma} \vdash \begin{pmatrix} \mathsf{D}_1 : x_1 \ldots x_m . t_1; \\ \cdots; \\ \mathsf{D}_n : y_1 \ldots y_k . t_n \end{pmatrix} \Rightarrow \left\langle \mathit{\Gamma} \left| \begin{matrix} \mathsf{D}_1 : \lambda x_1 \ldots x_m . t_1 \\ \vdots \\ \mathsf{D}_n : \lambda y_1 \ldots y_k . t_n \end{matrix} \right. \right\rangle} \quad \text{Record}$$

$$\frac{f = \lambda y_1 \ldots y_n . \mathsf{peek}\ y_1\ \mathsf{of}\ \{\cdots\}, \mathit{\Gamma} \vdash t \Rightarrow u}{\mathit{\Gamma} \vdash \mathsf{fold}\ f(y_1, \ldots, y_n)\ \mathsf{as}\ \{\cdots\}\ \mathsf{in}\ t \Rightarrow u} \quad \text{Fold}$$

$$\frac{f = \lambda y_1 \ldots y_n . (\cdots), \mathit{\Gamma} \vdash t \Rightarrow u}{\mathit{\Gamma} \vdash \mathsf{unfold}\ f(y_1, \ldots, y_n)\ \mathsf{as}\ (\cdots)\ \mathsf{in}\ t \Rightarrow u} \quad \text{Unfold}$$

$$\frac{\mathit{\Gamma} \vdash s \Rightarrow v \quad x = v, \mathit{\Gamma} \vdash t \Rightarrow u}{\mathit{\Gamma} \vdash t\ \mathsf{where}\ x = s \Rightarrow u} \quad \text{Where}$$

Fig. 6. Operational semantics.

$$
\begin{array}{rcll}
\Gamma \rhd \|x\| &=& 1 & \text{Variable}\\
\Gamma \rhd \|(t_1,\ldots,t_n)\| &=& 1 + \sum_{i=1}^{n} \Gamma \rhd \|t_i\| & \text{Tensor/product}\\
\Gamma \rhd \|f(t_1,\ldots,t_n)\| &=& 1 + \sum_{i=1}^{n} \Gamma \rhd \|t_i\| + & \text{Function call}\\
&& f_{\text{time}}(\Gamma \rhd |t_1|, \ldots, \Gamma \rhd |t_n|) & \\
&& \text{where } f_{\text{size}} \in \Gamma & \\
\Gamma \rhd \|\mathsf{C}(t_1,\ldots,t_n)\| &=& 1 + \sum_{i=1}^{n} \Gamma \rhd \|t_i\| & \text{Construction}\\
\Gamma \rhd \left\| \begin{array}{c} \mathsf{peek}\ t\ \mathsf{of} \\ \{\ \mathsf{C}_i(x_i,y_i).u_i\ \} \end{array} \right\| &=& 1 + \Gamma \rhd \|t\| + \max_i \Gamma, a/x_i, b/y_i \rhd \|u_i\| & \text{Case/peek}\\
&& \text{where } (a,b) = \Gamma \rhd |t|_{\mathsf{C}_i} & \\
\Gamma \rhd \left\| \begin{array}{c} \mathsf{fold}\ f(w,z)\ \mathsf{as} \\ \{\ \mathsf{C}_i(x_i,y_i).u_i\ \} \\ \mathsf{in}\ t \end{array} \right\| &=& \Gamma, \left(\begin{array}{l} \lambda wz. \sum_i a \cdot (\Gamma' \rhd \|u_i\|) \\ \quad \text{where } (a,b) = \Gamma \rhd |w|_{\mathsf{C}_i} \\ \quad \Gamma' = \Gamma, 0/f_{\text{time}}, \\ \quad a/x_i, b/y_i \end{array} \right) / f_{\text{time}} \rhd \|t\| & \text{Fold}\\
\Gamma \rhd \|\mathsf{D}_i(r,t_1,\ldots,t_m)\| &=& 1 + \sum_{j=1}^{m} \Gamma \rhd \|t_j\| + \|r\| + & \text{Destruction}\\
&& \|r\|^{P}_{\mathsf{D}_i}(\Gamma \rhd |t_1|, \ldots, \Gamma \rhd |t_m|) & \\
\Gamma \rhd \|(\cdots \mathsf{D}_i : x_i.t_i; \cdots)\| &=& 1 & \text{Record}\\
\Gamma \rhd \left\| \begin{array}{c} \mathsf{unfold}\ g(\cdots)\ \mathsf{as} \\ (\cdots)\ \mathsf{in}\ t \end{array} \right\| &=& (\Gamma, 1/g_{\text{time}} \rhd \|t\|) & \text{Unfold}\\
\Gamma \rhd \|t\ \mathsf{where}\ x = s\| &=& 1 + (\Gamma \rhd \|s\|) + & \text{Where}\\
&& (\Gamma, 1/x_{\text{time}}, (\Gamma \rhd |s|)/x_{\text{size}} \rhd \|t\|) &
\end{array}
$$

Fig. 7. Inferring time bounds from Pola terms.

6 Bounds inference

We sketch how to infer bounds on Pola terms: this is still a work in progress. There are two bounds which need to be inferred simultaneously: the time bound (indicating the amount of computational time needed to evaluate according to the operational semantics) and the size bound (indicating the size of the value yielded by the computation). Each function defined in Pola has, along with a type signature, a time signature (denoted f_{time} and either an inductive size signature f_{size} or potential time and size signatures (f_{pot}).

We do not prove that the bounds are correct according to the operational semantics given in section 5, but rather give some intuition for them. A proof of polynomial-time soundness has been provided in [3] and a complete proof of correctness of bounds inference will be provided in future work.

For inductive data types, the size bound is a count of how many of each constructor a type has, paired with the maximum sizes of its constituent data. For example, the list $[5,7,2] : \mathsf{List}(\mathsf{Nat})$ would be of size $\langle \mathsf{Nil} : 1, \mathsf{Cons} : (3, \langle \mathsf{Zero} : 1, \mathsf{Succ} : 7\rangle)\rangle$ to indicate the list has at most (in this case exactly) 1 Nil constructor and 3 Cons constructors and that each Cons constructor contains a number containing at most 1 Zero constructor and at most 7 Succ constructors. In this example the counts were simple integers, but in general they are polynomials over integer coefficients, typically with variables being the inputs to the term or function being analysed. The size of a tuple is a tuple of the sizes of its elements.

Note that one could provide a simpler system for inferring bounds by neglecting to keep a list of constructor counts and instead maintain only an aggregate count of all constructors. However, to get tighter, and thus more useful, bounds, this level of detail is necessary. E.g., a fold over natural numbers may have high computational cost for the Zero branch but little computation cost for the Succ case; taking the maximum cost across the branches and multiplying by the total number of constructors would yield an unacceptable loose upper bound.

Time bounds are given in figure 7. Many of these are constant or given recursively in terms of their subterms and the sizes of the terms they work over. For instance, the time taken to execute a fold construct depends on the number of constructors present in the subject of the fold multiplied by the time taken to execute each branch of the fold. Within the body of the fold, the time bound signature of recursive function f is taken to be 0.

Coinductive potential time bounds are omitted as future work, though referred to as the $|\_|^P$ function, which yields destructors paired with potential time bounds and potential size bounds.

Special attention must be paid to variables. Sizes of variables introduced by patterns, such as by case constructs, are determined by the terms which they match. For instance, if the size of x is $\langle \mathsf{Nil} : x_{\mathsf{Nil}} \mid \mathsf{Cons} : x_{\mathsf{Cons}}, x_{\text{data}} \rangle$, then in the term $\mathsf{case}\ x\ \mathsf{of}\ \{\cdots; \mathsf{Cons}(z, zs).zs\ \}$, the size of term zs is $\langle \mathsf{Nil} : x_{\mathsf{Nil}} \mid \mathsf{Cons} : x_{\mathsf{Cons}} - 1, x_{\text{data}} \rangle$, i.e., it is given in terms of x.

Figure 8 shows the method to determine the inductive size of a term, or more precisely, the size of the value that a term evaluates to under the operational semantics. Surprisingly, the most complex size term is that associated with constructors. In that case we separate the arguments into non-recursive and recursive, $\mathsf{C}_i(t, u)$ where u is recursive. In this case we recursively determine the sizes of the u terms; this yields a tuple of inductive sizes, all of the inductive sizes being over the same constructors. We "sum" the elements of that tuple, summing the constructor counts and taking the maximum of its constituent data.

For example, should we want to find $|\mathsf{Node}(\mathsf{Node}(\mathsf{Leaf}(1), \mathsf{Leaf}(0)), \mathsf{Leaf}(0))|$, we must first find $|(\mathsf{Node}(\mathsf{Leaf}(1), \mathsf{Leaf}(0)), \mathsf{Leaf}(1))|$ which is calculated recursively to be $\langle\langle \mathsf{Leaf} : (2, \langle \mathsf{Zero} : 1, \mathsf{Succ} : 1\rangle), \mathsf{Node} : 1\rangle, \langle \mathsf{Leaf} : (1, \langle \mathsf{Zero} : 1, \mathsf{Succ} : 0\rangle), \mathsf{Node} : 0\rangle\rangle$. Summing the two elements of the tuple gives $\langle \mathsf{Leaf} : (3, \langle \mathsf{Zero} : 1, \mathsf{Succ} : 1\rangle), \mathsf{Node} : 1\rangle$. We then add one to the count of the Node constructor to give the final size value of $\langle \mathsf{Leaf} : (3, \langle \mathsf{Zero} : 1, \mathsf{Succ} : 1\rangle), \mathsf{Node} : 2\rangle$.

6.1 Inferring size bounds from folds

Inferring size bounds from fold constructs requires special care. To determine $|\mathsf{fold}\ f(x, y)\ \mathsf{as}\ \{\ \mathsf{C}_i(x_i, w_i).u_i\ \}\ \mathsf{in}\ f(t, u)|$, we first must determine the maximum values of $y = (y_1, \ldots, y_m)$. Initially, $|y_j| = |t_j|$ for each $1 \leq j \leq m$; however, the values of y_j change through each recursive call to function f in general. The case where these y_j values change is left to future work. As the y_j variables are

$$\begin{array}{rcll}
\mathit{\Gamma}, m/x_{\text{size}} \rhd |x| &=& m & \text{Variable}\\
\mathit{\Gamma} \rhd |(t_1, \ldots, t_n)| &=& \langle \mathit{\Gamma} \rhd |t_1|, \ldots, \mathit{\Gamma} \rhd |t_n| \rangle & \text{Tensor/product}\\
\mathit{\Gamma} \rhd |f(t_1, \ldots, t_n)| &=& f_{\text{size}}(\mathit{\Gamma} \rhd |t_1|, \ldots, \mathit{\Gamma} \rhd |t_n|) \text{ where } f_{\text{size}} \in \mathit{\Gamma} & \text{Function call}\\
\mathit{\Gamma} \rhd |\mathsf{C}_i(t, u)| &=& \langle \mathsf{C}_1 : (m_1, z_1), \ldots, & \text{Construction}\\
&& \quad \mathsf{C}_i : (1 + m_i, \max(z_i, \mathit{\Gamma} \rhd |t|)), & \\
&& \quad \cdots, \mathsf{C}_n : (m_n, z_n) \rangle & \\
&& \text{where } \langle \mathsf{C}_1 : (m_1, z_1), \cdots, \mathsf{C}_n : (m_n, z_n) \rangle & \\
&& \quad = \sum \mathit{\Gamma} \rhd |u| & \\
\mathit{\Gamma} \rhd |\mathsf{peek}\ t\ \mathsf{of}\ \{\ \mathsf{C}_i(x_i, y_i).u_i\ \}| &=& \max_i \mathit{\Gamma}, a/x_i, b/y_i \rhd |u_i| \text{ where } (a, b) = |t|_{\mathsf{C}_i} & \text{Case/peek}\\
\mathit{\Gamma} \rhd |\mathsf{fold}\ f(\cdots)\ \mathsf{as}\ \{\cdots\}\ \mathsf{in}\ t| &=& \mathit{\Gamma}, \text{See section 6.1}/f_{\text{size}} \rhd |t| & \text{Fold}\\
\mathit{\Gamma} \rhd |\mathsf{D}_i(r, t_1, \ldots, t_m)| &=& \mathit{\Gamma} \rhd |r|^{P}_{\mathsf{D}_i}(\mathit{\Gamma} \rhd |t_1|, \ldots, \mathit{\Gamma} \rhd |t_m|) & \text{Destruction}\\
\mathit{\Gamma} \rhd |t\ \mathsf{where}\ x = s| &=& \mathit{\Gamma}, (\mathit{\Gamma} \rhd |s|)/x \rhd |t| & \text{Where}
\end{array}$$

Fig. 8. Inferring size bounds from inductive Pola terms.

player variables, these values cannot affect inference of time bounds, but the size bounds may be expressed in terms of the y_j variables.

Once bounds on the sizes of y_j can be determined, bounds for the terms u_i of the fold can be determined. We denote $u_i^\star$ to be the size of each branch and the size of the entire fold term to be $\max_i u_i^\star$. If u_i does not contain any recursive calls to function f, then simply $u_i^\star = |u_i|$. If u_i contains a recursive call to function f, we determine the operations performed on the recursive function and iterate those by the the number of constructors C_i in the subject of the fold. Again, the details of this are left to future work.

As an example, consider the fold term in the body of the *add* function given in section 3. In that case we have $|y| = m$, since y is never modified in a recursive call and thus has the same size as its initial value, m. Looking at the branches of the fold, then, $u_1^\star = (u_1)^\star = |y| = m$. For the Succ branch we have:

$$\begin{aligned}
u_2^\star &= n_{\mathsf{Succ}} \cdot \langle \mathsf{Zero} : 0 \mid \mathsf{Succ} : 1 \rangle + m\\
&= \langle \mathsf{Zero} : m_{\mathsf{Zero}} \mid \mathsf{Succ} : m_{\mathsf{Succ}} + n_{\mathsf{Succ}} \rangle
\end{aligned}$$

We find that $\max(u_1^\star, u_2^\star) = u_2^\star$ and thus the size of the value resulting from the fold is $\langle \mathsf{Zero} : m_{\mathsf{Zero}} \mid \mathsf{Succ} : m_{\mathsf{Succ}} + n_{\mathsf{Succ}} \rangle$. In this case this is a tight bound: the value resulting from an addition will have a number of Succ constructors equal to $m + n$. In general, the bound given by this method may be quite loose, however.

6.2 Potential time and size bounds

For coinductive data types, inductive size bounds are not relevant. The typing system enforces that coinductive values will never be considered in the context of inferring inductive size bounds. However, we must consider the *potential* time and size bounds of coinductive values, i.e., the time and size costs that would be incurred if the coinductive object were to be destructed. Coinductive data works from a state and destruction causes state change and production of values. But each destructor is constant time so their cumulative effect can be obtained both in time and size by adding effects.

7 Conclusion

We have provided an overview of the implementation details surrounding the programming language Pola. Pola is a restricted language wherein every program halts in time polynomial with respect to its input, a property enforced by the typing system. It is a functional language offering type inference and laziness. We also presented a method for automatically inferring upper bounds on time and size requirements.

References

1. S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the poly-time functions. *Computational Complexity*, 2:97–110, 1992.
2. M.J. Burrell, R. Cockett, and B.F. Redmond. Pola project page. http://projects.wizardlike.ca/projects/pola.
3. M.J. Burrell, R. Cockett, and B.F. Redmond. Pola: a language for PTIME programming. In *Tenth International Workshop on Logic and Computational Complexity*, Los Angeles, USA, August 2009.
4. R. Cockett and R. Seely. Polarized category theory, modules and game semantics. *Theory and application of categories*, 18:4–101, 2007.
5. L. Colson. About primitive recursive algorithms. *Theoretical Computer Science*, 83(1):57–69, 1991.
6. M. Hofmann. Type systems for polynomial-time computation. Habilitation thesis. University of Darmstdat, 1999.
7. M. Hofmann. Linear types and non-size-increasing polynomial time computation. *Information and Computation*, 183(1):57–85, 2003.
8. D. Leivant. Stratified functional programs and computational complexity. In *Proc. 20th IEEE Symp. on Principles of Programming Languages*, pages 325–333, 1993.
9. D. Leivant and J.-Y. Marion. Ramified recurrence and computational complexity II: Substitution and poly-space. In *Proc. CSL'94, Springer LNCS*, volume 933, pages 486–500, 1994.
10. P. O'Hearn. On bunched typing. *Journal of Functional Programming*, 13(4):747–796, 2003.
11. P. O'Hearn and D. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, 1999.
12. Benjamin C. Pierce. *Types and programming languages*, chapter 22. MIT Press, 2002.

A local criterion for polynomial time stratified computations

Luca Roversi[1], Luca Vercelli[2] *

[1] Dip. di Informatica, Univ. di Torino (`http://www.di.unito.it/~rover/`)
[2] Dip. di Matematica, Univ. di Torino (`http://www.di.unito.it/~vercelli/`)

Abstract. This work is a consequence of studying the (*un*)relatedness of the principles that allow to implicitly characterize the polynomial time functions (PTIME) under two perspectives. One perspective is predicative recursion, where we take Safe Recursion on Notation as representative. The other perspective is structural proof theory, whose representative can be Light Affine Logic (LAL). A way to make the two perspectives closer is to devise polynomial sound generalizations of LAL whose set of interesting proofs-as-programs is larger than the set LAL itself supplies. Such generalizations can be found in MS.
MS is a *Multimodal Stratified framework* that contains *subsystems* among which we can find, at least, LAL. Every subsystem is essentially determined by two sets. The first one is countable and finite, and supplies the modalities to form modal formulæ. The second set contains building rules to generate proof nets. We call MS *multimodal* because the set of modalities we can use in the types for the proof nets of a subsystem is arbitrary. MS is also *stratified*. This means that every box, associated to some modality, in a proof net of a subsystem can never be opened. So, inside MS, we preserve *stratification* which, we recall, is the main structural proof theoretic principle that makes LAL a polynomial time sound deductive system. MS is expressive enough to contain LAL and Elementary Affine Logic (EAL), which is PTIME-*un*sound. We supply a set of syntactic constraints on the rules that identifies the PTIME-maximal subsystems of MS, i.e. the PTIME-sound subsystems that contain the largest possible number of rules. It follows a syntactic condition that discriminates among PTIME-sound and PTIME-*un*sound subsystems of MS: a subsystem is PTIME-sound if its rules are among the rules of some PTIME-maximal subsystem. All our proofs widely use the techniques Context Semantics supplies, and in particular the geometrical configuration that we call *dangerous spindle*: a subsystem is polytime if and only if its rules cannot build dangerous spindles.

1 Introduction

This work fits the theoretical side of Implicit Computational Complexity (ICC). Our primary goal is looking for the systems that can replace the question marks (1), (2), and (3) in Fig.1. In there, SRN is Safe Recursion on Notation [1], namely a polynomial time

* Both the authors have been supported by MIUR PRIN CONCERTO — protocol number 2007BHXCFH

$$\begin{tikzcd}
\text{PTIME Turing Machines} \arrow[r, Leftrightarrow] \arrow[dd, Leftrightarrow] & \text{SRN} \arrow[r, mapsto] \arrow[d, phantom, "\rotatebox{90}{$\subseteq$}"] & ?\ (3) \arrow[d, phantom, "\rotatebox{90}{$\subseteq$}"] \\
 & ?\ (1) \arrow[r, mapsto] \arrow[d, phantom, "\rotatebox{90}{$\subseteq$}"] & ?\ (2) \arrow[d, phantom, "\rotatebox{90}{$\subseteq$}"] \\
\text{BC}^{\pm} \arrow[r, phantom, "\supseteq"] \arrow[rr, bend right, ""] & \text{BC}^{-} \arrow[r, mapsto] & \text{LAL} \arrow[uull, bend right, leftrightarrow]
\end{tikzcd}$$

Fig. 1. Relations between SRN and LAL.

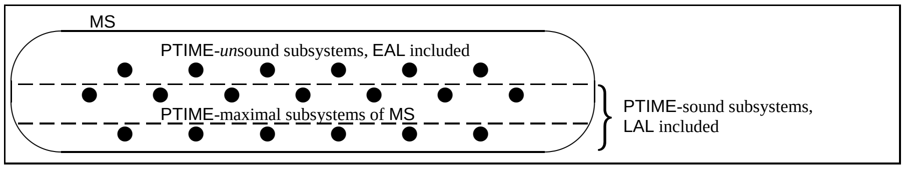
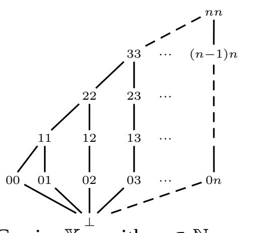

MS
PTIME-*un*sound subsystems, EAL included
PTIME-maximal subsystems of MS
PTIME-sound subsystems, LAL included

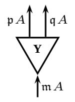
Fig. 2. PTIME-*un*sound, PTIME-sound, and PTIME-maximal subsystems of MS.

(PTIME) sound and complete restriction of Primitive Recursion; LAL is Light Affine Logic [2], a deductive system derived from Linear Logic (LL, [3]), based on proof nets, which is PTIME sound and complete under the proofs-as-programs analogy. As shown in [4], there exists a subalgebra BC^- of SRN that compositionally embeds into LAL. However, it is not possible to extend the same embedding to the whole SRN. As far as we know, any reasonable replacement of the question marks (1), (2), and (3) in Fig. 1 is still unknown. The results in [5] and [6] justify the obstructions to the extension of the embedding in [4] that would replace LAL for (2), or (3). Indeed, [5] shows the *strong* PTIME-soundness of LAL, while [6] shows that SRN is just *weakly* PTIME-sound, once we see it as a term rewriting system. Since the strong PTIME-sound programs are, intuitively speaking, far less than the weak ones, the misalignment looks evident. So, a way to fill the gaps in Fig. 1 is looking for an extension of LAL. This is not impossible since LAL does not supply the largest set of programs we can write under the structural proof theoretic constraints that LAL itself relies on. To better explain this, we need some steps.

Recalling LAL. The proof nets of LAL inherits the !-boxes from LL. Every !-box identifies a specific region of a proof net that can be duplicated by the cut elimination. *In* LAL *each box that can be duplicated, called* !*-box, depends on at most a single assumption*. Besides the duplicable !-boxes, LAL has §-boxes, which cannot be duplicated. §-boxes implement the *weak dereliction* principle $!A \multimap \S A$. Namely, every §-box allows to access the content of a !-box, preserving its border. Accessing the proof net inside a !-box is useful to program iterations through Church numerals, for example. Since also the §-boxes can never be "opened", *the proof nets of* LAL *are stratified*. "Stratification" means that every node u of every proof net of LAL either gets erased by the cut elimination, or the number of nested boxes the residuals of u are contained in keeps being constant when the cut elimination proceeds.



(a) Carrier $\mathbb{X}_n$, with $n \in \mathbb{N}$.



(b) Contraction nodes, with $\mathfrak{m}, \mathfrak{p}, \mathfrak{q} \in \mathbb{X}_n$ such that $\mathfrak{p}, \mathfrak{q}$ are incomparable and $\mathfrak{m}$ is their least upper bound.

Fig. 3. Carrier and Contraction nodes of $\mathsf{Linear}_{\mathbb{X}_n}$.

PTIME-sound and PTIME-*un*sound extensions of LAL. Once recalled LAL, we propose some extensions of it that preserve stratification. We start extending LAL to $\mathsf{LAL}_{\mathtt{A}\nabla}$ by adding an "*asymmetric*" Contraction $!A \multimap (!A \otimes \S A)$. We shall prove the PTIME-soundness of $\mathsf{LAL}_{\mathtt{A}\nabla}$ thanks to Proposition 1. The PTIME-completeness of $\mathsf{LAL}_{\mathtt{A}\nabla}$ should be evident since it contains LAL. Though, what is worth remarking now is that (i) LAL *can be* PTIME-*soundly extended* to another system by adding some rules, and (ii) *the new building rule may allow to write new interesting programs*. This potentially candidates $\mathsf{LAL}_{\mathtt{A}\nabla}$ to replace (2) or (3) in Fig.1 because $\mathsf{LAL}_{\mathtt{A}\nabla}$ might have programs missing in LAL that allow to simulate terms not in BC^-. Instead, extending LAL to $\mathsf{LAL}_{\nabla\S}$ by adding a Contraction $\S A \multimap (\S A \otimes \S A)$ yields a PTIME-*un*sound system. The reason is that $\mathsf{LAL}_{\nabla\S}$ contains EAL, the affine version of ELL [7], which soundly and completely characterizes the class of Elementary Functions. Finally, we might think to extend LAL to $\mathsf{LAL}_{\not{1}}$, in which the !-boxes are allowed to depend on more than one assumption. Once again, $\mathsf{LAL}_{\not{1}}$ is PTIME-*un*sound.

The above experiments show that extending LAL by new contraction nodes or new boxes, flexibly depending on parameters, we can hope to devise PTIME-sound generalizations of LAL, able to replace (2) or (3) in Fig. 1.

The stratified an multimodal framework MS (Section 2). Abstracting away from the experiments on LAL led to this work. Fig. 2 visualizes what we mean. MS was first introduced in [8]. Here we make the definition more essential, while extending it so that the (sub)systems of proof nets MS contains can use unconstrained Weakening. MS is a *Multimodal* and *Stratified* framework. MS is a class of triples $(\mathbb{X}, \mathcal{B}_{\mathbb{X}}, \mathcal{R}_{\mathbb{X}})$. The first element $\mathbb{X}$, we call carrier, is an *arbitrary* countable and finite set of elements we use as modalities inside a language $\mathcal{F}_{\mathbb{X}}$ of formulæ. The second element $\mathcal{B}_{\mathbb{X}}$ is a set of *building rules*, ideally partitioned into *linear* and *modal* ones. The *linear* building rules define the proof nets of Multiplicative and Second Order Fragment (MLL_2) of LL. The *exponential* building rules are specific to the triple, and define both (i) which modal formulæ of $\mathcal{F}_{\mathbb{X}}$ that label the premises of a proof net can be contracted into a single premise, and (ii) which modal formulæ are associated to the conclusion of a box around a proof net. At this point, to keep things intuitive, we can think a *subsystem* $\mathcal{P}$ of MS

is every triple that satisfies an essential requirement to use the proof nets $\mathbf{PN}(\mathcal{P})$ that $\mathcal{B}_{\mathbb{X}}$ can generate as a rewriting system. This amounts to require that $\mathcal{R}_{\mathbb{X}}$ is the largest rewriting relation in $\mathbf{PN}(\mathcal{P}) \times \mathbf{PN}(\mathcal{P})$.
An example of a whole class of subsystems of MS is Linear_n, with $n \in \mathbb{N}$, already defined in [8]. Linear_n allows to remark the freedom we have when choosing a carrier set. Fig. 3(b) shows the partial order that we can take as $\mathbb{X}_n$ to define Linear_n, while Fig. 3(a) shows a subset of the Contraction nodes induced by $\mathbb{X}_n$. Remarkably, (i) Linear_n has a linear normalization cost, like MLL_2, but (ii) it can represent the Church numerals as much long as n, together with the basic operations on them, so strictly extending the expressiveness of MLL_2.
The local criterion (Section 3-4). Its statement relies on the notion of PTIME-maximal subsystem $\mathcal{P}$ of MS. Specifically, $\mathcal{P}$ is PTIME-maximal if it is PTIME-sound, and any of its extensions becomes PTIME-*un*sound. Our criterion says that a given $\mathcal{P}$ is PTIME-maximal by listing a set of sufficient and necessary conditions on the syntax of the building rules in $\mathcal{P}$. As a corollary, any given $\mathcal{P}'$ is PTIME-sound if its rules are among those ones of a PTIME-maximal subsystems of MS. To conclude, *spindle* (Section 3, Fig. 12(a)) is the technical notion we base our criterion on. A spindle is a conceptual abstraction of the general quantitative analysis tools that Context Semantics (CS) [9] supplies. Intuitively, if a subsystem $\mathcal{P}$ allows to concatenate arbitrary long *chains of spindles* (Fig. 12(b)), then it is PTIME-*un*sound, namely some of its rules cannot by instance of any PTIME-maximal subsystem of MS.

Acknowledgments. We like to thank the anonymous referees whose comments allowed to better justify the main aspects of this work, we hope.

2 The framework MS

We define MS by extending, and cleaning up, the definition in [8]. Our current MS generates subsystems with unconstrained weakening, like LAL, to easy programming.
The Formulæ. Let $\mathbb{X}$ be an alphabet of *modalities*, ranged over by $\mathfrak{m}, \mathfrak{n}, \mathfrak{p}, \mathfrak{q}, \ldots$, and $\mathcal{V}$ be a countable set of propositional variables, ranged over by $x, y, w, \ldots$. The set $\mathcal{F}_{\mathbb{X}}$ of *formulæ*, generated with F as start symbol, is:

$$F ::= L \mid E \qquad E ::= \mathfrak{m}F \qquad L ::= x \mid F \otimes F \mid F \multimap F \mid \forall x.F$$

E generates *modal* formulæ, L *linear* (*non-modal*) ones. A, B, C range over formulæ of $\mathcal{F}_{\mathbb{X}}$. Γ, Δ, Φ range over, possibly empty, multisets of formulæ. $A\left[{}^{B}/_{y}\right]$ is the substitution of B for y in A. The *number of modalities* of $\mathcal{F}_{\mathbb{X}}$ is the cardinality of $\mathbb{X}$.
The framework. MS is a class of triples $(\mathbb{X}, \mathcal{B}_{\mathbb{X}}, \mathcal{R}_{\mathbb{X}})$, for every countable set $\mathbb{X}$. The element $\mathcal{B}_{\mathbb{X}}$ is the set of the *building rules* in Fig. 4, typed with formulæ of $\mathcal{F}_{\mathbb{X}}$. The nodes $\textcircled{i}, \textcircled{o}$ just show inputs and output, respectively. The other nodes are standard ones. Both **Promotion** and **Contraction** are *modal* as opposed to the *linear* remaining ones. The second component $\mathcal{R}_{\mathbb{X}}$ is the set of *rewriting rules* in Fig. 5, 6, 7, and 8, typed with formulæ of $\mathcal{F}_{\mathbb{X}}$. The name of every rewriting rule recalls the nodes it involves. Fig. 5 defines the *linear rewriting rules* which, essentially, just rewires a proof net. Fig. 6 contains the *modal rewriting rules*, those ones that, once instantiated, may cause exponential blow up. Fig. 7 and 8 describe *garbage collection*.

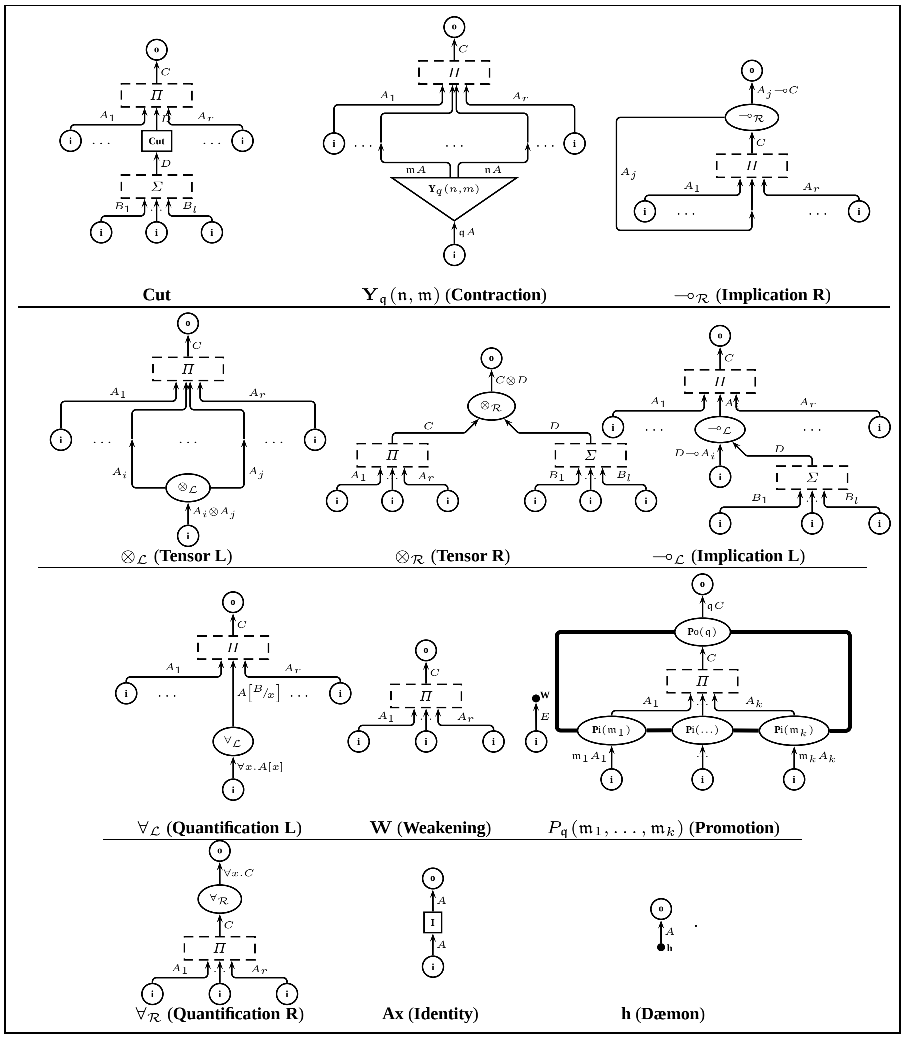

Cut
$\mathbf{Y}_{\mathsf{q}}(\mathfrak{n},\mathfrak{m})$ (Contraction)
$\multimap_{\mathcal{R}}$ (Implication R)
$\otimes_{\mathcal{L}}$ (Tensor L)
$\otimes_{\mathcal{R}}$ (Tensor R)
$\multimap_{\mathcal{L}}$ (Implication L)
$\forall_{\mathcal{L}}$ (Quantification L)
W (Weakening)
$P_{\mathsf{q}}(\mathfrak{m}_1,\ldots,\mathfrak{m}_k)$ (Promotion)
$\forall_{\mathcal{R}}$ (Quantification R)
Ax (Identity)
h (Dæmon)

Fig. 4. Building rules, with short and long names.

Subsystems. A triple $\mathcal{P} = (\mathbb{X}, \mathcal{B}_{\mathbb{X}}, \mathcal{R}_{\mathbb{X}})$ in MS is a *subsystem* (of MS) whenever:

1. $\mathcal{B}_{\mathbb{X}}$ contains all the instances of the linear building rules;
2. $\mathcal{B}_{\mathbb{X}}$ contains every instance of **Promotion** $P_{\mathfrak{n}}()$, that generates *closed* $\mathfrak{n}$*-boxes*;
3. $\mathcal{B}_{\mathbb{X}}$ is *downward closed*. Namely, for every $P_{\mathsf{q}}(\mathfrak{m}_0,\ldots,\mathfrak{m}_k)$ in $\mathcal{B}_{\mathbb{X}}$, $P_{\mathsf{q}}(\mathfrak{n}_0,\ldots,\mathfrak{n}_l)$ belongs to $\mathcal{B}_{\mathbb{X}}$ as well, for every $\{\mathfrak{n}_0,\ldots,\mathfrak{n}_l\} \subseteq \{\mathfrak{m}_0,\ldots,\mathfrak{m}_k\}$;
4. If we denote by $\mathbf{PN}(\mathcal{P})$ the set of proof nets that $\mathcal{B}_{\mathbb{X}}$ inductively generates, using **Identity** and **Dæmon** as base cases, then $\mathcal{R}_{\mathbb{X}}$ is the *largest* rewriting relation inside $\mathbf{PN}(\mathcal{P}) \times \mathbf{PN}(\mathcal{P})$.

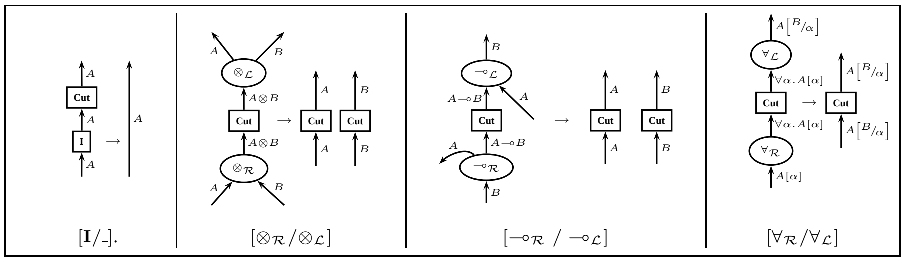


Fig. 5. *Linear* rewriting rules. $[\forall_{\mathcal{R}}/\forall_{\mathcal{L}}]$ substitutes B for α as usual.

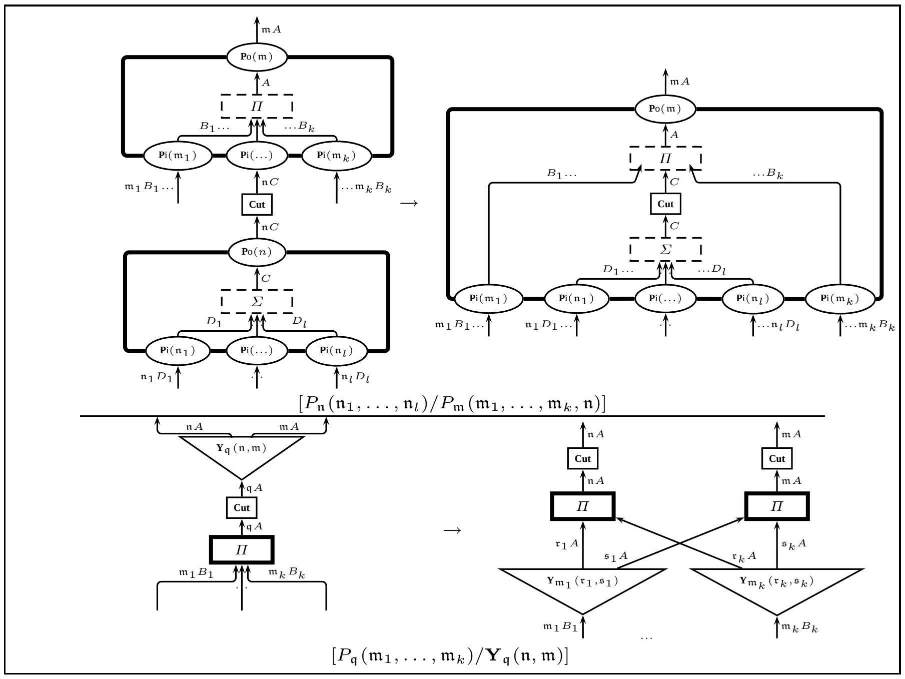
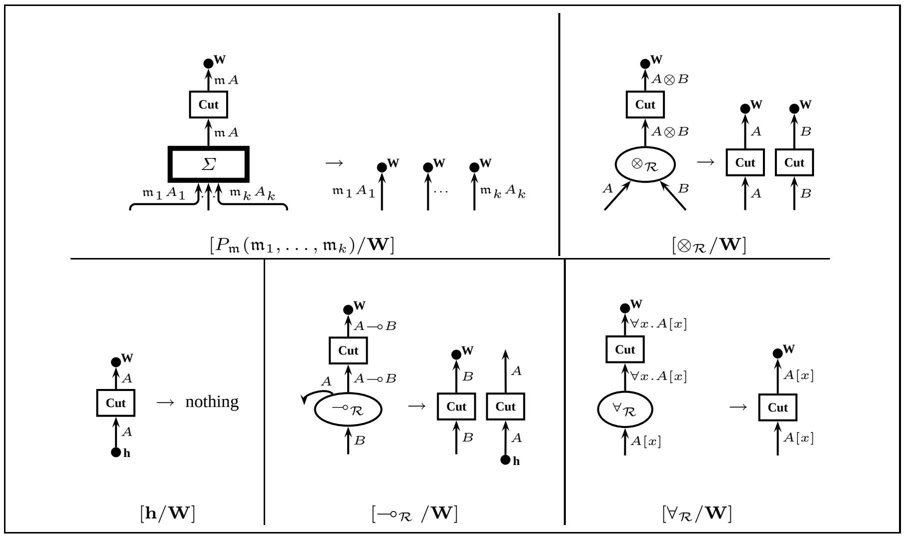


Fig. 6. Modal rewriting rules.

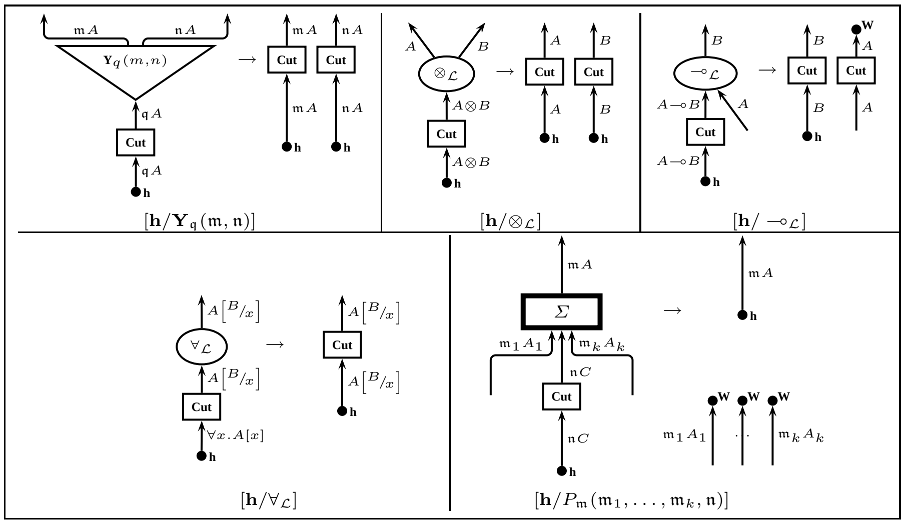
By abusing the notation, we write $\mathcal{P} \subseteq \mathsf{MS}$ to abbreviate that $\mathcal{P}$ is a subsystems of MS.

An example of subsystem. Let $M \geq 2$, and $\mathbb{X} = \{i! \mid i \leq M\} \cup \{i\S \mid i \leq M\}$. Then $\mathsf{sLAL} = (\mathbb{X}, \mathcal{B}_{\mathbb{X}}, \mathcal{R}_{\mathbb{X}})$ is the subsystem of MS such that $\mathcal{B}_{\mathbb{X}}$ contains the modal building rules in Fig.9. The "s" in front of LAL stands for *sorted*. With $\mathfrak{n}^?$ we mean that the premise it represents can occur at most once. With $\mathfrak{n}^*$ we mean that the premise it represents can occur an unbounded, but finite, number of times. By the forthcoming



$[P_{\mathfrak{m}}(\mathfrak{m}_1,\ldots,\mathfrak{m}_k)/\mathbf{W}]$
$[\otimes_{\mathcal{R}}/\mathbf{W}]$
$[\mathbf{h}/\mathbf{W}]$
nothing
$[\multimap_{\mathcal{R}}/\mathbf{W}]$
$[\forall_{\mathcal{R}}/\mathbf{W}]$
Cut

Fig. 7. Garbage collection rewritign rules, involving $\mathbf{W}$.



$[\mathbf{h}/\mathbf{Y}_{\mathfrak{q}}(\mathfrak{m},\mathfrak{n})]$
$[\mathbf{h}/\otimes_{\mathcal{L}}]$
$[\mathbf{h}/\multimap_{\mathcal{L}}]$
$[\mathbf{h}/\forall_{\mathcal{L}}]$
$[\mathbf{h}/P_{\mathfrak{m}}(\mathfrak{m}_1,\ldots,\mathfrak{m}_k,\mathfrak{n})]$
Cut

Fig. 8. Garbage collection rewriting rules, involving $\mathbf{h}$.

Proposition 1, sLAL is PTIME-sound. Notice that sLAL strictly extends LAL. We shall get back to the relevance of sLAL in Section 5.

Remarks on standard computational properties of subsystems. A subsystem $\mathcal{P}$ does not necessarily enjoy standard computational properties. For example, in a given $\mathcal{P}$ a

$$P_{i!}(i!^{?}, j_1\S^*, \ldots, j_m\S^*) \text{ for every } j_1, \ldots, j_m < i \leq M$$
$$P_{i\S}(i!^*, i\S^*, j_1\S^*, \ldots, j_m\S^*) \text{ for every } j_1, \ldots, j_m < i \leq M$$
$$\mathbf{Y}_{i!}(i!, i!) \text{ for every } i \leq M$$

Fig. 9. The modal rules of sLAL.

full normalization may fail because some building rule is missing. Analogously, the Church-Rosser property may not hold, because, for example, $[P_{\mathfrak{q}}(\mathfrak{r})/\mathbf{Y}_{\mathfrak{q}}(\mathfrak{n},\mathfrak{m})]$ generates a non-confluent critical pair. This might be considered a drawback of the "wild" freedom in the definition of the instances of $[P_{\mathfrak{q}}(\mathfrak{r})/\mathbf{Y}_{\mathfrak{q}}(\mathfrak{n},\mathfrak{m})]$. We believe such a freedom necessary to have a chance to find some replacement of the question marks in Fig. 1. Instead, every $\mathcal{P}$ is strongly normalizing since $\mathbf{PN}(\mathcal{P})$ embeds into the proof nets of EAL by collapsing all its modalities into the single one of EAL. So, standard results imply that $\mathcal{P}$ is Church-Rosser if it is locally confluent.

Notations. Let Π be a proof net of a given $\mathcal{P} \subseteq$ MS. The set of its nodes is V_Π, and E_Π the set of edges. Moreover, B_Π is the set of *Box-out* nodes, in natural bijection with the set of the *boxes*. A box, corresponding to some instance of $P_{\mathfrak{q}}(\mathfrak{m}_1, \ldots, \mathfrak{m}_k)$, has one *conclusion* of type $\mathfrak{q}C$ and k *assumptions* of type respectively $\mathfrak{m}_1 A_1, \ldots, \mathfrak{m}_k A_k$, for some $C, A_1, \ldots, A_k$. The *depth*, or *level*, $\partial(u)$ of $u \in V_\Pi \cup E_\Pi$ is the greatest number of nested boxes containing u. The *depth* $\partial(\Pi)$ *of* Π is the greatest $\partial(u)$ with $u \in V_\Pi$. The *size* $|\Pi|$ counts the number of the nodes in Π. We notice that *Box-in/out* nodes do not contribute to the size of the proof net inside the box they delimit.

3 Polynomial time soundness

The goal of this section is an intermediate step to get the local criterion. Here we characterize the class PMS of *polynomial time sound* (PTIME-sound) subsystems of MS. MS is the one in Sec. 2, which, recall, generalizes, while cleaning it up, the one in [8] by adding unconstrained weakening, and the corresponding dæmon. This is why we shall briefly recall the main tools and concepts that allow to prove a sufficient structural condition for PTIME-soundness, relatively to subsystems of MS. Later, we prove the new PTIME-sound necessary condition.

3.1 Sufficient condition for PTIME-soundness

Context Semantics (CS) [9]. The basic tool to identify a sufficient structural condition on the subsystems of MS that implies PTIME-soundness is CS. We use a simplified version of CS, because the proof nets that any subsystem of MS can generate are free of both dereliction and digging.
CS identifies CS-paths to travel along any proof net of any subsystem. Every CS-path simulates the annihilation of pairs of nodes that, eventually, will interact thanks to the application of *r.steps*. The goal of using CS to analyze our proof nets is to count the number $R_\Pi(u)$ of *maximal* CS-*paths* which, traversing a proof net Π, go from any box root u of Π to either a weakening node, erasing it, or to the terminal node of Π or of

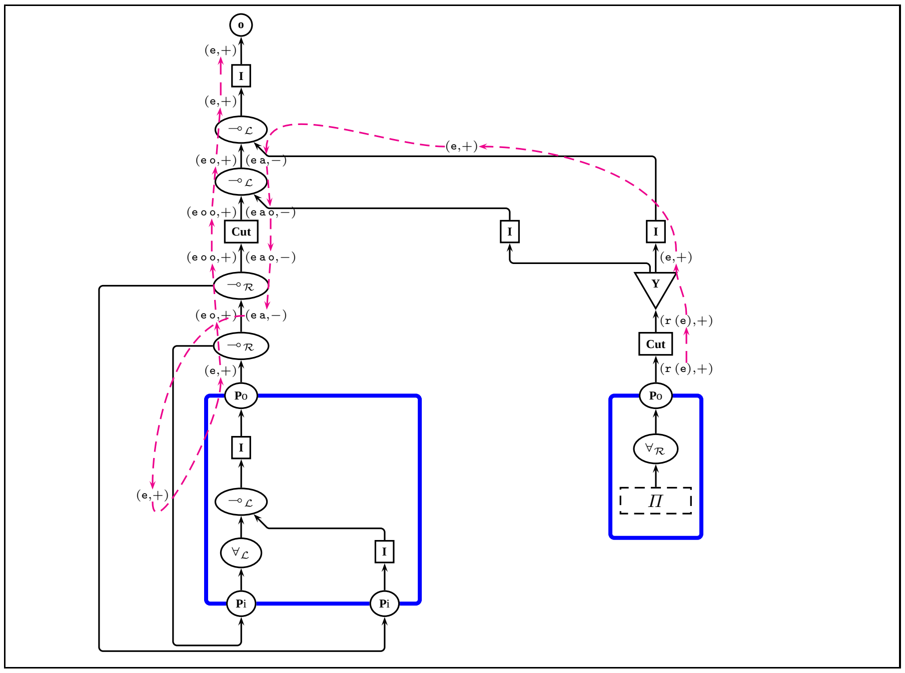


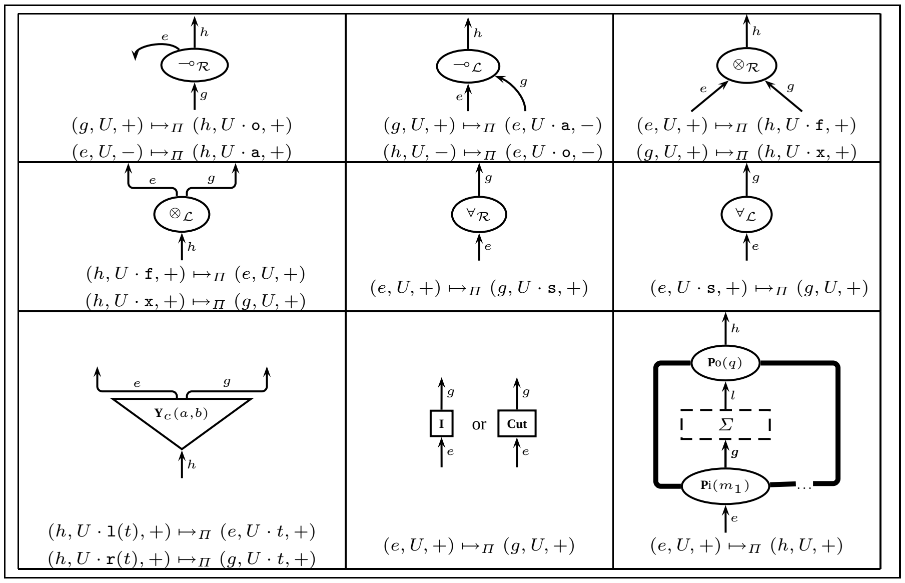
Fig. 10. An example of maximal path.

the proof net inside a box that contains u. The paths that we consider do never cross the border of a box. The value of $R_{\Pi}(u)$ counts the contraction nodes that, possibly, will duplicate the box rooted at u.

Figure 10 shows an example of maximal CS-path from the rightmost box to the conclusion of the proof net. It is built by interpreting every node as it was a kind of operator that manipulates the top element of a stack whose elements can contain symbols of a specific signature. For example, let us focus on the pair $(\mathtt{r}(\mathtt{e}), +)$ the CS-path in Figure 10 starts from. The polarity $+$ says we are feeding the contraction with the value $\mathtt{r}(\mathtt{e})$, coherently with the direction of its premise. We have to think that the top cell of a stack stores $\mathtt{r}(\mathtt{e})$. The contraction node replaces $\mathtt{r}(\mathtt{e})$ by $\mathtt{e}$ in the top. The axiom node behaves as an identity operator on the top. The first $\multimap_{\mathcal{L}}$ we meet pushes $\mathtt{a}$ on the top, so that such a symbol can be popped out by the second $\multimap_{\mathcal{R}}$ we meet. We keep going with these corresponding push-pop sequences until we reach the conclusion. By the way, we notice that the leftmost box is interpreted as an identity w.r.t. the stack content. This is because we are in a stratified setting.

Figure 11 recalls the set of transition steps that formally realize the stack machine that builds the CS-paths and that we can use to work out the details about how constructing the maximal CS-path just described.

The point of determining $R_{\Pi}(u)$. CS allows to define a weight $\mathrm{W}(\Pi)$, for every Π in a subsystem of MS. The weight has two relevant properties.



$(g, U, +) \mapsto_\Pi (h, U \cdot \mathtt{o}, +)$
$(e, U, -) \mapsto_\Pi (h, U \cdot \mathtt{a}, +)$

$(g, U, +) \mapsto_\Pi (e, U \cdot \mathtt{a}, -)$
$(h, U, -) \mapsto_\Pi (e, U \cdot \mathtt{o}, -)$

$(e, U, +) \mapsto_\Pi (h, U \cdot \mathtt{f}, +)$
$(g, U, +) \mapsto_\Pi (h, U \cdot \mathtt{x}, +)$

$(h, U \cdot \mathtt{f}, +) \mapsto_\Pi (e, U, +)$
$(h, U \cdot \mathtt{x}, +) \mapsto_\Pi (g, U, +)$

$(e, U, +) \mapsto_\Pi (g, U \cdot \mathtt{s}, +)$

$(e, U \cdot \mathtt{s}, +) \mapsto_\Pi (g, U, +)$

$(h, U \cdot \mathtt{l}(t), +) \mapsto_\Pi (e, U \cdot t, +)$
$(h, U \cdot \mathtt{r}(t), +) \mapsto_\Pi (g, U \cdot t, +)$

$(e, U, +) \mapsto_\Pi (g, U, +)$

$(e, U, +) \mapsto_\Pi (h, U, +)$

Fig. 11. Rewriting relation among contexts. If $(e, U, b) \mapsto_\Pi (e', U', b')$ then also $(e', U', -b') \mapsto_\Pi (e, U, -b)$, where $-b$ is the polarity opposed to b.

The first one is that the *r.steps* strictly decrease $\mathrm{W}(\Pi)$, for any Π. Namely, the weight bounds both the normalization time of Π and the size of the reducts of Π under a standard normalization strategy which is proved to be the worst one. So, the weight is a bound for any strategy.
The second property is that $\mathrm{W}(\Pi)$, for any Π, is dominated by $\sum_{b \in B_\Pi} R_\Pi(u)$, up to a polynomial. More formally, it exists a polynomial $p(x, y)$ such that, for every Π, $W(\Pi) \leq p\left(\sum_{b \in B_\Pi} R_\Pi(u), |\Pi|\right)$.
Spindles. They are the structure of proof nets whose absence allows to put a polynomial bound on $\sum_{b \in B_\Pi} R_\Pi(u)$, giving us a sufficient condition for PTIME-soundness.
Figure 12(a) shows an example of *spindle* between u and d. A spindle contains two, and only two, distinct CS-paths to go from one node of a proof net to another. The point is that two different CS-paths can sum up to duplicate structure, a potentially harmful behavior, when the control over the normalization complexity is a concern.

Definition 1 (Spindles and Dangerous Spindles). *Let* $\Pi \in \mathbf{PN}(\mathcal{P})$*;* $e \in E_\Pi$ *an edge labelled* $\mathtt{m}A$ *entering a contraction* u*;* $f \in E_\Pi$ *an edge outgoing a* $\mathbf{Po}$ *node* b*;* $g \in E_\Pi$ *an edge labelled* B *(possibly* $g = f$*);* $\partial(e) = \partial(f) = \partial(g)$*. A* spindle $\mathtt{m}A : \Sigma : B$ *between* e *and* f *(or also between* u *and* d*) is a triple of* CS*-paths:* τ *from* e *to* f *passing through the left conclusion of* u*;* ρ *from* e *to* f *passing through the right conclusion of* u*;* χ *from* f *to* g *(possibly empty); and such that* τ *and* ρ *are the only* CS*-paths connecting* e *with* f*.* e*,* g*,* u *are resp. the* principal premise, principal conclusion *and* principal contraction *of* Σ*. An edge that is premise (resp. conclusion) of*

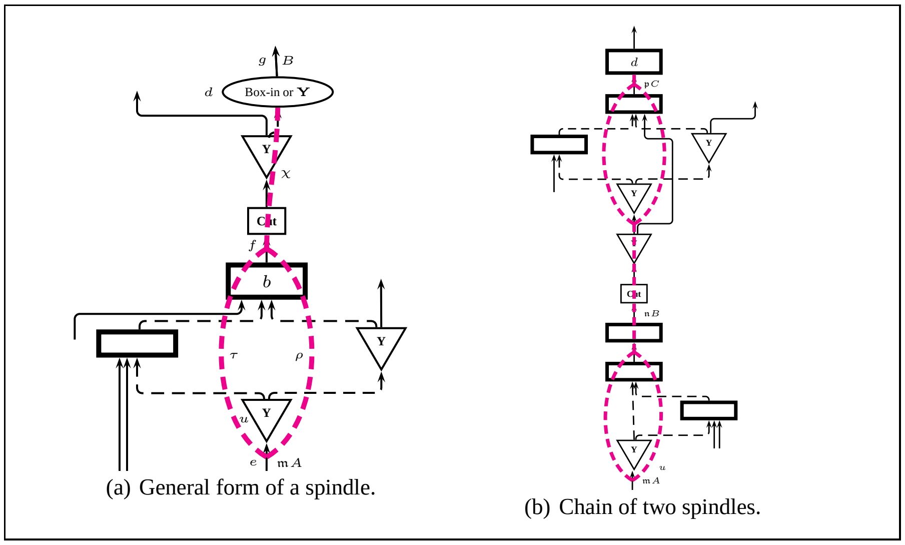


Fig. 12. A spindle and a chain of two spindles.

a node of Σ, but that is not part of Σ, is said non-principal *premise (resp. conclusion). We shorten $\mathfrak{m}A : \Sigma : \mathfrak{n}B$ with $\mathfrak{m} : \Sigma : \mathfrak{n}$. Finally, every $\mathfrak{m}A : \Sigma : \mathfrak{m}B$ is* dangerous.

Chains of spindles. The spindles that a subsystem can generate can be composed into chains. In Figure 12(b) there is a chain of two spindles $\mathfrak{m}A : \Sigma : \mathfrak{n}B$ and $\mathfrak{n}B : \Sigma' : \mathfrak{p}C$. Of course a concatenation can cut together an arbitrary number $r \geq 1$ of spindles $\mathfrak{m}_1 A_1 : \Sigma_1 : \mathfrak{m}_2 A_2, \ldots, \mathfrak{m}_r A_r : \Sigma_r : \mathfrak{m}_r A_r$ yielding a chain that we abbreviate as $\mathfrak{m}_1 A_1 : \Sigma_1; \ldots; \Sigma_r : \mathfrak{m}_r A_r$, or simply $\mathfrak{m}_1 A_1 : \Theta_r : \mathfrak{m}_r A_r$, with Θ_r equal to $\Sigma_1; \ldots; \Sigma_r$.
Intuitively, a subsystem of MS that can build arbitrarily long chains of spindles cannot be PTIME-sound essentially because we cannot bound the amount of duplicated structure during the normalization. For example, arbitrarily long chains exist as soon as the rules of a subsystem allow to build $\mathfrak{m}A : \Theta_r : \mathfrak{m}A$, which can compose with itself an arbitrary number L of times, yielding a chain $\mathfrak{m}A : \Theta_{r \cdot L} : \mathfrak{m}A$.
Finally, the sufficient condition for PTIME-soundness that extends the one in [8]:

Proposition 1 (PTIME-soundness: Sufficient Condition). *Let $\mathbb{X}$ be finite, and $\mathcal{P} \subseteq$ MS. If $\mathcal{P}$ cannot build dangerous spindles, then $\mathcal{P} \in$ PMS.*

Proof (Sketch). Both the absence of dangerous spindles in any proof net Π of $\mathcal{P}$ and the finiteness of $\mathbb{X}$ lead to a constant bound L on the length of the chains of spindles in Π. L only depends on $\mathcal{P}$. The bound L implies the existence of a polynomial that only depends on $\mathcal{P}$ and on the depth of Π, which bounds $\sum_{b \in B_\Pi} R_\Pi(u)$. For what observed above, this implies the PTIME-soundness of $\mathcal{P}$. □

3.2 Necessary condition for PTIME-soundness

Here we present a necessary condition for PTIME-soundness of subsystems in MS.

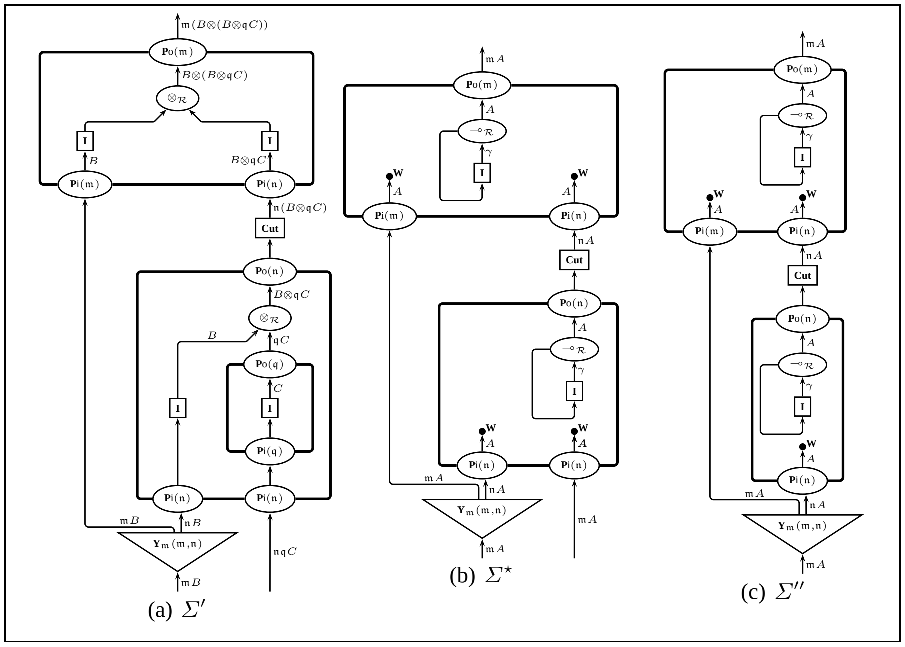


Fig. 13. Examples relative the proof of Lemma 1.

Proposition 2 (PTIME**-soundness Necessary Condition).** *Let $\mathcal{P} \subseteq$ MS. If $\mathcal{P} \in$ PMS then $\mathcal{P}$ cannot build dangerous spindles.*

The idea is that if a dangerous spindle $\mathfrak{m}A : \Sigma : \mathfrak{m}B$ exists in $\mathcal{P}$, then Σ can be transformed into another one $\mathfrak{m}C : \Sigma' : \mathfrak{m}C$ which, obviously, can be freely composed with itself, leading to an exponential blow-up. We now develop the proof of Proposition 2.

Fact 1. *Let $\mathcal{P} \subseteq$ MS.*
1. *For every $l \geq 0$, $\mathcal{P}$ contains $\Pi_l \triangleright A, \ldots, A \vdash A$, with $A = \gamma \multimap \gamma$, l occurrences of A as assumptions, and $\partial(\Pi_l) = 1$.*
2. *If $\mathcal{P}$ proves $A, \ldots, A \vdash A$, with l occurrences of A, then it proves also $\mathfrak{m}_1 A, \ldots, \mathfrak{m}_k A \vdash \mathfrak{q}A$ for $k < l$, for every $P_{\mathfrak{q}}(\mathfrak{m}_1, \ldots, \mathfrak{m}_k) \in \mathcal{P}$.*

For example, Π_l in Fact 1 can just contain the nodes $\multimap_{\mathcal{R}}$ and $\bullet$**w** .

Lemma 1. *Let $\mathcal{P} \subseteq$ MS be a subsystem that can build a (dangerous) spindle Σ. In $\mathcal{P}$ there is another (dangerous) spindle Σ'', obtained constructively from Σ, such that: (a) Σ'' contains only Contractions, Box-out and Cut nodes at level 0. (b) Every edge e at level 0 has label $\mathfrak{q}A$, for some fixed A. (c) $\partial(\Sigma'') = 1$. (d) The only premise of Σ'' is the principal one.*

Proof. Let $\Pi \in \mathbf{PN}(\mathcal{P})$ be the proof net containing Σ.

We reduce all the linear cuts in Π at level 0. We get to $\Pi' \in \mathbf{PN}(\mathcal{P})$ with Σ', the reduct of Σ, in it. An example of a possible Σ' is in Figure 13(a). Σ' is still a (dangerous) spindle, as well as the residuals of the three CS-paths ρ, τ, χ of Σ are still CS-paths in Π'. The three CS-paths cannot contain linear nodes. Indeed, we just observe that, e.g., τ must begin in a contraction, with label $\mathfrak{n}A$, and must stop in a box, with label $\mathfrak{m}B$. So it cannot cross neither any right-node, otherwise it would add a non-modal symbol to $\mathfrak{n}A$, and so the last formula could not be $\mathfrak{m}B$, nor any left-node, otherwise it would remove a non-modal symbol from $\mathfrak{n}A$.
We transform Σ', just generated, into another graph $\Sigma^\star$ by replacing every proof net enclosed in a box $P_{\mathfrak{q}}(\mathfrak{m}_1, \ldots, \mathfrak{m}_k)$ of Σ' by $\Pi_k \rhd A, \ldots, A \vdash A$ as in point 1 of Fact 1. Consequently, every $\mathfrak{q}B$ at level 0 in Σ' becomes $\mathfrak{q}A$ in $\Sigma^\star$, for some $\mathfrak{q}$. This replacement implies $\partial(\Sigma^\star) = 1$ (Figure 13(b)).
$\Sigma^\star$ is a graph satisfying the requirements (a)-(b)-(c) in the statement. However in general it does not satisfy (d). Moreover $\Sigma^\star$ is not necessarily a spindle, because it may be not contained in a proof net of $\mathcal{P}$. We can modify $\Sigma^\star$ to get a Σ'' that satisfies the point (d). If e is a non-principal premise of $\Sigma^\star$, entering some box b of $\Sigma^\star$, then we remove the edge e and we reduce the number of premises of b, according to point 2 of Fact 1. Now, we can plug every non-principal conclusion of Σ'', exiting upward from some contraction u of Σ'', with a weakening node: we get a proof net Π'' containing Σ'', thus showing that Σ'' is a spindle. If Σ was dangerous, Σ'' is too. □

In particular, $\mathfrak{n} : \Sigma : \mathfrak{n}$ dangerous implies that both the principal premise and conclusion of Σ'' have $\mathfrak{n}A$ as type.

Proof (of Proposition 2). By contraposition, we show that: "If $\mathcal{P}$ can build a dangerous spindle Σ, then it is not PTIME-sound." Let us assume $\mathfrak{m} : \Sigma : \mathfrak{m}$ has premises $\Gamma, \mathfrak{m}B$ and conclusions $\Delta, \mathfrak{m}C$, recalling that Σ may not be a proof net, so admitting more non-principal conclusions Δ. By Lemma 1 we build a dangerous spindle Σ'' of depth 1 with premise $\mathfrak{m}A$ and conclusions $\mathfrak{m}A, \tilde{\Delta}$. The dangerous spindle Σ'' becomes a proof net $\Pi'' \rhd \mathfrak{m}A \vdash \mathfrak{m}A$, using weakening. Now we concatenate as many copies of Π'' as we want, with a closed box $b \rhd \vdash \mathfrak{m}A$, obtaining a family $\langle \Theta_n \rhd \vdash \mathfrak{m}A \mid n \in \mathbb{N} \rangle$. Every Θ_n has depth 1 and size $O(n)$. The canonical strategy replicates b, until the level 0 is normal. This takes linear time. Though, the final size at level 1 has grown exponentially, implying that Θ_n reduces in time $O\left(2^n\right)$. So, $\mathcal{P} \notin$ PMS. □

Remark 1. The proof of Proposition 2 highlights a peculiar aspect of PTIME-sound subsystems. PTIME-soundness combines two more primitive properties we can call *polynomial step soundness* (pstep) and *polynomial size soundness* (psize). A subsystem Π is pstep (resp. psize) iff there is a polynomial $p(x)$ such that, for every $\Pi \in \mathcal{P}$, $\Pi \rightarrow^k \Sigma$ implies that k (resp. $|\Sigma|$) is bounded by $p\left(|\Pi|\right)$. So, the proof of Proposition 2 shows also that "*if $\mathcal{P} \subseteq$ MS is psize, then it is pstep as well, namely PTIME-sound*".

4 PTIME-maximal subsystems

This section supplies the local criterion that distinguishes PTIME-sound and PTIME-*un*sound subsystems of MS, just looking at their rules.

Definition 2 **(PTIME-maximal subsystems).** *Let $\mathcal{P} = (\mathbb{X}, \mathcal{B}_{\mathbb{X}}, \mathcal{R}_{\mathbb{X}})$ be a subsystem of* MS*. We say that $\mathcal{P}$ is* PTIME-maximal *if, for every $\mathcal{P}' = (\mathbb{X}, \mathcal{B}'_{\mathbb{X}}, \mathcal{R}'_{\mathbb{X}})$, $\mathcal{B}_{\mathbb{X}} \subset \mathcal{B}'_{\mathbb{X}}$ implies $\mathcal{P}' \notin$* PMS.

Let $\mathcal{P} \subseteq$ MS. We write $\mathfrak{m} \preceq \mathfrak{n}$ whenever there exists a proof net $\Pi \in \mathbf{PN}(\mathcal{P})$ containing a CS-path whose edges are all labeled by modal formulæ, the first one exiting upwards from a $\mathbf{Po}$ node with modality $\mathfrak{m}$, and the last one with modality $\mathfrak{n}$. $\preceq$ is clearly transitive.

Fact 2. *Let $\mathcal{P}$ be a* PTIME*-maximal subsystem.*

1. *If $\mathcal{P}$ has $P_{\mathfrak{q}}(\overrightarrow{\mathfrak{m}}, \mathfrak{n}, \mathfrak{n})$ it also has $P_{\mathfrak{q}}(\overrightarrow{\mathfrak{m}}, \mathfrak{n}^*)$, where $\mathfrak{n}^*$ stands for any unlimited and finite sequence of assumptions with a $\mathfrak{n}$-modal formula.*
2. *If $\mathcal{P}$ has $P_{\mathfrak{q}}(\mathfrak{m}, \mathfrak{n})$ and $P_{\mathfrak{q}}(\mathfrak{n}, \mathfrak{r})$ it also has $P_{\mathfrak{q}}(\mathfrak{m}, \mathfrak{n}, \mathfrak{r})$.*
3. *$\forall \mathfrak{q} \in \mathbb{X}$, $\mathcal{P}$ has at least $P_{\mathfrak{q}}(\mathfrak{q}^?)$, where $\mathfrak{n}^?$ stands for at most one assumption with a $\mathfrak{n}$-modal formula. This is like requiring a reflexive $\preceq$.*

The justification to the first point develops as follows. $\mathcal{P}$ cannot have $P_{\mathfrak{q}}(\overrightarrow{\mathfrak{m}}, \mathfrak{n}^*)$ if such a rule generates a spindle. In that case, the same spindle exists thanks to $P_{\mathfrak{q}}(\overrightarrow{\mathfrak{m}}, \mathfrak{n}, \mathfrak{n})$ against the PTIME-maximality of $\mathcal{P}$. An analogous argument can be used to justify the second point. The third one is obvious because $P_{\mathfrak{q}}(\mathfrak{q}^?)$ has at most one assumption.

Lemma 2. *Let $\mathcal{P}$ be a* PTIME*-maximal subsystem. Then $\preceq$ is a* linear quasi-order*, i.e. $\preceq$ is transitive, reflexive and connected.*

Proof. $\preceq$ is always transitive. Here it is also reflexive, because of the presence of $P_{\mathfrak{q}}(\mathfrak{q})$. We have to prove that it is connected, that is: $\forall \mathfrak{m}, \mathfrak{n}\ (\mathfrak{m} \preceq \mathfrak{n} \vee \mathfrak{n} \preceq \mathfrak{m})$. Let us assume that $\mathcal{P}$ is PTIME-maximal and $\neg(\mathfrak{m} \preceq \mathfrak{n})$. We show that $\mathfrak{n} \preceq \mathfrak{m}$. $\neg(\mathfrak{m} \preceq \mathfrak{n})$ implies that adding $P_{\mathfrak{n}}(\mathfrak{m})$ to $\mathcal{P}$ gives a PTIME-*un*sound system $\mathcal{P}'$. So we can find a dangerous spindle $\mathfrak{q} : \Sigma : \mathfrak{q}$ in $\mathcal{P}'$ containing $P_{\mathfrak{n}}(\mathfrak{m})$. This spindle is not in $\mathcal{P}$. However, in $\mathcal{P}$, we can find a graph Σ' that, extended with a suitable number of $P_{\mathfrak{n}}(\mathfrak{m})$ boxes, gives Σ. In particular, we find in Σ' (and so in $\mathcal{P}$) two paths, one from modality $\mathfrak{q}$ to $\mathfrak{m}$ and the other one from $\mathfrak{n}$ to $\mathfrak{q}$, that can be composed to create a path from $\mathfrak{n}$ to $\mathfrak{m}$. So, $\mathfrak{n} \preceq \mathfrak{m}$. □

Remark 2. We focus only on PTIME-maximal systems with a *linear order* $\preceq$. This because any $\mathcal{P}$ whose $\preceq$ is not as such is as expressive as another $\mathcal{P}'$ with less modalities and whose $\preceq$ is linear. For example, if $\mathcal{P}$ is PTIME-maximal and contains $P_{\mathfrak{n}}(\mathfrak{m})$ and $P_{\mathfrak{m}}(\mathfrak{n})$, then $\mathfrak{m}$ and $\mathfrak{n}$ cannot be distinguished, as $\preceq$ is transitive, so $\mathfrak{n}, \mathfrak{n}$ can be identified. We shall write $\prec$ for the strict order induced by $\preceq$.

Theorem 1 **(Structure of PTIME-maximal Subsystems).** *Let $\mathcal{P}$ be a subsystem of* MS *with a linear $\preceq$. $\mathcal{P}$ is* PTIME*-maximal iff $\mathcal{P}$ contains exactly the following rules:*

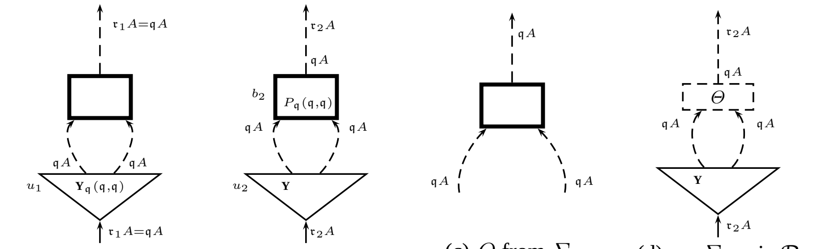
1. *all the rules $\mathbf{Y}_{\mathfrak{q}}(\mathfrak{m}, \mathfrak{n})$ for every $\mathfrak{q} \prec \mathfrak{m}, \mathfrak{n}$;*
2. *all the rules $\mathbf{Y}_{\mathfrak{q}}(\mathfrak{q}, \mathfrak{n})$ for every $\mathfrak{q} \prec \mathfrak{n}$;*
3. *all the rules $P_{\mathfrak{q}}(\mathfrak{q}^?, \mathfrak{m}_1^*, \dots, \mathfrak{m}_k^*)$ for every $\mathfrak{m}_1, \dots, \mathfrak{m}_k \prec \mathfrak{q}$;*
4. *only one among $\mathbf{Y}_{\mathfrak{q}}(\mathfrak{q}, \mathfrak{q})$ and $P_{\mathfrak{q}}(\mathfrak{q}^*, \mathfrak{m}_1^*, \dots, \mathfrak{m}_k^*)$, for every $\mathfrak{m}_1 \dots, \mathfrak{m}_k \prec \mathfrak{q}$.*

For example, LAL is not maximal. It can be extended to a PTIME-maximal system letting $! \prec \S$, and adding the missing rules.

Proof. "Only if" direction. We want to prove that every system $\mathcal{P}$ containing the rules described in points *1*, *2*, *3*, 4 is PTIME-maximal. The order $\preceq$ prevents to build dangerous spindles thanks to its linearity. So, Proposition 1 implies $\mathcal{P}$ is PTIME-sound. Moreover, if $\mathbf{Y}_{\mathfrak{q}}(\mathfrak{q}, \mathfrak{q})$ belongs to $\mathcal{P}$, adding $P_{\mathfrak{q}}(\mathfrak{q}^*, \mathfrak{m}_1^*, \ldots, \mathfrak{m}_k^*)$, against point *4*, would allow to construct a dangerous spindle. The same would be by starting with $P_{\mathfrak{q}}(\mathfrak{q}^*, \mathfrak{m}_1^*, \ldots, \mathfrak{m}_k^*)$ in $\mathcal{P}$ and, then, adding $\mathbf{Y}_{\mathfrak{q}}(\mathfrak{q}, \mathfrak{q})$.
"If" direction. We assume $\mathcal{P}$ is PTIME-maximal and we want show that it must contain at least all the rules described in the statement. We prove it by contradiction first relatively to points *1*, *2*, and *3*, then to point 4. A contradiction relative to points *1*, *2*, and *3* requires to assume that $\mathcal{P}$ has not a rule R described in *1*, *2*, *3*. Let $\mathcal{P}' = \mathcal{P} \cup \{R\}$. The PTIME-maximality of $\mathcal{P}$ implies that $\mathcal{P}'$ is not PTIME-sound. Now, thanks to the contraposition of Proposition 1, $\mathcal{P}' \notin$ PMS implies that $\mathcal{P}'$ can build a dangerous spindle $\mathfrak{r} : \Sigma : \mathfrak{r}$. We are going to see that this causes various kinds of contradictions.
Let u be an instance of $R = \mathbf{Y}_{\mathfrak{q}}(\mathfrak{m}, \mathfrak{n})$. For example, we can assume that one of the two distinct CS-paths in Σ crosses u from $\mathfrak{q}A$ to $\mathfrak{m}A$. By definition of $\preceq$, $\mathfrak{r} \preceq \mathfrak{q} \prec \mathfrak{m} \preceq \mathfrak{r}$, so contradicting the linearity of $\preceq$.
Let u be an instance of $R = \mathbf{Y}_{\mathfrak{q}}(\mathfrak{q}, \mathfrak{n})$. The previous case excludes that one of the two distinct CS-paths in Σ crosses u from $\mathfrak{q}A$ to $\mathfrak{n}A$. So, it must cross u from its premise, labeled $\mathfrak{q}A$ to its conclusion, labeled $\mathfrak{q}A$. This means that $\mathfrak{r} : \Sigma' : \mathfrak{r}$, obtained from Σ just removing u, exists in $\mathcal{P}$, which could not be PTIME-sound.
The case $R = P_{\mathfrak{q}}(\mathfrak{q}, \mathfrak{m}_1, \ldots, \mathfrak{m}_k)$ combines the two previous ones.
Now we move to prove point *4*. Without loss of generality, we can prove the thesis for $P_{\mathfrak{q}}(\mathfrak{q}, \mathfrak{q})$ instead of $P_{\mathfrak{q}}(\mathfrak{q}^*, \mathfrak{m}_1^*, \ldots, \mathfrak{m}_k^*)$. By contradiction, let $\mathcal{P}$ be PTIME-maximal, and let the thesis be false. So, there are $R_1 = \mathbf{Y}_{\mathfrak{q}}(\mathfrak{q}, \mathfrak{q})$ and $R_2 = P_{\mathfrak{q}}(\mathfrak{q}, \mathfrak{q})$ such that either $\mathcal{P}$ has both of them, *or* $\mathcal{P}$ has none of them. In the first case the two rules would build a spindle, making $\mathcal{P}$ not PTIME-sound. In the second case, neither R_1 nor R_2 belong to $\mathcal{P}$. This means that neither $P_1 = \mathcal{P} \cup \{R_1\}$, nor $\mathcal{P}_2 = \mathcal{P} \cup \{R_2\}$ are PTIME-sound, because, recall, $\mathcal{P}$ is PTIME-maximal. So, there has to exist both $\mathfrak{r}_1 : \Sigma_1 : \mathfrak{r}_1$ in $\mathcal{P}_1$ that involves an instance u_1 of R_1 (Figure 14(a)), and $\mathfrak{r}_2 : \Sigma_2 : \mathfrak{r}_2$ in $\mathcal{P}_2$ involving the instance b_2 of R_2 (Figure 14(b)). Thanks to Lemma 1 we can assume that all the labels in Σ_1 and Σ_2 are of the form $\mathfrak{n}A$, for a fixed A and some $\mathfrak{n}$'s. Then, u_1 must be the *principal* contraction of Σ_1. If not, we could eliminate it as we did for u in point 2, proving that $\mathcal{P}$ is not PTIME-sound. This is why $\mathfrak{r}_1 = \mathfrak{q}$ in Σ_1 of Figure 14(a). Now, let Θ be Σ_1 without u_1 that we can build in $\mathcal{P}$ (Figure 14(c)). Θ has two premises $\mathfrak{q}A$ and one conclusion $\mathfrak{q}A$, exactly as the box b_2 in Σ_2. So, we can replace b_2 in Σ_2 with Θ getting a new spindle $\mathfrak{r}_2 : \Sigma : \mathfrak{r}_2$ in $\mathcal{P}$. But $\Sigma \in \mathbf{PN}(\mathcal{P})$ implies $\mathcal{P}$ is not PTIME-sound. $\square$

5 Conclusions

We supply a criterion that discriminates PTIME-sound stratified systems, based on a quite general structure of proof nets, from the PTIME-*un*sound ones. Such systems are subsystems of the framework MS. Roughly, every subsystem is a rewriting system, based on proof nets. The proof nets are typable with modal formulæ whose modalities can be quite arbitrary. We justify the "arbitrariness" of deciding the set of modalities to



$\mathfrak{r}_1 A{=}\mathfrak{q}A$
$\mathfrak{q}A$
u_1
$\mathbf{Y}_\mathfrak{q}(\mathfrak{q},\mathfrak{q})$
$\mathfrak{r}_2 A$
b_2
$P_\mathfrak{q}(\mathfrak{q},\mathfrak{q})$
u_2
$\mathbf{Y}$
Θ

(a) $\mathfrak{r}_1 : \Sigma_1 : \mathfrak{r}_1$ in $\mathcal{P}_1$ (b) $\mathfrak{r}_2 : \Sigma_2 : \mathfrak{r}_2$ in $\mathcal{P}_2$ (c) Θ from Σ_1 (d) $\mathfrak{r}_2 : \Sigma : \mathfrak{r}_2$ in $\mathcal{P}$

Fig. 14. The proof nets used in the proof of Theorem 1

use by our will to extend as much as we can the set of programs-as-stratified-proof nets. To this respect, the "largest" subsystems are the PTIME-maximal ones that we can recognize thanks to the form of their proof nets building rules. The "state transition" from a PTIME-maximal subsystem to a PTIME-*un*sound one corresponds to moving from a system that composes chains of spindles with bounded length to a system with unbounded long chains.
Finally, we can state that MS accomplishes the reason we devised it. The subsystem sLAL in Section 2 can replace (2) in Fig. 1 for a suitable extension of BC^-, inside SRN, in place of (1). To show how, however, really requires a whole work, whose technical details will be included in Vercelli's doctoral thesis, which shall also present some conditions able to assure if a subsystem enjoys cut-elimination.

References

1. Bellantoni, S., Cook, S.: A new recursion-theoretic characterization of the polytime functions. Computational Complexity **2** (1992) 97–110
2. Asperti, A., Roversi, L.: Intuitionistic light affine logic. ACM Trans. Comput. Log. **3**(1) (2002) 137–175
3. Girard, J.Y.: Linear Logic. Theo. Comp. Sci. **50** (1987) 1–102
4. Murawski, A.S., Ong, C.H.L.: On an interpretation of safe recursion in light affine logic. Theor. Comput. Sci. **318**(12) (2004) 197—223
5. Terui, K.: Light affine lambda calculus and polynomial time strong normalization. Arch. Math. Logic **46**(3-4) (2007) 253–280
6. Beckmann, A., Weiermann, A.: A term rewriting characterization of the polytime functions and related complexity classes. Arch. Math. Logic **36**(1) (December 1996) 11 – 30
7. Girard, J.Y.: Light linear logic. Inf. Comput. **143**(2) (1998) 175–204
8. Roversi, L., Vercelli, L.: Some Complexity and Expressiveness results on Multimodal and Stratified Proof-nets. In: Proceedings of TYPES'08. Volume 5497 of LNCS., Springer (2009) 306 – 322 To appear. Available on the home pages of the authors.
9. Dal Lago, U.: Context semantics, linear logic and computational complexity. ACM Transactions on Computational Logic (2008) To appear.

Polynomial Time Computation in the Context of Recursive Analysis

Walid Gomaa

INRIA Nancy Grand-Est Research Centre, France,
Faculty of Engineering, Alexandria Univercsity, Alexandria, Egypt
walid.gomaa@loria.fr

Abstract. Recursive analysis was introduced by A. Turing [1936], A. Grzegorczyk [1955], and D. Lacombe [1955] as an approach for investigating computation over the real numbers. It is based on enhancing the Turing machine model by introducing oracles that allow the machine to access finitary portions of the real infinite objects. Classes of computable real functions have been extensively studied as well as complexity-theoretic classes of functions restricted to compact domains. However, much less have been done regarding complexity of arbitrary real functions. In this article we give a definition of polynomial time computability of arbitrary real functions. We survey some of the results based on this definition, in particular relationships with polynomial time computability of continuous rational functions. O. Bournez, E. Hainry, and myself have been developing a function algebra of real functions based on the predicative recursion philosophy developed by S. Bellantoni and S. Cook [1992] and was used by them to give an algebraic machine-independent characterization of the polynomial time computable integer functions. The proposed algebra is able to capture the Bellantoni-Cook class, however, it is still weaker than the whole polynomial time real computability.

1 Introduction

The theory of *recursive analysis* was introduced by A. Turing in 1936 [23], A. Grzegorczyk in 1955 [14], and D. Lacombe in 1955 [17]. It is an approach for investigating computation over the real numbers that is based on enhancing the normal *Turing machine model* by introducing oracles that provide access to the $\mathbb{R}$-inputs. Recursive analysis is a reductionist approach where the real number is deconstructed into some finitary representation such as Cauchy sequences. Compare this with the Blum-Shub-Smale (BSS) model [3, 19] where real numbers are considered as atomic entities that can be written directly on the machine tape cells. Given a function $f\colon \mathbb{R} \to \mathbb{R}$, computability of f in the context of recursive analysis simply means the existence of a Turing machine that when successively fed with increasingly accurate representation of $x \in \mathbb{R}$, it will be able to successively output increasingly accurate representation of the function value $f(x)$. Turing machines represent discrete-time discrete-space model of $\mathbb{R}$-computation; they are finitary objects, hence only countable number of real functions are computable.

Unlike the case of discrete computation, which has enjoyed a kind of uniformity and conceptual universality, there have been several different approaches to real computation some of which are not even comparable. A survey on the wide spectrum of models of $\mathbb{R}$-computation is written by P. Orponen in 1997 [22]. An up-to-date version of Orponen's has been written by O. Bournez and M. Campagnolo in 2008 [21]. V. Blondel and J. Tsitsiklis wrote a more specialized survey about the role of classical computational complexity in systems and control theory [2]. A dedicated survey about the class of BSS models was written by K. Meer and C. Michaux [19]; this survey essentially focuses on the complexity-theoretic aspects of these models. A less detailed survey of the main BSS model is given in [11]; this article focuses on extending the Grzegorczyk hierarchy to the reals through that model [12]. In spite of this diversity recursive analysis seems to be the most practical and physically realizable approach to real computation and so is typically considered as the most plausible theoretical framework for numerical analysis algorithms.

Much of the current research have been directed towards giving machine-independent characterizations of recursive analysis. This is mostly done through the use of *function algebras*.

Definition 1 (Function algebra). *A function algebra $\mathcal{F} = [\mathcal{B}; \mathcal{O}]$ is the smallest class of functions that contains some basic functions $\mathcal{B}$ and their closure under a set of operations $\mathcal{O}$. An operation is a functional that takes a finite set of functions as arguments and returns a new function.*

A typical example is Kleene's class that characterizes discrete computability:

$$\mathcal{K} = [0, s, P; Comp, Rec, \mu] \tag{1}$$

It contains the constant 0, successor function s, a set of projection functions P, and closed under the operations of composition, primitive recursion, and minimalization. Naturally, when moving to the reals primitive recursion is replaced by differential equations. Minimalization has been extended to the reals in various ways (see for example, [20],[5]), however, this operator is not intuitive over $\mathbb{R}$ and does not appear natural especially from the mathematical analysis perspective. Several articles have addressed this issue by replacing minimalization by strong forms of differential equations as well as different forms of the limit operator, see for example, [4, 9, 18, 15, 6, 8, 7]. It seems that the limit operator is essential in capturing the whole class of recursive analysis functions.

From the complexity perspective much work have been devoted to investigate real functions defined only over compact domains. The interested reader may consult [16]. Much less has been done regarding the complexity-theoretic properties of arbitrary real functions. The current article is an attempt in that direction. A definition of polynomial time real computability over arbitrary domains is given along with a quick survey of the results based on that definition, in particular relationships with polynomial time rational computation. O. Bournez, E. Hainry, and myself have been developing a function algebra of real functions based on the philosophy of *predicative recursion* developed by S. Bellantoni and S. Cook in 1992 [1]. The proposed class coincides with polynomial time integer computation, however, it can only partially capture polynomial time real computation.

We will assume the typical binary alphabet $\{0, 1\}$, so for convenience we restrict our attention to the set of dyadic rationals: $\mathbb{D} = \{d \in \mathbb{Q} \colon d = \frac{a}{2^b}, a \in \mathbb{Z}, b \in \mathbb{N}\}$; these are the rationals with finite binary notation. Notice that $\mathbb{D}$ forms a dense subset of $\mathbb{R}$.

The article is organized as follows. Section 1 is an introduction. Section 2 discusses polynomial time computation over $\mathbb{D}$ and the role of continuity in such computation. Section 3 introduces the basic concepts of recursive analysis, defines polynomial time computation of real functions over non-compact domains, and contrasts with the corresponding notion over the rationals. Section 4 introduces the Bellantoni-Cook class which is a function algebra that captures polynomial time integer computation. Based on this class a function algebra of real functions is proposed in Section 5 and the main properties of this class are proved. The article ends with a conclusion in Section 6.

2 Polynomial Time Dyadic Computation

In the following we consider domains of the form (a, b) where $a \in \mathbb{R} \cup \{-\infty\}$ and $b \in \mathbb{R} \cup \{\infty\}$. When restricted to $\mathbb{D}$ it will be denoted $(a, b)_{\mathbb{D}}$. For $d \in \mathbb{D}$, let $\alpha(d)$ denote the binary representation of d.

Definition 2 (Polynomial time computability of dyadic functions). *Assume a function $f \colon (a, b)_{\mathbb{D}} \to \mathbb{D}$. We say that f is* polynomial time computable *if there exists a Turing machine M such that for every $d \in (a, b)_{\mathbb{D}}$ the following holds:*

$$M(\alpha(d)) = \alpha(f(d))$$

and the computation time of M is bounded by $p(|\alpha(d)|)$ for some polynomial function p.

Dyadics are finite objects, hence the computation is exact. Next we define a notion of continuity for dyadic functions.

Definition 3 (Continuous dyadic functions).

1. *Assume a function $f \colon \mathbb{D} \to \mathbb{D}$. We say that f is continuous if f has a modulus of continuity, that is, if there exists a function $m \colon \mathbb{N}^2 \to \mathbb{N}$ such that for every $k, n \in \mathbb{N}$ and for every $x, y \in [-2^k, 2^k]$ the following holds:*

$$if\ |x - y| \leq 2^{-m(k,n)}, then\ |f(x) - f(y)| \leq 2^{-n} \tag{2}$$

2. *Assume a function* $f\colon (a,b)_{\mathbb{D}} \to \mathbb{D}$ *where* $a, b \in \mathbb{D}$. *We say that* f *is continuous if* f *has a* modulus of continuity*, that is, if there exists a function* $m\colon \mathbb{N}^2 \to \mathbb{N}$ *such that for every* $k, n \in \mathbb{N}$ *and for every* $x, y \in [a+2^{-k}, b-2^{-k}]$ *the following holds:*
$$if\ |x-y| \leq 2^{-m(k,n)}, then\ |f(x)-f(y)| \leq 2^{-n} \tag{3}$$
3. *Given the two previous cases other domains can be handled in the obvious way.*

In the following k will be referred to as the *extension argument* and n as the *precision argument*. In the previous definition continuity of a dyadic function f over an open domain is reduced to continuity over successively enlarging compact subintervals of the domain. Compact domains are usually controllable, for example, the continuous function is always bounded over any compact subinterval of its domain. As will be seen below besides determining the continuity of a function, the modulus also controls how smooth and well behaved (especially from the real computation perspective) the function is. It is clear that the completion of a continuous dyadic function gives a continuous real function with the same domain plus the limit points and the same range plus the limit points, and more importantly it has the same modulus. Unlike the case of real computation where continuity is a necessary condition we can easily give an example of a computable dyadic function that is discontinuous. Furthermore, the smoothness of a dyadic function does not affect its complexity as indicated by the following proposition.

Proposition 1 ([13]). *There exist polynomial time computable continuous dyadic functions that have arbitrarily large moduli.*

Proof. (outline) Construct a dyadic function f that is piecewise linear and grows polynomially in terms of the whole length of the input, hence it is continuous and polynomial time computable. However, f grows exponentially (or any superpolynomial growth) in terms of the length of the integer part of the input which makes the function very steep and have exponential lower bound for the modulus.

3 Polynomial Time Real Computation

3.1 Representation of real numbers

Real numbers are infinite objects so in order to perform computations over them using Turing machines, which are inherently finite objects, we must have a finitary representation of $\mathbb{R}$. Given $x \in \mathbb{R}$, there are several such representations for x among which are the following:

1. Binary expansion: x is represented by a function $\psi_x\colon \mathbb{N} \cup \{-1\} \to \mathbb{N} \cup \{-1\}$, such that $\psi_x(-1) \in \{-1, 1\}$ (the sign of x) and $\psi_x(k) \in \{0, 1\}$ for every $k \geq 1$. Then
$$x = \psi_x(-1) \cdot (\psi_x(0) + \sum_{k \geq 1} \psi_x(k) \cdot 2^{-k}) \tag{4}$$

2. Left cut: x is represented by the set $L_x = \{d \in \mathbb{D} \colon d < x\}$.
3. Cauchy sequence: x is represented by a Cauchy function $\varphi_x \colon \mathbb{N} \to \mathbb{D}$ that *binary converges* to x

$$\forall n \in \mathbb{N} : |\varphi_x(n) - x| \leq 2^{-n} \tag{5}$$

So from this given sample of representations we see that x is represented either by a set of finite objects or a by a function over finite objects. We say that x is *computable* with respect to some representation $\mathcal{R}$ if there exists a Turing machine that either decides the set or computes the function that represent x with respect to $\mathcal{R}$. Examples of computable real numbers include the rationals, algebraic numbers such as $\sqrt{2}$, and transcendental numbers such as π and e. An important remark is that all the above representations give the same class of computable real numbers, however, on the sub-computable and complexity-theoretic levels they induce different classes. In particular we have:

Proposition 2. $PTime^{[BE]} \equiv PTime^{[LC]} \subsetneq PTime^{[CF]}$

For the remaining part of this article we adopt the Cauchy representation. For $x \in \mathbb{R}$ let CF_x denote the set of all Cauchy sequences that represent x. Apparently, only a countable subset of the real numbers are polynomial time computable; this set forms a real closed field.

3.2 Polynomial computation of real functions

For a comprehensive treatment of polynomial time computation of real functions over compact domains consult [16]. One of the basic results of recursive analysis is that *continuity is a necessary condition for computing real functions* and as we will see below the smoothness of the function plays an essential role in its complexity.

Definition 4. *(Polynomial time computation of real functions) Assume a function $f \colon \mathbb{R} \to \mathbb{R}$. We say that f is polynomial time computable if the following conditions hold:*

1. *There exists a function oracle Turing machine M such that for every $x \in \mathbb{R}$, for every $\varphi_x \in CF_x$, and for every $n \in \mathbb{N}$ the following holds:*

$$|M^{\varphi_x}(n) - f(x)| \leq 2^{-n} \tag{6}$$

2. *The* computation time *of $M^{\varphi_x}(n)$ is bounded by $p(k, n)$ for some polynomial p, where $k = \min\{j \colon x \in [-2^j, 2^j]\}$.*

Assume a function $g \colon (a, b) \to \mathbb{R}$ where $a, b \in \mathbb{R}$. Fix some $\varphi_a \in CF_a$ and some $\varphi_b \in CF_b$. We say that g is polynomial time computable if the following conditions hold:

1. *There exists a three-function oracle Turing machine N such that for every $x \in (a, b)$, for every $\varphi_x \in CF_x$, and for every $n \in \mathbb{N}$ the following holds:*
$$|N^{\varphi_x,\varphi_a,\varphi_b}(n) - f(x)| \leq 2^{-n} \tag{7}$$
2. *The computation time of $N^{()}(n)$ is bounded by $p(k, n)$ for some polynomial p, where $k = \min\{j : x \in [a + 2^{-j}, b - 2^{-j}]\}$.*

Remark 1. Some remarks about Definition 4:

1. Intuitively, polynomial time computation of real functions (in the context of recursive analysis) implies the existence of a Turing machine that when successively fed with a Cauchy sequence that represents the input it will be able to successively print out a Cauchy sequence that represents the function value. And the machine does that efficiently with respect to the particular components of the Cauchy sequences. This view is more explicit in the approach taken by K. Weihrauch [24].
2. In this definition polynomial time computability over a non-compact domain is reduced to polynomial time computability over compact subintervals of the function domain. This is made possible by the continuity of the underlying function. Compact intervals are easier to handle.
3. In addition to the extension argument k which exclusively accounts for the complexity of rational functions, the precision argument is part of the complexity measure of real functions.
4. The precision argument is assumed in its unary notation when measuring the complexity. Intuitively, this argument implies the required length of the fractional part of the machine output.
5. The interpretation of the argument k differs in the two kinds of functions mentioned in the definition. For unbounded domains it accounts for the length of the integer part of the input, whereas for open bounded domains it accounts for how far the input is from the boundary points.

Let $P_{\mathbb{R}}$ denote the class of polynomial time computable real functions. The following theorem relates the computational and the analytic properties of real functions.

Theorem 1 ([13]). *Assume a function $f : \mathbb{R} \to \mathbb{R}$. Then f is polynomial time computable iff there exist two functions over finite objects: $m : \mathbb{N}^2 \to \mathbb{N}$ and $\psi : \mathbb{D} \times \mathbb{N} \to \mathbb{D}$ such that*

1. *m is a modulus function for f and it is polynomial with respect to both the extension parameter k and the precision parameter n, that is, $m(k, n) = (k + n)^b$ for some $b \in \mathbb{N}$,*
2. *ψ is an approximation function for f, that is, for every $d \in \mathbb{D}$ and every $n \in \mathbb{N}$ the following holds:*
$$|\psi(d, n) - f(d)| \leq 2^{-n} \tag{8}$$
3. *$\psi(d, n)$ is computable in time $p(|d| + n)$ for some polynomial p.*

Proof. (outline) Assume the existence of m and ψ that satisfy the given conditions. Assume an f-input $x \in \mathbb{R}$ and let $\varphi_x \in CF_x$. Assume $n \in \mathbb{N}$. Let $M^{\varphi_x}(n)$ be an oracle Turing machine that does the following:

1. let $d_1 = \varphi_x(2)$,
2. use d_1 to determine the least k such that $x \in [-2^k, 2^k]$ (taking into account the error in d_1),
3. let $\alpha = m(k, n+1)$ (locating the appropriate component of the Cauchy sequence of x),
4. let $d = \varphi_x(\alpha)$,
5. let $e = \psi(d, n+1)$ and output e.

Note that every step of the above procedure can be performed in polynomial time with respect to both k and n. Now verifying the correctness of $M^{()}(n)$:

$$\begin{aligned}
|e - f(x)| &\leq |e - f(d)| + |f(d) - f(x)| \\
&\leq 2^{-(n+1)} + |f(d) - f(x)|, \qquad \text{by definition of } \psi \\
&\leq 2^{-(n+1)} + 2^{-(n+1)}, \qquad |d - x| \leq 2^{-m_k(n+1)} \text{ and definition of } m \\
&= 2^{-n}
\end{aligned}$$

This completes the first part of the proof. As for the reverse direction the existence of the efficiently computable approximation function ψ follows directly from the existence of a Turing machine $M^{()}$ that efficiently computes f. Using the Heine-Borel compactness theorem and the computation of $M^{()}$ a finite open covering of the compact interval $[-2^k, 2^k]$ can be effectively found and the modulus then follows almost immediately.

As indicated by the previous theorem, the modulus function plays a crucial role in the computability and complexity of real functions. On one hand it enables the machine to access an appropriate finite approximation of the input (which component of the Cauchy sequence). On the other hand it enables evaluating the machine output: how good it is as an approximation to the actual desired value.

Proposition 1 and Theorem 1 directly imply the following.

Theorem 2. *There exist continuous dyadic functions that are polynomial time computable, however, their extensions to $\mathbb{R}$ are not polynomial time computable.*

The converse of this also holds as indicated in the next theorem.

Theorem 3 ([13]). *There exists a dyadic-preserving function $g\colon \mathbb{R} \to \mathbb{R}$ such that g is polynomial time computable, however, $g \restriction \mathbb{D}$ is not polynomial time computable.*

Proof. (outline) Construct a function $g\colon \mathbb{R} \to \mathbb{R}$ such that g is piecewise linear, preserves $\mathbb{D}$, and is smooth enough (has a polynomial modulus), hence g is polynomial time computable as a real function. However, for any $x \in \mathbb{D}$, the fractional part of $g(x)$ has an exponential length with respect to the length of x. Hence, $g \restriction \mathbb{D}$ is not polynomial time computable.

The last two theorems indicate that polynomial time real computation is not simply an extension of the corresponding notion over the rationals. This can be justified from both directions as follows. Going from the rationals to the reals we saw that continuity and smoothness do not play any role in the computability and complexity of rational functions; whereas the situation is completely the opposite in the case of real functions where continuity is a necessary condition for computability and smoothness is a necessary condition for efficient computability. On the other hand going from the reals down to the rationals a fundamental difference can be easily observed: real computation is *approximate* whereas rational computation is *exact*. Looking at the function constructed in Theorem 3 we see that since rational computation is exact the whole long fraction of the function value must be printed out by the machine. Whereas in the real computation case generally the fraction is partially printed out and this part is accounted for by the precision argument.

4 The Bellantoni-Cook Class

The first characterization of the polynomial time discrete class was given by A. Cobham in 1964 [10]. However, Cobham's class still contains explicit reference to resource bounds. This was alleviated in the work of S. Bellantoni and S. Cook in 1992 [1]. Their philosophy is that the functions that are not polynomial time computable are inherently impredicative, that is, their definitions presume the existence of totalities such as the whole set of natural numbers. So in order to capture feasible polynomial time computation, only predicative operators should be applied to generate new functions from a small basic set of efficiently computable ones. They achieve that through the use of two types of function arguments: *normal* arguments and *safe* arguments. Any function in this class would have the general from $f(x_1, \ldots, x_m; y_1, \ldots, y_n)$ where the normal arguments $x_1, \ldots, x_m$ come first, followed by the safe ones $y_1, \ldots, y_n$ with the ';' separating these two groups. The class is defined as follows.

Definition 5. *Define the following class of functions that operate on binary strings.*

$$B = [0, U, s_0, s_1, pr, cond; SComp, SRec] \tag{9}$$

1. *a zero-ary function for the constant* 0*:* $0(;) = 0$*,*
2. *a set of projection functions* $U_i^{m+n}(x_1, \ldots, x_m; x_{m+1}, \ldots, x_{m+n}) = x_i$*,*
3. *two successor functions:* $s_i(;x) = x \circ i$ *for* $i \in \{0,1\}$*; this arithmetically corresponds to* $2x + i$*,*
4. *a predecessor function:*

$$pr(;x) = \begin{cases} 0 & x = 0 \\ y & x = y \circ i \end{cases}$$

5. *a conditional function*

$$cond(;x,y,z) = \begin{cases} y & x \equiv_2 0 \\ z & ow \end{cases} \tag{10}$$

6. *safe composition operator SComp: assume a vector of functions $\bar{g}_1(\bar{x};) \in B$, a vector of functions $\bar{g}_2(\bar{x};\bar{y}) \in B$, and a function $h \in B$ of arity $|\bar{g}_1| + |\bar{g}_2|$. Define new function f*

$$f(\bar{x};\bar{y}) = h(\bar{g}_1(\bar{x};);\bar{g}_2(\bar{x};\bar{y}))$$

7. *safe recursion on notation operator SRec: assume functions $g, h_0, h_1 \in B$, define a new function f as follows:*

$$\begin{aligned} &f(0,\bar{y};\bar{z}) = g(\bar{y};\bar{z}) \\ &f(s_i(;x),\bar{y};\bar{z}) = h_i(x,\bar{y};\bar{z},f(x,\bar{y};\bar{z})) \end{aligned}$$

Several observations are in place here: (1) all the basic functions (except projections) have only safe arguments, (2) all the basic functions increase the length of the input by at most a constant amount, this leads to the fact that safe arguments contribute only by a constant amount to the output of the function, (3) in *SComp* the arguments $\bar{y}$ which appear safe with respect to f do not appear in normal positions with respect to h, this means that safe arguments can not be repositioned as normal arguments; that is why the functions in $\bar{g}_1$ must have exclusively normal arguments, (4) in *SComp* both kinds of arguments $\bar{x}, \bar{y}$ of f appear in safe positions with respect to h, this means that normal arguments can be repositioned as safe ones; because of this asymmetry adding a function that operates on safe arguments to B is generally more powerful than adding the same function operating only on normal arguments, and (5) in *SRec* the recursion variable x appears in the defined function f in a normal position, whereas the recurred value $f(x,\bar{y};\bar{z})$ appears in a safe position in the defining function h.

The main result concerning the class B is the following.

Theorem 4. *([1]) The class of polynomial time computable discrete functions is exactly captured by those functions in B that have exclusively normal arguments.*

5 Partially Capturing $P_{\mathbb{R}}$

In the following we define a class of real functions which are essentially extensions to $\mathbb{R}$ of the Bellantoni and Cook class [1]. That former class was developed as a joint work with O. Bournez and E. Hainry. For any $n \in \mathbb{Z}$ we call $[2n, 2n+1]$ an even interval and $[2n+1, 2n+2]$ an odd interval.

Definition 6. *Define the function algebra*

$$\mathcal{W} = [0, U, s_0, s_1, pr_0, pr_1, \theta_1, e, o; SComp, SI, Lin] \tag{11}$$

1. *successor functions, $s_i(;x) = 2x + i$ for $i \in \{0, 1\}$,*

2. *two predecessor functions*

$$pr_0(;x) = \begin{cases} n & 2n \leq x \leq 2n+1, n \in \mathbb{Z} \\ n + (\epsilon - 1) & 2n+1 \leq x \leq 2n+\epsilon, 1 \leq \epsilon \leq 2 \end{cases}$$

$$pr_1(;x) = \begin{cases} n + \epsilon & 2n \leq x \leq 2n+\epsilon, 0 \leq \epsilon \leq 1 \\ n+1 & 2n+1 \leq x \leq 2n+2 \end{cases}$$

So the function pr_0 acts as $\lfloor \frac{x}{2} \rfloor$ over even intervals and piecewise linear otherwise, whereas the function pr_1 acts as $\lceil \frac{x}{2} \rceil$ over odd intervals and piecewise linear otherwise.

3. *a continuous function to sense inequalities, $\theta_1(;x) = \max\{0, x\}$,*
4. *parity distinguishing functions*

$$e(;x) = \frac{\pi}{2}\theta_1(sin\pi x)$$
$$o(;x) = \frac{\pi}{2}\theta_1(-sin\pi x)$$

The function e is non-zero only over even intervals and conversely o is non-zero only over odd intervals. Notice that on integer values $e(;k) = o(;k) = 0$.

5. *safe integration operator SI: assume functions g, h_0, h_1, define a new function f satisfying*

$$\begin{aligned} f(0,\bar{y};\bar{z}) =& g(\bar{y};\bar{z}) \\ \partial_x f(x,\bar{y};\bar{z}) =& e(x;)[h_1(pr_0(x;),\bar{y};\bar{z}, f(pr_0(x;),\bar{y};\bar{z})) \\ &- h_0(pr_0(x;),\bar{y};\bar{z}, f(pr_0(x;),\bar{y};\bar{z}))] \\ &+ o(x;)[h_0(pr_1(x;),\bar{y};\bar{z}, f(pr_1(x;),\bar{y};\bar{z})) \\ &- h_1(pr_1(x;)-1,\bar{y};\bar{z}, f(pr_1(x;)-1,\bar{y};\bar{z}))] \end{aligned}$$

Notice that for simplicity we misused the basic functions so that their arguments are now in normal positions (the alternative is to redefine a new set of basic functions with arguments in normal positions).

6. *A linearization operator Lin: given functions g, h, define a new function f by*

$$f(x,\bar{y};\bar{z}) = \begin{cases} \delta h(2pr_0(x;)+1,\bar{y};\bar{z}) + (1-\delta)g(2pr_0(x;),\bar{y};\bar{z}) & e(;x) \geq o(;x) \\ \delta' g(2pr_1(x;),\bar{y};\bar{z}) + (1-\delta')h(2pr_1(x;)-1,\bar{y};\bar{z}) & o(;x) \geq e(;x) \end{cases}$$

where $\delta = x - 2pr_0(x;)$, $\delta' = x + 1 - 2pr_1(x;)$. Note that at even integers f reduces to g whereas at odd integers it reduces to h.

The class $\mathcal{W}$ as defined above is an attempt to capture efficient real computation in a syntactical way that depends on the particular binary representation of numbers. We believe that the *syntactical approach* is more appropriate for characterizing efficient lower complexity classes (at least in the Turing mechanistic framework) for it provides means to control the complexity growth of a function through the exclusive use of predicative definitions. On the other hand, more abstract definitions are typically inherently impredicative leading to uncontrollable growth of the functions being defined. In the following we list some of the properties of $\mathcal{W}$ starting with the fact that it preserve $\mathbb{N}$.

Proposition 3. *Let $f \in \mathcal{W}$. Then $f \restriction \mathbb{N} \colon \mathbb{N} \to \mathbb{N}$.*

Proof. Proof is by induction on the construction of functions in $\mathcal{W}$. Note that for integer values the parity functions e and o are always 0. It is easy to see that the other basic functions preserve the integers and that composition preserves this property. Assume two functions $g, h \in \mathcal{W}$ that preserve $\mathbb{N}$ and consider the application of linearization. Given an input $x = 2n$ any of the two cases of the definition of f can be applied (with $\delta = 0$ and $\delta' = 1$) to give $f(\bar{y}; 2n, \bar{z}) = g(\bar{y}; 2n, \bar{z})$ which is integer. Alternatively, given $x = 2n+1$ then again any of the two cases can be applied (with $\delta = 1$ and $\delta' = 0$) to give $f(\bar{y}; 2n+1, \bar{z}) = h(\bar{y}; 2n+1, \bar{z})$ which is again an integer by the induction hypothesis. Assume functions $g, h_0, h_1 \in \mathcal{W}$ that preserve $\mathbb{N}$ and consider the application of the safe integration operator to define a new function $f \in \mathcal{W}$. We then use strong induction over the integration variable to show f preserves $\mathbb{N}$. The base case $f(0, \bar{y}; \bar{z}) = g(\bar{y}; \bar{z})$ holds by assumption on g. Assume $f(j, \bar{y}; \bar{z}) \in \mathbb{N}$ for every integer $j \leq 2n$ and consider an input $x \in [2n, 2n+1]$. Then $o(; x) = 0$, $e(; x) \neq 0$, and

$$\begin{aligned}&h_1(pr_0(x;), \bar{y}; \bar{z}, f(pr_0(x;), \bar{y}; \bar{z})) - h_0(pr_0(x;), \bar{y}; \bar{z}, f(pr_0(x;), \bar{y}; \bar{z})) = \\ &h_1(n, \bar{y}; \bar{z}, f(n, \bar{y}; \bar{z})) - h_0(n, \bar{y}; \bar{z}, f(n, \bar{y}; \bar{z}))\end{aligned}$$

This latter difference is independent of the integration variable x. Furthermore, by the hypotheses of the main and secondary inductions it is an integer value. Notice also that

$$\begin{aligned}\int_{2n}^{2n+1} e(; u) du &= \frac{\pi}{2} \int_{2n}^{2n+1} \theta_1(; \sin \pi u) du \\ &= \frac{\pi}{2} \int_{2n}^{2n+1} \sin \pi u \, du = 1\end{aligned}$$

This implies that

$$f(2n+1, \bar{y}; \bar{z}) = f(2n, \bar{y}; \bar{z}) + h_1(n, \bar{y}; \bar{z}, f(n, \bar{y}; \bar{z})) - h_0(n, \bar{y}; \bar{z}, f(n, \bar{y}; \bar{z}))$$

which is an integer value. Similarly for doing induction over odd intervals. Hence, the safe integration operator preserves $\mathbb{N}$. This completes the proof of the proposition.

Let $dp(\mathcal{W}) = \{f \restriction \mathbb{N} \colon f \in \mathcal{W}\}$. The following theorem shows that the discrete part of $\mathcal{W}$ coincides with the Bellantoni-Cook class implying that the class $\mathcal{W}$ exactly captures polynomial time discrete computability.

Theorem 5. $dp(\mathcal{W}) = B$

Proof. $B \subseteq dp(\mathcal{W})$: proof is by induction on the construction of functions in B. It is obvious that the basic functions $0, U, s_0, s_1$ exist in $dp(\mathcal{W})$. The function pr_0 from $\mathcal{W}$ acts exactly like pr when restricted to $\mathbb{N}$. Using U we can define the identity function inside $\mathcal{W}$. Then using linearization we can define the function:

$$f(;x,y,z) = \begin{cases} \delta z + (1-\delta)y & e(;x) \geq o(;x) \\ \delta' y + (1-\delta')z & o(;x) \geq e(;x) \end{cases} \tag{12}$$

where $\delta = x - 2pr_0(;x)$, $\delta' = x + 1 - 2pr_1(;x)$. It can be easily verified that $f(;2n,y,z) = y$ and $f(;2n{+}1,y,z) = z$ for every $n \in \mathbb{N}$, hence $f \restriction \mathbb{N} = cond$. The case for safe composition is easy. Now assume $f \in B$ that is defined from g, h_0, h_1 by safe recursion. Then by the induction hypothesis there exist $\tilde{g}, \tilde{h}_0, \tilde{h}_1 \in \mathcal{W}$ such that $\tilde{g} \restriction \mathbb{N} = g$, $\tilde{h}_0 \restriction \mathbb{N} = h_0$, and $\tilde{h}_1 \restriction \mathbb{N} = h_1$. Define the function $\tilde{f} \in \mathcal{W}$ using safe integration from $\tilde{g}, \tilde{h}_0$, and $\tilde{h}_1$. We claim that $\tilde{f} \restriction \mathbb{N} = f$. Proof is by strong induction over the recursion variable. For readability we will exclude the $\bar{y}$ and $\bar{z}$ arguments. At the base case we have $\tilde{f}(0;) = \tilde{g}(;) = g(;) = f(0;)$. From the proof of Proposition 3 we have

$$\begin{aligned}
\tilde{f}(2n+1;) &= \tilde{f}(2n;) + \tilde{h}_1(n;\tilde{f}(n;)) - \tilde{h}_0(n;\tilde{f}(n;)) \\
&= f(2n;) + \tilde{h}_1(n;f(n;)) - \tilde{h}_0(n;f(n;)), \quad \textit{by induction hypothesis} \\
&= f(2n;) + h_1(n;f(n;)) - h_0(n;f(n;)), \quad \textit{by assumption} \\
&= h_0(n;f(n;)) + h_1(n;f(n;)) - h_0(n;f(n;)), \quad \textit{by safe recursion} \\
&= h_1(n;f(n;)) \\
&= f(2n+1;), \quad \textit{by safe recursion}
\end{aligned}$$

Similarly, it can be shown that $\tilde{f}(2n+2;) = f(2n+2;)$. This completes the first part of the proof.

$dp(\mathcal{W}) \subseteq B$: proof is by induction on the construction of functions in $\mathcal{W}$ and using Proposition 3. The case for the basic functions $0, U, s_i, pr_0$ is obvious. For $x \in \mathbb{N}$ we have $(pr_1 \restriction \mathbb{N})(;x) = \lceil \frac{x}{2} \rceil$ which is polynomial time computable. $\theta_1 \restriction \mathbb{N}$ is the identity function. For all $x \in \mathbb{N}$ we have $e(;x) = o(;x) = 0$. Safe composition sustains polynomial time computability. Assume a function $f \in \mathcal{W}$ defined by linearization from g and h. At integer values f reduces either to h or g, hence by the induction hypothesis $f \restriction \mathbb{N}$ is polynomial time computable. Finally, assume a function $f \in \mathcal{W}$ defined by safe integration from g, h_0, and h_1. Let $\hat{g} = g \restriction \mathbb{N}$, $\hat{h}_0 = h_0 \restriction \mathbb{N}$, and $\hat{h}_1 = h_1 \restriction \mathbb{N}$. Then by the induction hypothesis we have $\hat{g}, \hat{h}_0, \hat{h}_1 \in B$. Define $\hat{f} \in B$ by safe recursion from $\hat{g}, \hat{h}_0, \hat{h}_1$. We claim that $f \restriction \mathbb{N} = \hat{f}$; proof is by strong induction on the integration variable and is similar to that given in the first part. This completes the proof of the theorem.

Efficient computability of the functions in $\mathcal{W}$ is indicated by the following theorem.

Theorem 6. $\mathcal{W} \subseteq P_{\mathbb{R}}$

Proof. Proof is by induction on the construction of functions in $\mathcal{W}$. It is easy to see that the basic functions $0, U, s_i, pr_i$, and θ_1 are all polynomial time computable. The constant π is polynomial time computable. The trigonometric sine and cosine functions are computable in polynomial time using, for example, Taylor series expansion as an approximation. Hence, the parity functions e and o are polynomial time computable. Composition preserves polynomial time computability. Given polynomial time computable functions g and h, then clearly their linearization is polynomial time computable. Assume a function $f \in \mathcal{W}$ that is defined by safe integration from g, h_0, h_1 where these latter functions are polynomial time computable. Then from Theorem 5 we have $f \upharpoonright \mathbb{N}$ is polynomial time computable. As can be seen from the proof of Proposition 3 the function f is piecewise trigonometric with breakpoints at $\mathbb{N}$, hence it is also polynomial time computable at non-integer points.

6 Conclusion

The computational complexity of real functions over non-compact domains is accounted for by two parameters: the *precision* of the desired output and the *location* of the input either with respect to the integer line or to the limit points of a bounded domain. When considering polynomial time computation it is apparent that there is a conceptual gap between applying this notion to the reals and to the rationals. This is due to two main reasons: (1) though smoothness is necessary for efficient real computability, it does not play any role in rational computation and (2) real computation is approximate whereas rational computation is exact. The next move is to study this transition phenomenon from the perspective of computability and other complexity-theoretic classes. We proposed a function algebra of real functions based on the Bellantoni-Cook class. This algebra is able to capture this latter class, though it is still weaker than polynomial time real computation. Still work needs to be done with strengthening this class in order to give a fully algebraic machine-independent characterization of polynomial time real computability.

References

1. S. Bellantoni and S. Cook. A New Recursion-Theoretic Characterization of the Polytime Functions. *Computational Complexity*, 2:97–110, 1992.
2. V. Blondel and J. Tsitsiklis. A Survey of Computational Complexity Results in Systems and Control. *Automatica*, 36(9):1249–1274, 2000.
3. L. Blum, M. Shub, and S. Smale. On a Theory of Computation Over the Real Numbers; NP Completeness, Recursive Functions and Universal Machines. *Bulletin of the American Mathematical Society*, 21(1):1–46, 1989.
4. O. Bournez, M. Campagnolo, D. Graça, and E. Hainry. Polynomial Differential Equations Compute All Real Computable Functions on Computable Compact Intervals. *Journal of Complexity*, 23(3):317–335, 2007.

5. O. Bournez and E. Hainry. Recursive Analysis Characterized as a Class of Real Recursive Functions. *Fundamenta Informaticae*, 74(4):409–433, 2006.
6. M. Campagnolo, C. Moore, and J. Costa. An Analog Characterization of the Subrecursive Functions. In *4th Conference on Real Numbers and Computers*, pages 91–109.
7. M. Campagnolo, C. Moore, and J. Costa. Iteration, Inequalities, and Differentiability in Analog Computers. *Journal of Complexity*, 16(4):642–660, 2000.
8. M. Campagnolo, C. Moore, and J. Costa. An Analog Characterization of the Grzegorczyk Hierarchy. *Journal of Complexity*, 18(4):977–1000, 2002.
9. M. Campagnolo and K. Ojakian. Characterizing Computable Analysis with Differential Equations. *Electronic Notes in Theoretical Computer Science*, 221:23–35, 2008.
10. A. Cobham. The Intrinsic Computational Difficulty of Functions. In Y. Bar-Hillel, editor, *Proc. of the 1964 International Congress for Logic, Methodology, and the Philosophy of Science*, pages 24–30, Amsterdam, 1964.
11. J. Gakwaya. A Survey of the Grzegorczyk Hierarchy and its Extension through the BSS Model of Computability. Technical report, Royal Holloway, University of London, 1997. NeuroCOLT Technical Report Series.
12. W. Gomaa. A Survey of Recursive Analysis and Moore's Notion of Real Computation. *Submitted*.
13. W. Gomaa. Characterizing Polynomial Time Computability of Rational and Real Functions. *To appear in EPTCS*.
14. A. Grzegorczyk. Computable functionals. *Fundamenta Mathematicae*, 42:168–202, 1955.
15. A. Kawamura. Differential Recursion. *ACM Transactions on Computational Logic*, 10(3):1–20, 2009.
16. K. Ko. *Complexity Theory of Real Functions*. Birkhäuser, 1991.
17. D. Lacombe. Extension de la Notion de Fonction Récursive aux Fonctions d'une ou Plusieurs Variables Réelles III. *Comptes Rendus de l'Académie des sciences Paris*, 241:151–153, 1955.
18. B. Loff, J. Costa, and J. Mycka. Computability on Reals, Infinite Limits and Differential Equations. *Applied Mathematics and Computation*, 191(2):353–371, 2007.
19. K. Meer and C. Michaux. A Survey on Real Structural Complexity Theory. *Bulletin of the Belgian Mathematical Society*, 4(1):113–148, 1997.
20. C. Moore. Recursion Theory on the Reals and Continuous-Time Computation. *Theoretical Computer Science*, 162(1):23–44, 1996.
21. Olivier Bournez and Manuel L. Campagnolo. A Survey on Continuous Time Computations. In S.B. Cooper and B. Löwe and A. Sorbi, editor, *New Computational Paradigms. Changing Conceptions of What is Computable*, pages 383–423. Springer-Verlag, New York, 2008.
22. P. Orponen. A Survey of Continous-Time Computation Theory. In *Advances in Algorithms, Languages, and Complexity*, pages 209–224, 1997.
23. A. Turing. On Computable Numbers, With an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936. (correction ibid. 43, pp 544-546, 1937).
24. K. Weihrauch. *Computable Analysis: An Introduction.* Springer, 1 edition, 2000.

Non-deterministic Boolean Proof Nets

Virgile Mogbil

LIPN – UMR7030, CNRS – Université Paris 13, France
virgile.mogbil@lipn.univ-paris13.fr [⋆]

Abstract. We introduce Non-deterministic Boolean proof nets to study the correspondence with Boolean circuits, a parallel model of computation. We extend the cut elimination of Non-deterministic Multiplicative Linear logic to a parallel procedure in proof nets. With the restriction of proof nets to Boolean types, we prove that the cut-elimination procedure corresponds to Non-deterministic Boolean circuit evaluation and reciprocally. We obtain implicit characterization of the complexity classes NP and NC (the efficiently parallelizable functions).

1 Introduction

The *proof nets* [Gir87,DR89] of the Linear logic (LL) are a parallel syntax for logical proofs without all the bureaucracy of sequent calculus. Their study is also motivated by the well known Curry-Howard isomorphism: there is a correspondence between proofs and programs which associates cut-elimination in proofs and execution in functional programs. Proof nets were used in subsystems of LL to give Curry-Howard characterizations of complexity classes. Usually this is done in LL by reducing the deductive power of the exponentials connectives, also known as modalities, controlling duplication in cut-elimination process. The most known restrictions characterize P, the class of decision problems which can be solved in polynomial time. E.g. it is the case of the Intuitionistic Light Affine Logic (ILAL). By expressing non-determinism by an explicit rule to choose between proofs of the same formula, the Non-deterministic extension of ILAL characterizes quite naturally *NP* [Mau03]. This *sum rule* is a logical counterpart to non-deterministic choice in process calculi. With proof nets, other characterizations were given by correspondence with models of parallel computation like Boolean circuits as we shall see later. In this paper we make use of the sum rule to extend these proof nets to a non-deterministic framework.

Boolean circuits are a standard models of parallel computation [Coo85,BS90] [Vol99]. Several important complexity classes are defined in terms of Boolean circuits. E.g. *NC* can be thought of as the problems that can be efficiently solved on a parallel computer just as the class P can be thought of as the tractable problems. Because a Boolean circuit has a fixed input size, an infinite family of Boolean circuits is needed to do computations on arbitrary inputs. With a *uniformity* property, a family can be however regarded as an implementation

⋆ Work partially supported by the french project Complice (ANR-08-BLANC-0211-01)

of an algorithm. The Boolean circuit depth is the time on a parallel computer where the size is the number of processors. For instance *NC* is the set of Boolean functions computable by uniform Boolean circuits of polynomial size and polylogarithmic depth.

By restricting proved formulae and with a logical depth notion, there is a proofs-as-programs correspondence between proof nets and *NC* such that both size and depth are preserved [Ter04]. In this setting, the proof nets allow among others to simulate gates by higher order. Here we consider a non-deterministic extension of proof nets, to give a proof-as-programs correspondence with $NNC(k(n))$, the class defined from *NC* Boolean circuits with $O(k(n))$ non-deterministic variables. In particular $NC = NNC(log\ n)$ and $NP = NNC(poly)$. So the Curry-Howard isomorphism for parallel model of computation gives us new tools to study theoretical implicit complexity.

In section 2 we present MLL_u, the multiplicative LL with arbitrary arity This is the smallest fragment of LL that encodes the unbounded fan-in Boolean circuits [Ter04]. We recall the reduction steps of the cut-elimination. We give its non-deterministic extension $nMLL_u$ and proof nets for it as in [Mau03]. We recall several size and depth notions used in the proofs, all are natural graph theoretic notions. In section 3 we mostly analyze and define a new cut elimination from a parallel point of view. We prove the central theorems which allow us to establish the results on the complexity classes. In section 4 we recall Boolean circuit definitions. They include uniformity of both proof net families and circuit families as well as hierarchy of *NC* and $NNC()$. We introduce the Boolean proof nets of $nMLL_u$, i.e with sum-boxes, which generalize the proof nets of MLL_u. In section 5 we apply the previous theorems to our Boolean proof nets with sum-boxes and we establish a proofs-as-programs correspondence with $NNC()$ Boolean circuits. They are translation and simulation theorems that preserve both size and depth of the Boolean proof nets and Boolean circuits. Finally in section 6 we summarize the obtained results via $UBPN()$, a hierarchy of proof net complexity classes defined analogously to the $NNC()$ hierarchy. The classes $UBPN(poly)$, $UBPN(polylog)$ and $UBPN(1)$ of uniform Boolean proof net families with respectively $n^{O(1)}$, $log^{O(1)}\ n$ and $O(1)$ sum-boxes are respectively equal to *NP*, $NNC(polylog)$ and *NC*.

2 Non-deterministic Linear logic

2.1 Formulae, Sequent calculus and cut-elimination

We write $\overrightarrow{A}$ (respectively $\overleftarrow{A}$) for an ordered sequence of formulae $A_1, \ldots, A_n$ (resp. $A_n, \ldots, A_1$).

The *formulae* of MLL_u and $nMLL_u$ are built on literals by conjunction $\otimes^n(\overrightarrow{A})$ and disjunction $\invamp^n(\overleftarrow{A})$ of Multiplicative Linear logic but with unbounded arities $n \geqslant 1$. The only difference with binary connectives is that this gives us depth-efficient proofs [DR89]. The *negation* of a non-literal formula is defined by De Morgan's duality: $(\otimes^n(\overrightarrow{A}))^\perp = \invamp^n(\overleftarrow{A}^\perp)$ and $(\invamp^n(\overrightarrow{A}))^\perp = \otimes^n(\overleftarrow{A}^\perp)$ where the negation applies to sequences of formulae as expected.

The *sequents* of $nMLL_u$ are of the form $\vdash \Gamma$, where Γ is a multiset of formulae. The rules of the $nMLL_u$ sequent calculus are described in Fig.1. As in Linear logic sequent calculus, MLL_u and $nMLL_u$ admit a cut-elimination theorem (Hauptsatz) and implicit exchange. The cut-elimination steps of $nMLL_u$

$$\text{(a)} \quad \frac{\vdash \Gamma, A \quad \vdash \Delta, A^\perp}{\vdash \Gamma, \Delta}\,cut \qquad \frac{}{\vdash A, A^\perp}\,ax \qquad\Bigg|\qquad \text{(b)} \quad \frac{\vdash \Gamma \quad \cdots \quad \vdash \Gamma}{\vdash \Gamma}\,sum$$

$$\frac{\vdash \Gamma_1, A_1 \quad \cdots \quad \vdash \Gamma_n, A_n}{\vdash \Gamma_1, \ldots, \Gamma_n, \otimes^n(\overrightarrow{A})}\,\otimes^n \quad \frac{\vdash \Gamma, A_n, \ldots, A_1}{\vdash \Gamma, \parr^n(\overleftarrow{A})}\,\parr^n$$

Fig. 1. Sequent calculus rules: (a) MLL_u and (a+b) $nMLL_u$

sequent calculus are n-ary versions of the Linear logic proofs rewriting: an axiom cut (i.e. a cut rule with an axiom premise) rewrites without the axiom rule, and the multiplicative cut (i.e. a cut rule between multiplicative formulae) rewrites in sequence of cuts between premisses. In Fig.2 we give the sum-rule cut-elimination step as in [Mau03].

$$\frac{\dfrac{\vdash \Gamma, A \quad \cdots \quad \vdash \Gamma, A}{\vdash \Gamma, A}\,sum \qquad \vdash \Delta, A^\perp}{\vdash \Gamma, \Delta}\,cut \longrightarrow \frac{\dfrac{\vdash \Gamma, A \quad \vdash \Delta, A^\perp}{\vdash \Gamma, \Delta}\,cut \quad \cdots \quad \dfrac{\vdash \Gamma, A \quad \vdash \Delta, A^\perp}{\vdash \Gamma, \Delta}\,cut}{\vdash \Gamma, \Delta}\,sum$$

Fig. 2. $nMLL_u$ sum-rule cut-elimination step

2.2 Proof nets and reduction rules

We suppose the reader familiar with proof nets and more specifically with the reduction rules corresponding to the cut-elimination steps [Gir96].

A *proof net* [Gir87,DR89] is a graph that just keeps the structure of sequent calculus proofs without some computational irrelevant part. It is a set of connected links that we build by inference from the sequent calculus rules. There

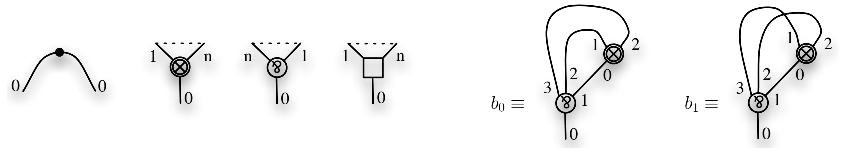


Fig. 3. ax-link, $\otimes$-link, $\parr$-link and $\square$-link ; small proof nets

are several *sorts* of links: ax-link, $\otimes$-link and $\parr$-link corresponding respectively to MLL_u rules. Every link has several ports numbered by convention as in Fig.3. This numbering is omitted when there is no ambiguity. The *conclusion port*, also called conclusion of the link, is the port numbered 0. We use a wire between two conclusion ports rather than a cut-link that corresponds to sequent calculus cut-rule. Now we focus us on the *nMLL*$_u$ proof nets which extend those of MLL_u. As we only consider proof nets inferred from sequent calculus, we can inductively built them in a equivalent way from the *ax*-links with the constructors of Fig. 4. The sequent calculus sum-rule corresponds to a box (Fig.4), called *sum-box*, enclosing proof nets, called *sub-nets*, such that we associate one $\square$-link for each shared conclusion.

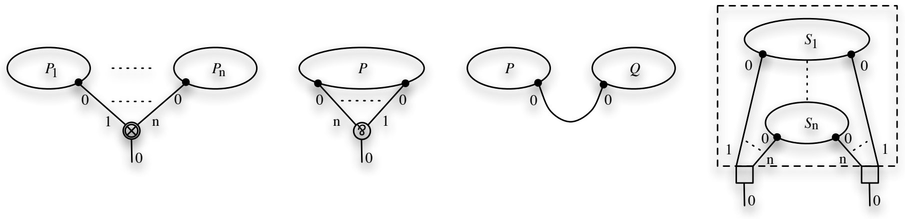


Fig. 4. Proof net constructors: $\otimes$-link, $\parr$-link, cut and sum-box

Definition 1. *A* proof net *is (i) a finite set L of links, (ii) a function $\sigma : L \to \{\bullet, \square\} \cup \{\otimes^n, \parr^n\}_{n\geqslant 1}$ giving the sort of each link, (iii) a symmetric binary relation over $L \times \mathbb{N}$ for each wire between two ports of distinct links, and (iv) a function $\tau : L \to \mathbb{N}$ which associates to each link a unique number corresponding to the sum-box that encloses it.*

A summand *of a proof net is a proof net obtained by erasing all but one sub-net in every sum-box.*

In section 4 we give a Logspace description which extends this one. We call the *conclusions* of a proof net the set of links whose conclusion ports are unrelated with other link ports. We write proof nets either with graphs or just with the link names. For instance the last proof net in Fig.4 is $sum(S_1, \ldots, S_n)$.

Remark that in every summand of a proof net the $\square$-links correspond to superfluous sum-rules with only one premise. So every summand of a proof net of *nMLL*$_u$ easily induces a proof net of MLL_u.

Definition 2. *The reduction rules*[1] *for MLL_u proof nets, called respectively ax-*reduction *and m-*reduction*, are defined by:*

$$cut(ax(A, A^{\perp}), A) \ \rightarrow_{ax} \ A \text{ and } cut(\otimes^n(\overrightarrow{A}), \parr^n(\overleftarrow{A}^{\perp})) \ \rightarrow_m \ \{cut(A_i, A_i^{\perp})\}_{1\leqslant i\leqslant n}$$

*For nMLL$_u$ proof nets there are moreover the reduction rules, called respectively merge-*reduction *and down-*reduction*, defined for an arbitrary context C by:*

$$sum(sum(\overrightarrow{A}), \overrightarrow{B}) \rightarrow_{merge} sum(\overrightarrow{A}, \overrightarrow{B})$$

$$sum(C[sum(\overrightarrow{A})], \overrightarrow{B}) \ \rightarrow_{down} \ sum(sum(C[\overrightarrow{A}]), \overrightarrow{B})$$

[1] Up to the order when they are not drawn

We define various notions of depth in a natural way: the box-depth associated to sum-boxes, the depth of formulae and the logical depth associated to cuts.

Definition 3. *The* box-depth $b(l)$ *of a link* l *is the number of sum-boxes that enclose it. The box-depth of a sum-box is the box-depth of its* $\Box$*-link links. The box-depth* $b(P)$ *of a proof net* P *is the maximal box-depth of its links.*

The depth $d(A)$ *of a formula* A *in a sequent calculus derivation* π *is the length of the longest sequence of rules in* π *since axioms until the introduction of the formula* A*. The depth* $d(\pi)$ *of a* $nMLL_u$ *sequent calculus derivation* π *is the maximal depth of its cut formulas.*

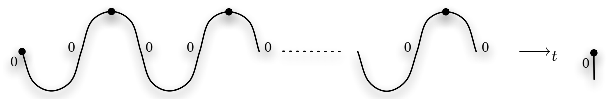
The logical depth $\underline{c}(P)$ *of a proof net* P *is* $\underline{c}(P) = min\{d(\pi) \mid P$ *is inferred by the sequent calculus derivation* $\pi\}$*. We write* $c(P)$ *the logical depth without counting the sum-rules.*

Remark that by definition all $\Box$-links of the same sum-box are at same box-depth. When a proof net is induced by a sequent calculus proof, he keeps traces of sequentialization by the sum-boxes. This corresponds to a stratification by the box-depth that we use in our reduction strategy.

The *size* $|P|$ of a proof net P is the number of links in P such that we count boxes but not the $\Box$-links.

3 Parallel reductions

Like the cut-elimination of sequent calculus the reduction of proof net is terminating. Even if the reduction of proof nets has the Church-Rosser property, reductions cannot be done in parallel as there is critical pairs. A critical pair arises when redexes of reduction rules overlap. In order to have efficient proof net reductions we consider all possible critical pairs. E.g. a cut between two distinct axioms has two overlapping redexes depending on if we consider one or the other axioms. This is solved for MLL_u in [Ter04] by introducing a *tightening reduction*: one reduces each maximal alternating chain of cuts and axioms in only one global step leading to what is expected by sequential reduction steps. Here is an example where the maximal chain starts with a cut and finishes with an axiom:



Remark that by maximality, no critical pairs can be tightening redexes. So such reductions can be done in parallel. We denote t-reduction a tightening reduction and we write it $\rightarrow_t$. We write $\Rightarrow_t$ for the simultaneous application of all the $\rightarrow_t$ reductions that can be fired in a given proof net. As $\rightarrow_m$ redexes are never critical pairs we also write $\Rightarrow_m$ for the simultaneous $\rightarrow_m$ reductions. In the rest of the paper we write $\Rightarrow^k$ for k parallel reduction steps ($\Rightarrow_t$ or $\Rightarrow_m$) in MLL_u.

In this section we introduce new reduction rules to answer the parallelization question for $nMLL_u$ such that reduction is confluent. We will give a bound on the number of reductions to normalize a proof net (theorem 1).

3.1 Parallel reductions of merged sum-boxes

The merge rules applied to distinct sum-boxes immediately within a same sum-box are not really a conflict case even if it is a critical pair. We could merge them in parallel if the sum-box itself would not be merged at the same time. Because the merge-rule is confluent one can consider a global rewriting rule by merging sum-boxes as expected. This corresponds to a sort of innermost sequential strategy. We define a new reduction $\rightarrow_M$ whose redex is a maximal tree of successively nested sum-boxes. So the root of the tree of sum-boxes is an outermost sum-box that cannot be merged further. The $\rightarrow_M$ reduction merges this tree in the root sum-box as it is mimicked in Fig.5. We do not draw neither wires between the $\Box$-links nor the sub-nets which are not sum-boxes. By maximality this reduction has no critical pairs, so we can do simultaneously $\rightarrow_M$ reductions.

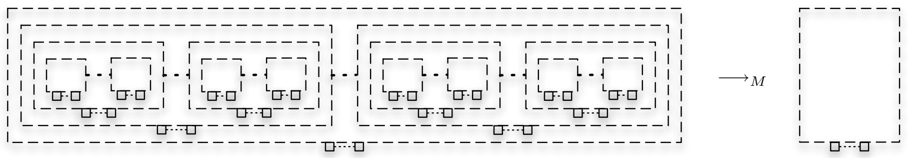


Fig. 5. $\rightarrow_M$ is the global reduction by merging

3.2 Parallel down-reductions

According to the context C, the pattern $sum(C[sum(\overrightarrow{A})], \overrightarrow{B})$ can reveal a critical pair for the down rule. E.g. within a sum-box $cut(sum(A_1, A_2), sum(B_1, B_2))$ reduces to $sum(sum(\{cut(A_i, B_j)\}_{1 \leqslant i,j \leqslant 2}))$ in four $\rightarrow_{down}$ steps. The same holds for a more general context C but remains convergent. To simplify the notations, we use the mathematical notion of section of indexed family of elements.

Definition 4. *Let B be a sum-box and let S be a sub-net of B, i.e. $B = sum(\overrightarrow{X}, S)$. Let $\{B_i\}_{i \in I}$ be the family of nested sum-boxes of B in S, i.e. $B_i \subset S$ and $b(B_i) = b(B) + 1$ for all $i \in I$. Let C be the context of the B_i with relation to S, i.e. $S = C[\{B_i\}_{i \in I}]$. We write $\overline{B_i}$ the family of sub-nets of B_i, i.e. $B_i = sum(\overline{B_i})$. The* section *$A \triangleleft F$ of a family $F = \{\overline{B_i}\}_{i \in I}$ is a family $A = \{a_k\}_{k \in I}$ such that for all $k \in I$ we have $a_k \in \overline{B_i}$. We define $\rightarrow_B$ the reduction of a sub-net of a sum-box B by:*

$$sum(\overrightarrow{X}, C[\{sum(\overline{B_i})\}_{i \in I}]) \quad \rightarrow_B \quad sum(\overrightarrow{X}, \{C[A] \mid A \triangleleft \{\overline{B_i}\}_{i \in I}\})$$

We abusively use a set to denote the reduct but the corresponding family is easily obtained from the implicit order associated to sections.

The intuition is to reduce in one (parallelizable) reduction step the sum-boxes B_i of one sub-net S, merging only affected sum-boxes: the result is a set of sub-nets which are the combination of all the contents of the B_i and the context outside the B_i. So the sum-boxes B_i go down whereas contexts go up. Remark that if the family of B_i is such that each B_i contains exactly one sub-net then by definition the context C applies on this set of sub-nets.

We give an example: let $B_1 = sum(a_1, a_2)$ and $B_2 = sum(b_1, b_2)$, let C be a context without other sum-boxes than B_1 and B_2. We have:
$sum(\overrightarrow{X}, C[B_1, B_2]) \to_B sum(\overrightarrow{X}, C[a_1, b_1], C[a_1, b_2], C[a_2, b_1], C[a_2, b_2])$.

Remark that the affected sum-boxes were merged by our reduction, but just to have a more readable definition. This reduction rule is a generalization of the case of context composed only with cut sum-boxes: in such case we need to reduct at fixed box-depth after all other reductions at this box-depth. But without this generalization we are not able to decrease at the same time the logical depth.

We redefine now $\to_B$ as the previous reduction extended to all sub-nets of the same sum-box B. This reduction is well defined as sub-nets are disjoints. We write $\Rightarrow_{Dx}$ such reductions applied in parallel to every sum-box at the same box-depth $x - 1$. By this way the sum-boxes at box-depth x go down. We do this in order to have no conflicts. For a given proof net P, we write ${\Rightarrow_D}^{b(P)}$ the reduction sequence $\Rightarrow_{Db(P)} \cdots \Rightarrow_{D1}$. So we have the following remarkable properties:

Lemma 1. *Let P be a proof net such that $b(P) > 0$.*
If $sum(P) \Rightarrow_{Db(P)} sum(P')$ then $b(P') = b(P) - 1$ and $c(P') = c(P)$.

Proof. Let S be a sub-net of a sum-box B of box-depth $b(P) - 1$ in $sum(P)$. We have $b(S) = 1$. By definition each link in the reduct of S by $\to_B$ has the box-depth $b(P) - 1$. It is the same for all sub-nets of B by $\to_B$ reduction. By definition the $\Rightarrow_{Db(P)}$ reduction applies to all sum-boxes of box-depth $b(P) - 1$ in $sum(P)$. So the conclusion follows. □

The above Lemma implies the following:

Lemma 2. *Let P be a proof net such that $b(P) > 0$.*
If $sum(P) \Rightarrow_{Db(P)} \cdots \Rightarrow_{D1} \Rightarrow_M sum(P')$ then we have:

i) $c(P') = c(P)$ and $b(P') = 0$ i.e. P' is sum-box free.
ii) For all choice of summand s_i for $i \in I$, $s_i(sum(P')) \Rightarrow^{c(P)} P_i$ cut-free.

Theorem 1. *There is a sequence of $O(c(P) + b(P))$ parallel reductions which reduces a $nMLL_u$ proof net P in a cut-free one.*

Proof. Let P' defined from P as in lemma 2. Because all the summands s_i for $i \in I$ are pairwise disjoint we have $\cup_{i \in I} s_i(sum(P')) \Rightarrow^{c(P)} \cup_{i \in I} P_i$ cut-free. So $sum(P) \Rightarrow_D^{b(P)} \Rightarrow_M^1 sum(P') \Rightarrow^{c(P)} sum(\{P_i\}_{i \in I})$ gives a cut-free proof net. □

Remark that $\underline{c}(P) = O(c(P) + b(P))$.

4 Boolean circuits and Boolean proof nets

In this section we recall the definitions and some properties of Boolean circuits and Boolean proof nets. We give also certain relationships between complexity classes associated to Boolean circuits [Coo85,All89,Wol94,Par89]. The novelties concern only the Boolean proof nets of $nMLL_u$ that extend those of MLL_u.

4.1 Boolean circuits

A *basis* is a finite set of sequences of Boolean functions. The standard basis are $\mathcal{B}_0 = \{\neg, \wedge, \vee\}$ and $\mathcal{B}_1 = \{\neg, (\wedge^n)_{n\in\mathbb{N}}, (\vee^n)_{n\in\mathbb{N}}\}$. The circuits over basis with an infinite sequence of Boolean functions (resp. without) are called *unbounded fan-in* (resp. *bounded fan-in*) circuits. We extend the basis with a $stCONN_2$ gate to test the strong connectivity of an edge set given in input. As in [Ter04] we will use this gate to simulate the tightening reduction of proof nets.

Definition 5. *A* deterministic Boolean circuit *with n inputs over a basis $\mathcal{B}$ is a directed acyclic graph with $n+1$ sources or inputs (vertices with no incoming edges) and one sink or output (a vertex with no out-going edges). Sources are labeled by literals from $\{x_1, \ldots, x_n\} \cup \{1\}$ and nodes of in-degree k are labeled by one of the k-ary Boolean functions of $\mathcal{B}$. Non-inputs nodes are called* gates*, and in-degree and out-degree are called* fan-in *and* fan-out *respectively. Let F_n be the set of all Boolean functions $f : \{0,1\}^n \to \{0,1\}$ for some $n \in \mathbb{N}$. A deterministic circuit* computes *a function in F_n (or* accepts *a set $X \subseteq \{0,1\}^n$) in a natural way.*

A non-deterministic *Boolean circuit C with n inputs over a basis $\mathcal{B}$ with k non-deterministic variables is a circuit with $n+k+1$ sources labeled by $\{x_1, \ldots, x_n\} \cup \{y_1, \ldots, y_k\} \cup \{1\}$ s.t. it computes a function $f \in F_n$ as follows: for $x \in \{0,1\}^n$, $f(x) = 1$ iff $\exists y \in \{0,1\}^k$ a witness s.t. $C(x, y)$ evaluates to 1.*

A family *of circuits $C = (C_n)_{n\in\mathbb{N}}$ computes a function $f : \{0,1\}^* \to \{0,1\}$ (or accepts a set $X \in \{0,1\}^*$) if for every $n \in \mathbb{N}$ the circuit C_n computes the restriction of f to F_n.*

The size *of a circuit is the number of gates and the* depth *is the length of a longest directed path.*

4.2 Boolean proof nets

Boolean values are represented with the type $\mathbf{B} = \parr^3(\alpha^\perp, \alpha^\perp, \otimes^2(\alpha, \alpha))$. The non-deterministic Boolean values are represented with the same type ! Let us consider the following proof nets of type $\mathbf{B}$: the non-deterministic Boolean represented by $b_2 \equiv sum(b_0, b_1)$ where b_0 and b_1 are the two cut-free proof nets of MLL_u respectively called *false* and *true* given in Fig.3.

Definition 6 ([Ter04]). *A* Boolean proof net *with n inputs $\overrightarrow{p} = p_1, \ldots, p_n$, one output and $k(n)$ sum-boxes is a proof net $P(\overrightarrow{p})$ of $nMLL_u$ of type:*

$$\vdash p_1 : \mathbf{B}^\perp[A_1], \ldots, p_n : \mathbf{B}^\perp[A_n], q : \otimes^{m+1}(\mathbf{B}, \overrightarrow{C})$$

with $k(n)$ a function of n, and some $\overrightarrow{A} \equiv A_1, \ldots, A_n$ and $\overrightarrow{C} \equiv C_1, \ldots, C_m$ where we denote $\mathbf{B}[A]$ the formula $\mathbf{B}$ where all occurrences of α are substituted by A. Given $\overrightarrow{x} \equiv b_{i_1}, \ldots, b_{i_n}$ we write $P(\overrightarrow{x})$ the proof net where we cut every b_i with p_i.

Without loss of generality we can always set $sum(P)$ for P: if a Boolean proof net P is without sum-boxes then $sum(P)$ is a sum-box with only one summand in MLL_u. This generalizes uniform MLL_u Boolean proof nets. We often omit this initial sum-box for the sake of simplicity.

Following the definition, for a given $\overrightarrow{x} \equiv b_{i_1}, \ldots, b_{i_n}$, a Boolean proof net $sum(P(\overrightarrow{x}))$ is of type $\otimes^{m+1}(\mathbf{B}, \overrightarrow{C})$ and reduces in a unique cut-free proof net of the same type (e.g. by the reduction sequence of the previous section). We say that $sum(P(\overrightarrow{x}))$ evaluates to 1 iff one of its summands is b_1 with some *garbage* $\overrightarrow{C}$. There is an asymmetry between 1 and 0 as for non-deterministic Turing machines.

Definition 7 ([Ter04]). *A Boolean proof net $P(\overrightarrow{p})$ with n inputs* computes *a function $f : \{0,1\}^n \to \{0,1\}$ (or* accepts *a set $X \subseteq \{0,1\}^n$) if $P(\overrightarrow{x})$ evaluates to $b_{f(x)}$ for every $\overrightarrow{x} \equiv b_{i_1}, \ldots, b_{i_n}$ corresponding to $x \equiv i_1 \cdots i_n \in \{0,1\}^n$.*

The Boolean proof net encoding of K. Terui [Ter04] of functions as negation, conditional, disjunction, composition, duplication, and so on, is trivially valid in the extended setting of $nMLL_u$. We give some of them in the Appendix.

4.3 Uniformity and complexity classes

The given notions are used for Boolean circuit families and for $nMLL_u$ proof net families. Both are denoted by $F = (F_n)_{n\in\mathbb{N}}$. From an algorithmic point of view the uniformity is an important issue because only a uniform family can be regarded as an implementation of an algorithm.

Definition 8 ([Ruz81]). *A family $F = (F_n)_{n\in\mathbb{N}}$ is called L-*uniform *(respectively P-*uniform*) if there is a function which computes a description of F_n from 1^n (unary numeral) in space $O(log\ |F_n|)$) (respectively in time $|F_n|^{O(1)}$).*

A description means all informations about elements like sort, predecessors, ... Usually the description is also chosen according to the uniformity notion used. Because we work with NP the P-uniformity is sufficient [All89]. Nevertheless if we want to study a property in NC then we use the L-uniformity with the following notion of description where links and sorts are identified by binary numbers. Let W be the set of binary words and let $\overline{x}$ be the binary representation of the integer x.

Definition 9 ([Ruz81]). *The* direct connection language *$L_{DC}(C)$ of a Boolean circuit family $C = (C_n)_{n\in\mathbb{N}}$ over basis $\mathcal{B}$ is the set of tuples $\langle \overline{y}, \overline{g}, \overline{w}, \overline{b} \rangle \in W^4$, where for $y = n$ we have g is a gate in C_n labeled by the function b from $\mathcal{B}$ if $\overline{w} = \varepsilon$ else b is the w^{th} predecessor gate of g. In case of input, b is not a function but a sort (deterministic input or non-deterministic variable).*

This definition adapted to MLL_u proof nets [MR07] is here extended analogously to $nMLL_u$ proof nets:

Definition 10. *The* direct connection language *$L_{DC}(P)$ of a $nMLL_u$ proof net family $P = (P_n)_{n\in\mathbb{N}}$ is the set of tuple $\langle \overline{y}, \overline{l}, \overline{w}, \overline{b}, \overline{s} \rangle \in W^5$ where for $y = n$ we have l is a link in P_n of sort b if $\overline{w} = \varepsilon$ else the w^{th} premise of l is the link b. The link l is also enclosed by the s^{th} sum-box.*

Remark that the length of all identifiers of such family P is bounded by $log\ |P|$.

We finish this section recalling the complexity classes defined with uniform families of Boolean circuits ([Coo85,All89,Vol99] and [Wol94,Par89] for non-deterministic Boolean circuits). All dimensions are defined w.r.t the length of the input, which we write everywhere n. We use abusively the words *poly* for $n^{O(1)}$ and *polylog* for $log^{O(1)}\ (n)$.

The *classes* NC^i and AC^i for $i \geqslant 0$ are the functions computable by uniform families of polynomial size, $O(log^i n)$ depth circuits over $\mathcal{B}_0$ and $\mathcal{B}_1$ respectively. We add the suffix ($stCONN_2$) to classes if we extend basis with a $stCONN_2$ gate. The *class* $NNC^i(k(n))$ (resp. $NAC^i(k(n))$) are the functions computable by NC^i (resp. AC^i) circuit families with $O(k(n))$ non-deterministic variables. We write NC, $NNC(k(n))$, AC and $NAC(k(n))$ the respective unions over the exponents of the depth. Relationships between complexity classes follow: $\forall i \in \mathbb{N}$ and $\forall j \in \mathbb{N}^*$

$$AC^0 \subsetneq NC^1 \subseteq L \subseteq NL \subseteq AC^1 \subseteq NC^2 \subseteq \ldots \subseteq AC = NC \subseteq P$$

$$AC^i \subseteq AC^i(stCONN_2) \subseteq AC^{i+1}$$

$$NNC^j(log\ n) = NC^j, \text{ and then } NNC(log\ n) = NC$$

$$NNC^j(poly) = NNC(poly) = NAC^i(poly) = NAC(poly) = NP$$

5 Translation and simulation

5.1 Logspace translation

Let $C = (C_n)_{n\in\mathbb{N}}$ be a uniform Boolean circuit family over the basis $\mathcal{B}_1(stCONN_2)$ with non-deterministic variables. The intermediate proof nets we build are called modules.

First of all without distinguishing the inputs, we associate a uniform Boolean proof net family $P = (P_n)_{n\in\mathbb{N}}$ to C using uniformity. After what we consider non-deterministic variables. Indeed the uniformity function of Boolean circuits builds the uniformity function of Boolean proof nets in the same way as in [MR07]. The main idea is already given in [Ter04]. Starting from $L_{DC}(C)$:

- For each n-fan-in gate labeled $f(n)$ read in $L_{DC}(C_n)$ we give a polysize module computing $f(n)$. If the gate is a non-deterministic variable we cut the corresponding input with the proof net $b_2 \equiv sum(b_0, b_1)$,
- For each n-fan-out gate read in $L_{DC}(C_n)$ we make a polysize duplication,
- For each *edge* read in $L_{DC}(C_n)$ we joint modules and duplications.

Just parsing the $L_{DC}(C_n)$ for $i = 0$ to $|C_n|$ we detail a Logspace translation into $L_{DC}(P_n)$: everything is identified with a binary number

1. For each $\langle n, i, \varepsilon, b\rangle$ we build the module associated to the function b of the basis of the family (or to the sort b in case of inputs). It is a subset of $L_{DC}(P_n)$ where relationships between links and sorts are given.

2. If there are several $\langle n, i, k, j\rangle$ (i.e. j is the k-th predecessor of i) for fixed n and j then the fan-out of j is multiple. We build the corresponding duplication. It is again a subset of $L_{DC}(P_n)$.
3. For each $\langle n, i, k, j\rangle$ (i.e. j is the k-th predecessor of i) we build $\langle n, a, b, c, 0\rangle$ (i.e. an edge from $(a, 0)$ to (b, c)) where a is the link associated to the output of the module (step 1) corresponding to j and b is the link associated to the c-th input of the module corresponding to i, modulo added duplications.

The novelty is the translation of non-deterministic variables into cuts with $b_2 \equiv sum(b_0, b_1)$. Only in this case the last bit of the description is not null.

Theorem 2. *For every uniform family C of unbounded fan-in Boolean circuit of size s and depth c over the basis $\mathcal{B}_1(stCONN_2)$ and with $k(n)$ non-deterministic variables, there is a uniform family of Boolean proof nets of $nMLL_u$ of size $s^{O(1)}$ and logical depth $O(c)$ and with $k(n)$ sum-boxes, which accepts the same set as C does.*

Proof. Let $C_n \in C$ and $P_n \in P$ the Boolean proof net obtained by translation. By translation $b(P_n) = 1$. Every gate is translated by a module of size $O(s^4)$ and constant depth, and only the composition of these modules increases linearly the depth [Ter04]. Let $x \in \{0,1\}^n$ an input of C_n and $\overrightarrow{x}$ corresponding to x as in definition 7. By theorem 1 proof we have: $sum(P_n(\overrightarrow{x}))\Rightarrow^1_{D1} \Rightarrow^1_M sum(\{P_i\}_{i\in I}) \Rightarrow^c sum(\{Q_i\}_{i\in I})$ is an $O(c)$ steps reduction such that $sum(\{Q_i\}_{i\in I})$ is cut free and there is a witness $y \in \{0,1\}^{k(n)}$ such that $C_n(x, y)$ evaluates to 1 if and only if $P_n(\overrightarrow{x})$ evaluates to 1 (i.e. $\exists i \in I$ such that $Q_i = b_1$). $\square$

By translation $c = O(\underline{c})$, so we have the same result for logical depth $O(\underline{c})$.

5.2 Simulation of parallel reductions

Let $P = (P_n)_{n\in\mathbb{N}}$ be a uniform Boolean proof net family of $nMLL_u$ with $k(n)$ sum-boxes. We associate a uniform Boolean circuit family $C = (C_n)_{n\in\mathbb{N}}$ to P in two big steps based on the reduction sequence of theorem 1:

- We both initialize the circuit descriptions and simulate all the parallel down reductions with a polysize and constant depth circuit, using uniformity,
- As in [Ter04] we simulate all the $\Rightarrow$ reductions of all summands using $stCONN_2$ gates for $\Rightarrow_t$ simulation and then we check the result of the last configuration.

In more detail, from the description of a proof net $P_n \in P$ we build Θ_0 an initial set of boolean values representing the proof net to simulate. A *configuration* $\Theta \in Conf(P_n)$ is the set of the following Boolean values: $alive(l)$, $sort(l, s)$, $box(l, i)$ and $edge(l, 0, l', i')$. These values are initialized to 1 respectively iff a link $l \in L$ is in P_n, l is of sort s, l is enclosed by the sum-box numbered i, and the conclusion port of l is in relation with the i'^{th} port of a link $l' \in L$. In other terms our initial configuration Θ_0 is the description itself extended to alive values. Every reduction step simulation is made by a small circuit which modifies a configuration in the other. All of this can be done in Logspace.

Lemma 3. *There is an unbounded fan-in circuit C of size $O(|P_n|^3)$ and constant depth over $\mathcal{B}_1$ with non-deterministic variables, which computes in Logspace $\Theta \in Conf(P')$ from $\Theta_0 \in Conf(P_n)$ whenever $P_n(b_{i_1}, \ldots, b_{i_n}) \Rightarrow_D^{b(P_n)} \Rightarrow_M^1 P'$ from given inputs $i_1, \ldots, i_n$.*

Proof. Like in the translation we parse the configuration of $P_n(b_{i_1}, \ldots, b_{i_n})$ without taking care of sum-boxes to build partially $L_{DC}(C)$ in Logspace. From $Conf(P_n)$, we complete $L_{DC}(C)$ in Logspace and compute $Conf(P')$ as follows: The simulation of one k-ary sum-box t which corresponds to k summands/choices, uses $log(k)$ non-deterministic variables $\{G^t\}$. Let l be a link of box-depth $b(l)$, i.e. l is enclosed in exactly $b(l)$ sum-boxes $\{t_i\}_{i \in I}$. Let k_i be the arity of the sum-box t_i. So the link l simulation depends on $\Sigma_{i \in I} log(k_i)$ non-deterministic gates. We initialize the value of the corresponding edges with the conjunction of the values of these non-deterministic gates $\cup_{i \in I} \{G^{t_i}\}$ like in Fig. 6. Globally this constant depth initialization uses one conjunction gate by edge in $Conf(P_n)$ and one negation gate by non-deterministic gates. □

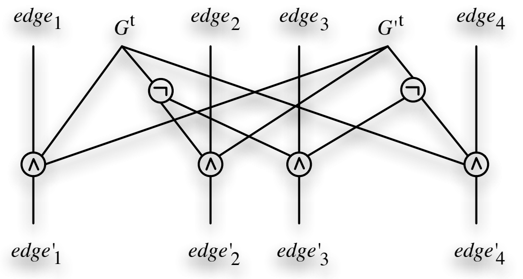


Fig. 6. 2^2-ary sum-box simulation where $edge'_i$ is in the i^{th} summand

Lemma 4. *[Ter04] There is an unbounded fan-in circuit C over $\mathcal{B}_1(stCONN_2)$ of size $O(|P|^3)$ and constant depth such that whenever a configuration $\Theta \in Conf(P)$ is given as input and $P \Rightarrow P'$, C outputs a $\Theta' \in Conf(P')$.*

Theorem 3. *For every uniform family P of Boolean proof nets of $nMLL_u$ of size s and logical depth c, with $k(n)$ sum-boxes of maximal arity k, there is a uniform family of unbounded fan-in Boolean circuit over the basis $\mathcal{B}_1(stCONN_2)$ of size $s^{O(1)}$ and depth $O(c)$ and with $O(k(n).log(k))$ non-deterministic variables, which accepts the same set as P does.*

Proof. By theorem 1, $i_1 \cdots i_n$ is accepted by P if and only if $b_1 \in \{Q_i\}_{i \in I}$ where $P_n(b_{i_1}, \ldots, b_{i_n}) \Rightarrow_D^{b(P)} \Rightarrow_M^1 P' \Rightarrow^{c(P)} sum(\{Q_i\}_{i \in I})$. By lemma 3 we build from P_n a uniform polysize constant depth circuit with $O(k(n).log(k))$ non-deterministic variables. For each of the $\Rightarrow^{c(P)}$ reductions we apply the same construction as in Terui's lemma starting from our previous configuration. At the end we easily build a polysize constant depth circuit to check if the last configuration represents b_1 or not. □

6 Proof net complexity

We define a hierarchy of complexity classes based on proof nets:

Definition 11. *For $i \in \mathbb{N}$, the* classes $UBPN^i(k(n))$ *and* $UBPN^i$ *are functions computable by uniform families of polynomial size,* $O(log^i n)$ *depth Boolean proof nets of respectively* $nMLL_u$ *with* $O(k(n))$ *sum-boxes and* MLL_u*. We write* $UBPN(k(n))$ *and* $UBPN$ *the respective unions over the exponents of the depth.*

From theorem 2 and theorem 3 we have:

Theorem 4. *For all $i \in \mathbb{N}$,*
$NAC^i(k(n))(stCONN_2) \subseteq UBPN^i(k(n)) \subseteq NAC^i(k(n) \times log\ n)(stCONN_2)$

Proof. For a Boolean proof net of size s the arity k of a sum-box is $O(s)$ in the worst case. By theorem 2 we have $O(s) = n^{O(1)}$. So $O(k(n).log\ k) = O(k(n) \times log\ n)$. □

Corollary 1. *For all $i, j \in \mathbb{N}$,*

1. $UBPN^i(poly) = NAC^i(poly)(stCONN_2) = NP$,
2. $NAC^i(log^j\ n)(stCONN_2) \subseteq UBPN^i(log^j\ n) \subseteq NAC^i(log^{j+1}\ n)(stCONN_2)$,
3. $UBPN(1) = UBPN = NC$,
4. $UBPN(log\ n) \supseteq NC$,
5. $UBPN(polylog) = NNC(polylog)$,
6. $UBPN(poly) = NNC(poly) = NP$.

Proof. Point 1. $O(n^{O(1)} \times log\ n) = O(n^{O(1)})$.
Point 3. $NAC(1)(stCONN_2) = NC = NNC(log\ n) = NAC(log\ n)(stCONN_2)$.
Point 5. by union over i and j from Point 2.
Point 6. by union over i from Point 1. □

Remark that $UBPN(1) = UBPN$ is what we expect: a constant number of sum-boxes corresponds to $n^{O(1)}$ summands/choices in the worst case, and so it can be simulated with a disjunction of a polynomial number of circuits of the same depth. I.e. it corresponds to $NNC(log\ n) = NC$.

7 Conclusion

Our study generalizes the work of K. Terui [Ter04] to non-deterministic extension as follows. We have defined the parallel reductions of proof nets of Non-deterministic Multiplicative Linear logic, $nMLL_u$. We have defined the uniform Boolean proof nets with an amount of explicit non-determinism analogously to the uniform Boolean circuits of $NNC()$, the non-deterministic NC class. By the way we give a proof-as-programs correspondence between this model of parallel computation and the uniform Boolean proof nets, preserving both size and depth. We define in a standard way the classes of uniform Boolean proof nets families $UBPN()$ and we establish the following results:

$$\begin{array}{ccccccccc} NC = UBPN(1) & \subseteq & UBPN(log\ n) & \subseteq & UBPN(polylog) & \subseteq & UBPN(poly) & = & NP \\ \shortparallel & & & & \shortparallel & & \shortparallel & & \\ UBPN & & & & NNC(polylog) & & NNC(poly) & & \end{array}$$

Remark that the proofs of translation and simulation theorems could apply for Boolean circuits without depth constraint. Such a Boolean circuit is simply called a polynomial size circuit and the corresponding class equal P. So there is a chain from P to NP for families of uniform polynomial size Boolean proof nets.

There is a reduction which replaces sequence of $\Rightarrow_D$ in our theorem in only one step: our circuit simulating parallel down reductions made it in lemma 3. The same thing could be done by parsing the sub-nets without reducing sum-boxes (only $k(n).log(n)$ bits are used) but with our theorem reduction we do fully parallel computation. Ad-hoc Boolean proof net classes can be given to have a more strictly correspondence with $NNC()$, using only binary $\square$-links. Then the encoding are no more constant depth but $O(k(n))$ depth: it corresponds to NC with $O(log\ n)$ sum-boxes. Remark that if we use a bigger uniformity notion then the descriptions are easier: e.g. for sum-boxes a relation between the $\square$-links is sufficient.

Acknowledgments. We would thank the reviewers for the improvements they have suggested and which will help the reader to understand this work.

References

[All89] Eric W. Allender. P-uniform circuit complexity. *Journal of the Association for Computing Machinery*, 36(4):912–928, 1989.

[BS90] R. B. Boppana and M. Sipser. *The complexity of finite functions.* MIT Press, 1990.

[Coo85] Stephen A. Cook. A taxonomy of problems with fast parallel algorithms. *Inf. Control*, 64(1-3):2–22, 1985.

[DR89] V. Danos and L. Regnier. The structure of multiplicatives. *Archive for Mathematical Logic*, 28(3):181–203, 1989.

[Gir87] Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50(1):1–102, 1987.

[Gir96] Jean-Yves Girard. Proof-nets : the parallel syntax for proof theory. *Logic and Algebra*, 180, 1996.

[Mau03] F. Maurel. Nondederministic light logics and NP-time. In *Proceedings of International Conference on Typed Lambda Calculus and Applications*, volume 2701 of *LNCS*, pages 241–255. Springer-Verlag, 2003.

[MR07] V. Mogbil and V. Rahli. Uniform circuits, & boolean proof nets. In *Proceedings International Symposium on Logical Foundations of Computer Science*, volume 4514 of *LNCS*, pages 401–421. Springer, 2007.

[Par89] I. Parberry. A note on nondeterminism in small, fast parallel computers. *IEEE Transactions on Computers*, 38(5):766–767, 1989.

[Ruz81] W. Ruzzo. On uniform circuit complexity. *J. of Computer and System Science*, 21:365–383, 1981.

[Ter04] K. Terui. Proof nets and boolean circuits. In *Proceedings IEEE Logic in Computer Science*, pages 182–191, 2004.

[Vol99] H. Vollmer. *Introduction to Circuit Complexity – A Uniform Approach.* Texts in Theoretical Computer Science. Springer Verlag, 1999.

[Wol94] Marty J. Wolf. Nondeterministic circuits, space complexity and quasigroups. *Theoretical Computer Science*, 125(2):295–313, 1994.

A Appendix: functions in Boolean proof nets

The ports of the proof nets are labeled by letters rather than numbers, so that proof nets described only by the name of their links are more readable. For a given link l, we denote $l_q^{p_1,\dots,p_n}$ if its conclusion port is q and the other ports are respectively $p_1,\dots,p_n$ in this order. Axiom link are more simply denoted ax_p.

The *conditional* (if-then-else) is the base of the Terui's gates translations: given two proof nets P_1 and P_2 of types $\vdash \Gamma, p_1 : A$ and $\vdash \Delta, p_2 : A$ resp., one can build a proof net $cond_r^{p_1,p_2}[P_1,P_2](q)$ of type $\vdash \Gamma, \Delta, q : \mathbf{B}[A]^\perp, r : A \otimes A$ (Fig.7(b)). Given a cut between b_i and q we have:

$$cond_r^{p_1,p_2}[P_1,P_2](b_1) \to^* tensor_r^{p_1,p_2}(P_1,P_2)$$
$$cond_r^{p_1,p_2}[P_1,P_2](b_0) \to^* tensor_r^{p_2,p_1}(P_2,P_1).$$

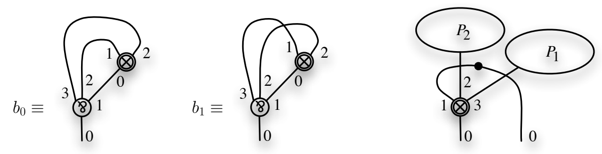


Fig. 7. (a) The Boolean b_0, b_1 and (b) The conditional Boolean proof net

Disjunction, *conjunction* and *duplication* are based on the conditional:
let $n \geqslant 2$ be an integer and $C \equiv \otimes(\mathbf{B}[A_1],\dots,\mathbf{B}[A_n])$, we have:

$or(p_1,p_2) \equiv cond[b_1, ax_{p_1}](p_2)$ of type $\vdash p_1 : \mathbf{B}^\perp, p_2 : \mathbf{B}[\mathbf{B}]^\perp, q : \mathbf{B} \otimes \mathbf{B}$,
$and(p_1,p_2) \equiv cond[ax_{p_1}, b_0](p_2)$ of type $\vdash p_1 : \mathbf{B}^\perp, p_2 : \mathbf{B}[\mathbf{B}]^\perp, q : \mathbf{B} \otimes \mathbf{B}$,
$copy^n(p) \equiv cond[tensor(\overrightarrow{b_1}), tensor(\overrightarrow{b_0})](p)$ of type $\vdash p : \mathbf{B}^\perp[C], q : C \otimes C$.

The *composition* of two translated circuits is defined by:
let $\Gamma \equiv p'_1 : A'_1,\dots,p'_n : A'_n$ and $\Delta \equiv q'_1 : B'_1,\dots,q'_n : B'_m$, let $P(\overrightarrow{p'})$ and $Q(\overrightarrow{q'})$ be proof nets of type $\vdash \Gamma, p : \otimes^2(\mathbf{B}, \overrightarrow{C})$ and $\vdash q : \mathbf{B}^\perp[A], \Delta, r : \otimes^2(\mathbf{B}, \overrightarrow{D})$, respectively. Then we have:

$$comp_s^{p,q,r}[P,Q](\overrightarrow{p'}, \overrightarrow{q'})$$

is of type $\vdash \Gamma[A], \Delta, s : \otimes^2(\mathbf{B}, \overrightarrow{D}, \overrightarrow{C[A]})$. With this composition one can construct n-ary versions of conjunction and disjunction.

Derivational Complexity is an Invariant Cost Model[⋆]

Ugo Dal Lago and Simone Martini

Dipartimento di Scienze dell'Informazione, Università di Bologna
Mura Anteo Zamboni 7, 40127 Bologna, Italy
{dallago,martini}@cs.unibo.it

Abstract. We show that in the context of orthogonal constructor rewriting systems, derivational complexity is an invariant cost model, both in innermost and in outermost reduction. This has some interesting consequences for (asymptotic) complexity analysis, since many existing methodologies only guarantee bounded derivational complexity.

1 Introduction

Genuine applications of computational complexity to rule-based programming are few and far between. Notwithstanding the development and diffusion of high-level programming languages of this kind, most part of the complexity analysis for them is bound to be done by reference to low level implementations with an explicit notion of constant-time computational step. Indeed, the basic step of computation for rule-based programming is not a constant-time operation, being it resolution, pattern matching, term-rewriting, higher-order beta-reduction, and so on. Therefore, to bound the actual execution time, one has to look at specific implementations of these mechanisms—in many cases, to revert to imperative programming and to an analysis on Turing machines.

In this paper we focus on (constructor) term rewriting systems and the problem of verifying the existence of a bound on the time needed to execute a program on any input, as a function of the input's length. Or, to devise sound (and necessarily incomplete) static techniques to infer such bounds. While real execution time clearly depends on the interpreter, for asymptotic bounds the details on the underlying interpreter become less crucial. More important, those details *should be* less crucial. For a sufficiently robust complexity class P (say, the polynomial-, or the elementary-time computable functions) proving that a given program may be executed in time bounded by a function in P should be the same whichever cost model is chosen, provided such a model is *invariant* [16]—that is, the cost attributed to the computation by the model differs by at most a polynomial from the cost of performing an equivalent computation on a Turing machine. And the simpler the cost model is, the easier is proving the existence of such bounds.

[⋆] The authors are partially supported by PRIN project "CONCERTO" and FIRB grant RBIN04M8S8, "Intern. Inst. for Applicable Math."

If programs are defined by rewriting, the most natural cost model for computation time is the one induced by rewriting. Time passes whenever rewriting is performed and, more importantly, the time needed to fire a redex is *assumed* to be unitary. Indeed, several recent papers propose methodologies for proving asymptotic bounds (typically, polynomial bounds) on the *derivational complexity* of term rewriting systems, where the derivational complexity of a term t is the number of rewriting steps necessary to reduce t to its normal form.

Is this cost model invariant? Or, which is the relation between the derivational complexity of a term and the time needed to rewrite it to normal form, observed on an efficient interpreter? The question is not trivial, even if a positive answer to the invariance issue seems part of the folklore. Indeed, the literature often distinguishes between results about computational complexity on the one hand and results about derivational complexity on the other. And there are examples of rewrite systems which produce exponentially big terms in a linear number of steps. For example, the one defined by the following rewrite rules (under innermost reduction): $f(0) = c$, $f(n+1) = g(f(n))$, and $g(x) = d(x, x)$.

Aim of this paper is to fill this gap, at least partially. More precisely, we prove that terms of orthogonal constructor rewrite systems can be reduced to their normal form in time polynomially related to their derivational complexity, both when innermost and outermost reduction is adopted. We prove this by showing that any such rewrite system can be implemented with (term) graph rewriting. In this setting, whenever a rewriting causes a duplication of a subterm (because of a non right-linear variable in a rule), the subgraph corresponding to the duplicated term is *shared* and not duplicated. This achieves two goals. First (and most important) the *size* of all the (graphs corresponding to) terms generated along the reduction of a term t remains under control—it is polynomial in the size of t and the number of reduction steps leading t to its normal form (see Section 5). Therefore, the actual cost of manipulating these graphs is bounded by a polynomial, thus giving also a polynomial relation between the number of reduction steps and the cost of producing the graph in normal form. Since we will prove that graph reduction simulates term rewriting *step by step* under innermost reduction, this gives us the desired invariance results in the innermost case.

In outermost reduction the situation is even better, because, in presence of sharing, every graph rewriting step corresponds to *at least* one term rewriting step. In graphs, in fact, shared redexes are reduced only once and, moreover, any redex appearing in an argument which will be later discarded, will not be reduced at all. Therefore, in presence of sharing outermost reduction becomes a call-by-need strategy. And hence the invariance result also for outermost reduction.

We believe the central argument and results of the paper may be classified as folklore (see, for instance, results of similar flavor, but for imperative programs with data-sharing, in Jones' computability textbook [10]). But we were struck by the observation that in the published literature it seems that such complexity related issues are never made explicit or used.

2 Term Rewriting

We will consider in this paper orthogonal constructor (term) rewrite systems (CRS, see [5]). Let Υ be a denumerable set of variables. A constructor (term) rewrite system is a pair $\Xi = (\Sigma_\Xi, \mathcal{R}_\Xi)$ where:

- Symbols in the signature Σ_Ξ can be either *constructors* or *function symbols*, each with its arity.
  - Terms in $\mathcal{C}(\Xi)$ are those built from constructors and are called *constructor terms*.
  - Terms in $\mathcal{P}(\Xi, \Upsilon)$ are those built from constructors and variables and are called *patterns*.
  - Terms in $\mathcal{T}(\Xi)$ are those built from constructor and function symbols and are called *closed terms*.
  - Terms in $\mathcal{V}(\Xi, \Upsilon)$ are those built from constructors, functions symbols and variables in Υ and are dubbed *terms*.
- Rules in $\mathcal{R}_\Xi$ are in the form $\mathbf{f}(\mathbf{p}_1, \ldots, \mathbf{p}_n) \rightarrow_\Xi t$ where $\mathbf{f}$ is a function symbol, $\mathbf{p}_1, \ldots, \mathbf{p}_n \in \mathcal{P}(\Xi, \Upsilon)$ and $t \in \mathcal{V}(\Xi, \Upsilon)$. We here consider orthogonal rewrite systems only, i.e. we assume that no distinct two rules in $\mathcal{R}_\Xi$ are overlapping and that every variable appears at most once in the lhs of any rule in $\mathcal{R}_\Xi$.
- Different notions of reduction can be defined on Ξ. The (binary) *rewriting relation* $\rightarrow$ on $\mathcal{T}(\Xi)$ is defined by imposing that $t \rightarrow u$ iff there are a rule $v \rightarrow_\Xi w$ in $\mathcal{R}_\Xi$, a term context (i.e., a term with a hole) C and a substitution σ with $t = C[v\sigma]$ and $u = C[w\sigma]$. Two restrictions of this definition are innermost and outermost reduction. In the *innermost rewriting relation* $\rightarrow_{\mathsf{i}}$ on $\mathcal{T}(\Xi)$ we require that $v\sigma$ do not contain another redex. Dually, the *outermost rewriting relation* $\rightarrow_{\mathsf{o}}$ on $\mathcal{T}(\Xi)$ is defined by requiring (the specific occurrence of) $v\sigma$ not to be part of another redex in $C[v\sigma]$.

For any term t in a CRS, $|t|$ denotes the number of symbol occurrences in t, while $|t|_{\mathbf{f}}$ denotes the number of occurrences of the symbol $\mathbf{f}$ in t.

Orthogonal CRSs are confluent but not necessarily strongly confluent [5]. As a consequence, different reduction sequences may have different lengths. This does not hold when considering only outermost (or only innermost) reductions:

Proposition 1. *Given a term t, every innermost (outermost, respectively) reduction sequence leading t to its normal form has the same length.*

Proof. Immediate, since there are no critical pairs when performing innermost or outermost reduction. □

As a consequence, it is meaningful to define the *outermost derivational complexity* of any t, written $\mathit{Time}_{\mathsf{o}}(t)$ as the unique n (if any) such that $t \rightarrow_{\mathsf{o}}^n u$, for u a normal form. Similarly, the *innermost derivational complexity* of t is the unique n such that $t \rightarrow_{\mathsf{i}}^n u$ (if any) and is denoted as $\mathit{Time}_{\mathsf{i}}(t)$.

3 Graph Rewriting

In this Section, we introduce term graph rewriting, following [4] but adapting the framework to (orthogonal) constructor rewriting.

Definition 1 (Labelled Graph). *Given a signature Σ, a* labelled graph *over Σ consists of a directed acyclic graph together with an ordering on the outgoing edges of each node and a (partial) labelling of nodes with symbols from Σ such that the out-degree of each node matches the arity of the corresponding symbols (and is 0 if the labelling is undefined). Formally, a labelled graph is a triple $G = (V, \alpha, \delta)$ where:*

- *V is a set of* vertices*.*
- *$\alpha : V \to V^*$ is a (total)* ordering function*.*
- *$\delta : V \rightharpoonup \Sigma$ is a (partial)* labelling function *such that the length of $\alpha(v)$ is the arity of $\delta(v)$ if $\delta(v)$ is defined and is 0 otherwise.*

A labelled graph (V, α, δ) is closed *iff δ is a total function.*

Consider the signature $\Sigma = \{a, b, c, d\}$, where arities of a, b, c, d are 2, 1, 0, 2 respectively, and b, c, d are constructors. Examples of labelled graphs over the signature Σ are the following ones:

$$\begin{tikzcd}
& a \arrow[d, bend right] \arrow[d, bend left] & \\
& b \arrow[d] & \\
& d \arrow[dl] \arrow[dr] & \\
b \arrow[d] & & c \\
\bot & &
\end{tikzcd}
\qquad
\begin{tikzcd}
a \arrow[dr, bend right] \arrow[dr, bend left] & & b \arrow[dl] \\
& \bot &
\end{tikzcd}
\qquad
\begin{tikzcd}
& a \arrow[d] \arrow[dd, bend right] & \\
& b \arrow[d] & \\
& a \arrow[dl] \arrow[dr] & \\
\bot & & b \arrow[d] \\
& & \bot
\end{tikzcd}$$

The symbol $\bot$ denotes vertices where the underlying labelling function is undefined (and, as a consequence, no edge departs from such vertices). Their role is similar to the one of variables in terms.

If one of the vertices of a labelled graph is selected as the *root*, we obtain a term graph:

Definition 2 (Term Graphs). *A* term graph*, is a quadruple $G = (V, \alpha, \delta, r)$, where (V, α, δ) is a labelled graph and $r \in V$ is the* root *of the term graph.*

The following are graphic representations of some term graphs. The root is the only vertex drawn inside a circle.

$$\begin{tikzcd}
& \textcircled{a} \arrow[d, bend right] \arrow[d, bend left] & \\
& b \arrow[d] & \\
& a \arrow[dl] \arrow[dr] & \\
b \arrow[d] & & c \\
\bot & &
\end{tikzcd}
\qquad
\begin{tikzcd}
a \arrow[dr, bend right] \arrow[dr, bend left] & & \textcircled{b} \arrow[dl] \\
& \bot &
\end{tikzcd}
\qquad
\begin{tikzcd}
& \textcircled{a} \arrow[d] \arrow[dd, bend right] & \\
& b \arrow[d] & \\
& a \arrow[dl] \arrow[dr] & \\
\bot & & b \arrow[d] \\
& & \bot
\end{tikzcd}$$

There are some classes of paths which are particularly relevant for our purposes.

Definition 3 (Paths). *A path $v_1, \ldots, v_n$ in a labelled graph $G = (V, \alpha, \delta)$ is said to be:*

- *A* pattern path *iff for every $1 \leq i \leq n$, $\delta(v_i)$ is either a constructor symbol or is undefined;*
- *A* left path *iff $n \geq 1$, the symbol $\delta(v_1)$ is a function symbol and $v_2, \ldots, v_n$ is a pattern path.*

The notion of an homomorphism between labelled graphs is not only interesting mathematically, but will be crucial in defining rewriting:

Definition 4 (Homomorphisms). *An* homomorphism *between two labelled graphs $G = (V_G, \alpha_G, \delta_G)$ and $H = (V_H, \alpha_H, \delta_H)$ over the same signature Σ is a function φ from V_G to V_H preserving the labelled graph structure. In particular*

$$\begin{aligned}\delta_H(\varphi(v)) &= \delta_G(v)\\ \alpha_H(\varphi(v)) &= \varphi^*(\alpha_G(v))\end{aligned}$$

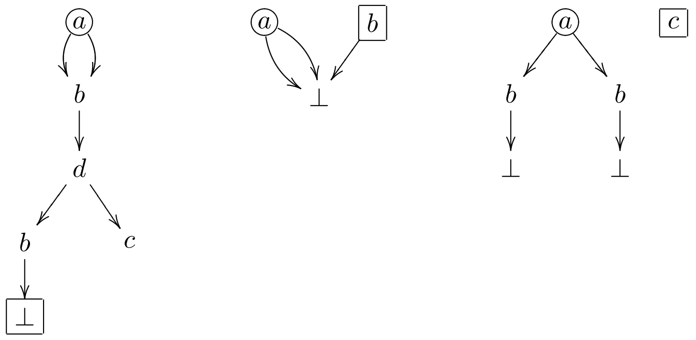
for any $v \in dom(\delta)$, where φ^* is the obvious generalization of φ to sequences of vertices. An homomorphism *between two term graphs $G = (V_G, \alpha_G, \delta_G, r_G)$ and $H = (V_H, \alpha_H, \delta_H, r_H)$ is an homomorphism between $(V_G, \alpha_G, \delta_G)$ and $(V_H, \alpha_H, \delta_H)$ such that $\varphi(r_G) = r_H$. Two labelled graphs G and H are isomorphic iff there is a bijective homomorphism from G to H; in this case, we write $G \cong H$. Similarly for term graphs.*

In the following, we will consider term graphs modulo isomorphism, i.e., $G = H$ iff $G \cong H$. Observe that two isomorphic term graphs have the same graphical representation.

Definition 5 (Graph Rewrite Rules). *A* graph rewrite rule *over a signature Σ is a triple $\rho = (G, r, s)$ such that:*

- *G is a labelled graph;*
- *r, s are vertices of G, called the* left root *and the* right root *of ρ, respectively.*
- *Any path starting in r is a left path.*

The following are three examples of graph rewriting rules, assuming a to be a function symbol and b, c, d to be constructors:



$$\begin{tikzcd}
& |[draw,circle]| a \arrow[d, bend left] \arrow[d, bend right] & \\
& b \arrow[d] & \\
& d \arrow[dl] \arrow[dr] & \\
b \arrow[d] & & c \\
\boxed{\perp} & &
\end{tikzcd}
\qquad
\begin{tikzcd}
|[draw,circle]| a \arrow[dr, bend left] \arrow[dr, bend right] & & \boxed{b} \arrow[dl] \\
& \perp &
\end{tikzcd}
\qquad
\begin{tikzcd}
& |[draw,circle]| a \arrow[dl] \arrow[dr] & & \boxed{c} \\
b \arrow[d] & & b \arrow[d] & \\
\perp & & \perp &
\end{tikzcd}$$

Graphically, the left root is the (unique) node inside a circle, while the right root is the (unique) node inside a square.

Definition 6 (Subgraphs). *Given a labelled graph $G = (V_G, \alpha_G, \delta_G)$ and any vertex $v \in V_G$, the* subgraph of G rooted at v, *denoted $G \downarrow v$, is the term graph $(V_{G\downarrow v}, \alpha_{G\downarrow v}, \delta_{G\downarrow v}, r_{G\downarrow v})$ where*

- *$V_{G\downarrow v}$ is the subset of V_G whose elements are vertices which are reachable from v in G.*
- *$\alpha_{G\downarrow v}$ and $\delta_{G\downarrow v}$ are the appropriate restrictions of α_G and δ_G to $V_{G\downarrow v}$.*
- *$r_{G\downarrow v}$ is v.*

We are finally able to give the notion of a redex, that represents the occurrence of the lhs of a rewrite rule in a graph:

Definition 7 (Redexes). *Given a labelled graph G, a* redex *for G is a pair (ρ, φ), where ρ is a rewrite rule (H, r, s) and φ is an homomorphism between $H \downarrow r$ and G.*

If $((H, r, s), \varphi)$ is a redex in G, we say, with a slight abuse of notation, that $\varphi(r)$ *is* itself a redex. In most cases, this does not introduce any ambiguity.

Given a term graph G and a redex $((H, r, s), \varphi)$, the result of firing the redex is another term graph obtained by successively applying the following three steps to G:

1. The *build phase*: create an isomorphic copy of the portion of $H \downarrow s$ not contained in $H \downarrow r$ (which may contain arcs originating in $H \downarrow s$ and entering $H \downarrow r$), and add it to G, obtaining J. The underlying ordering and labelling functions are defined in the natural way.
2. The *redirection phase*: all edges in J pointing to $\varphi(r)$ are replaced by edges pointing to the copy of s. If $\varphi(r)$ is the root of G, then the root of the newly created graph will be the newly created copy of s. The graph K is obtained.
3. The *garbage collection phase*: all vertices which are not accessible from the root of K are removed. The graph I is obtained.

We will write $G \xrightarrow{(H,r,s)} I$ (or simply $G \to I$, if this does not cause ambiguity) in this case.

Similarly to what we did for term rewriting, we can define two restrictions on $\to$ as follows. Let $((H, r, s), \varphi)$ be a redex in G. Then it is said to be

- An *innermost* redex iff for every redex $((J, p, q), \psi)$ in G, there is no proper path from $\varphi(r)$ to $\psi(p)$.
- An *outermost* redex iff for every redex $((J, p, q), \psi)$ in G, there is no proper path from $\psi(p)$ to $\varphi(r)$.

If the redex $((H, r, s), \varphi)$ is innermost we also write $G \xrightarrow{(H,r,s)}_{\mathsf{i}} I$ or $G \to_{\mathsf{i}} I$. Similarly, for an outermost redex $((H, r, s), \varphi)$ we write $G \xrightarrow{(H,r,s)}_{\mathsf{o}} I$ or $G \to_{\mathsf{o}} I$.

As an example, assuming again a to be a function symbol and b, c, d to be constructors, consider the term graph G and the rewriting rule $\rho = (H, r, s)$:

$$\begin{tikzcd}
& \textcircled{a} \arrow[dl] \arrow[dr] & \\
b \arrow[d] & & a \arrow[ll] \arrow[dll] \\
c & &
\end{tikzcd}
\qquad
\begin{tikzcd}
\textcircled{a} \arrow[d] \arrow[dr, to=3-2] & \boxed{b} \arrow[d] \\
b \arrow[d] & a \arrow[l] \arrow[d] \\
\bot & c
\end{tikzcd}$$

G $\qquad\qquad$ ρ

There is an homomorphism φ from $H \downarrow r$ to G. In particular, φ maps r to the rightmost vertex in G. Applying the build phase and the redirection phase we get J and K as follows:

$$\begin{tikzcd}
& \textcircled{a} \arrow[dl] \arrow[dr] & & b \arrow[d] \\
b \arrow[d] & & a \arrow[ll] \arrow[dll] & a \arrow[lll, bend right] \arrow[dlll, bend left] \\
c & & &
\end{tikzcd}
\qquad
\begin{tikzcd}
& \textcircled{a} \arrow[dl] \arrow[rr] & & b \arrow[d] \\
b \arrow[d] & & a \arrow[ll] \arrow[dll] & a \arrow[lll, bend right] \arrow[dlll, bend left] \\
c & & &
\end{tikzcd}$$

J $\qquad\qquad$ K

Finally, applying the garbage collection phase, we get the result of firing the redex (ρ, φ):

$$\begin{tikzcd}
& \textcircled{a} \arrow[d] \arrow[dddl, bend right] & \\
& b \arrow[d] & \\
& a \arrow[dl] \arrow[dr] & \\
b \arrow[rr] & & c
\end{tikzcd}$$

I

Given two graph rewrite rules $\rho = (H, r, s)$ and $\sigma = (J, p, q)$, ρ and σ are said to be *overlapping* iff there is a term graph G and two homomorphism φ and ψ such that (ρ, φ) and (σ, ψ) are both redexes in G with $\varphi(r) = \varphi(p)$.

Definition 8. *A constructor graph rewrite system (CGRS) over a signature Σ consists of a set of non-overlapping graph rewrite rules $\mathcal{G}$ on Σ.*

We now want to give some confluence results for CGRSs. Let us first focus on outermost rewriting. Intuitively, outermost rewriting is the most efficient way of performing reduction in presence of sharing, since computation is performed only if its result does not risk to be erased. First, we need the following auxiliary lemma:

Lemma 1. *Suppose $G \rightarrow_{\mathsf{o}} H$ and $G \rightarrow J$, where $H \neq J$. Then either $J \rightarrow_{\mathsf{o}} H$ or there is K such that $H \rightarrow K$ and $J \rightarrow_{\mathsf{o}} K$.*

Proof. Let v and w be the two redexes in G giving rise to $G \rightarrow_{\mathsf{o}} H$ and $G \rightarrow J$, respectively. Similarly, let ρ and σ be the two involved rewriting rules. Clearly, there cannot be any (non-trivial) path from w to v, by definition of outermost rewriting. Now, the two rewriting steps are independent from each other (because of the non-overlapping condition). There are now two cases. Either w is erased when performing $G \rightarrow_{\mathsf{o}} H$, or it is not erased. In the first case, w must be "contained" in v, and therefore, we may apply ρ to J, obtaining H. If w has not been erased, one can clearly apply ρ to J and σ to H, obtaining a fourth graph K. □

The observation we just made can be easily turned into a more general result on reduction sequences of arbitrary length:

Proposition 2. *Suppose that $G \rightarrow_{\mathsf{o}}^{n} H$ and $G \rightarrow^{m} J$. Then there are K and $k, l \in \mathbb{N}$ such that $H \rightarrow^{k} J$, $J \rightarrow_{\mathsf{o}}^{l} K$ and $n + k \leq m + l$.*

Proof. An easy induction on $n + m$. □

Proposition 2 tells us that if we perform n outermost steps and m generic steps from G, we can close the diagram in such a way that the number of steps in the first branch is smaller or equal to the number of steps in the second branch.

With innermost reduction, the situation is exactly dual:

Lemma 2. *Suppose $G \rightarrow H$ and $G \rightarrow_{\mathsf{i}} J$, where $H \neq J$. Then either $J \rightarrow H$ or there is K such that $H \rightarrow_{\mathsf{i}} K$ and $J \rightarrow K$.*

Proposition 3. *Suppose that $G \rightarrow^{n} H$ and $G \rightarrow_{\mathsf{i}}^{m} J$. Then there are K and $k, l \in \mathbb{N}$ such that $H \rightarrow_{\mathsf{i}}^{k} J$, $J \rightarrow^{l} K$ and $n + k \leq m + l$.*

In presence of sharing, therefore, outermost reduction is the best one can do, while innermost reduction is the worst strategy, since we may reduce redexes in subgraphs that will be later discarded. As a by-product, we get confluence:

Theorem 1. *Suppose that $G \rightarrow_{\mathsf{o}}^{n} H$, $G \rightarrow^{m} J$ and $G \rightarrow_{\mathsf{i}}^{k} K$, where H, J and K are normal forms. Then $H = J = K$ and $n \leq m \leq k$.*

Proof. From $G \rightarrow_{\mathsf{o}}^{n} H$, $G \rightarrow^{m} J$ and Proposition 2, it follows that $n \leq m$ and that $H = J$. From $G \rightarrow_{\mathsf{o}}^{m} J$, $G \rightarrow^{k} K$ and Proposition 3, it follows that $m \leq k$. □

4 From Term Rewriting to Graph Rewriting

Any term t over the signature Σ can be turned into a graph G in the obvious way: G will be a tree and vertices in G will be in one-to-one correspondence with symbol occurrences in t. Conversely, any term graph G over Σ can be turned into a term t over Σ (remember: we only consider acyclic graphs here).

Similarly, any term rewrite rule $t \rightarrow u$ over the signature Σ can be translated into a graph rewrite rule (G, r, s) as follows:

- Take the graphs representing t and u. They are trees, in fact.
- From the union of these two trees, share those nodes representing the same variable in t and u. This is G.
- Take r to be the root of t in G and s to be the root of u in G.
  As an example, consider the rewriting rule

$$a(b(x), y) \to b(a(y, a(y, x))).$$

Its translation as a graph rewrite rule is the following:

$$\begin{tikzcd}
 & \textcircled{a} \arrow[dl] \arrow[dr] & & \boxed{b} \arrow[d] & \\
b \arrow[d] & & \perp & a \arrow[l] \arrow[dr] & \\
\perp & & & & a \arrow[ull, bend left] \arrow[llll, bend left]
\end{tikzcd}$$

An arbitrary constructor rewriting system can be turned into a constructor graph rewriting system:

Definition 9. *Given a constructor rewriting system $\mathcal{R}$ over Σ, the corresponding constructor graph rewriting system $\mathcal{G}$ is defined as the class of graph rewrite rules corresponding to those in $\mathcal{R}$. Given a term t, $[t]_{\mathcal{G}}$ will be the corresponding graph, while the term graph G corresponds to the term $\langle G\rangle_{\mathcal{R}}$.*

Let us now consider graph rewrite rules corresponding to rewrite rules in $\mathcal{R}$. It is easy to realize that the following invariant is preserved while performing innermost rewriting in $[\mathcal{R}]_{\mathcal{G}}$: whenever any vertex v can be reached by two distinct paths starting at the root (i.e., v is *shared*), v cannot be a redex, i.e., there cannot be a redex $((G, r, s), \varphi)$ such that $\varphi(r) = v$. A term graph satisfying this invariant is said to be *redex-unshared.*

Redex-unsharedness holds for term graphs coming from terms and is preserved by innermost graph rewriting:

Lemma 3. *For every closed term t, $[t]_{\mathcal{G}}$ is redex-unshared. Moreover, if G is closed and redex-unshared and $G \to_{\mathsf{i}} I$, then I is redex-unshared.*

Proof. The fact $[t]_{\mathcal{G}}$ is redex-unshared for every t follows from the way the $[\cdot]_{\mathcal{G}}$ map is defined: it does not introduce any sharing. Now, suppose G is redex-unshared and

$$G \overset{(H,r,s)}{\longrightarrow}_{\mathsf{i}} I$$

where (H, r, s) corresponds to a term rewrite rule $t \to u$. The term graph J obtained from G by the build phase is itself redex-unshared: it is obtained from G by adding some new nodes, namely an isomorphic copy of the portion of $H \downarrow s$ not contained in $H \downarrow r$. Notice that J is redex-unshared in a stronger sense: any vertex which can be reached from the newly created copy of s by two distinct paths cannot be a redex. This is a consequence of (H, r, s) being a graph rewrite rule corresponding to a term rewrite rule $t \to u$, where the only shared vertices are those where the labelling function is undefined. The redirection phase preserves itself redex-unsharedness, because only one pointer is redirected (the vertex is labelled by a function symbol) and the destination of this redirection is a vertex (the newly created copy of s) which had no edge incident to it. Clearly, the garbage collection phase preserve redex-unsharedness. $\square$

Lemma 4. *A closed term graph G in $\mathcal{G}$ is a normal form iff $\langle G\rangle_{\mathcal{R}}$ is a normal form.*

Proof. Clearly, if a closed term graph G is in normal form, then $\langle G\rangle_{\mathcal{R}}$ is a term in normal form, because each redex in G translates to a redex in $\langle G\rangle_{\mathcal{R}}$. On the other hand, if $\langle G\rangle_{\mathcal{R}}$ is in normal form, then G is in normal form: each redex in $\langle G\rangle_{\mathcal{R}}$ translates back to a redex in G. $\square$

Reduction on graphs correctly simulates reduction on terms:

Lemma 5. *If $G \rightarrow I$, then $\langle G\rangle_{\mathcal{R}} \rightarrow^{+} \langle I\rangle_{\mathcal{R}}$. Moreover, if $G \rightarrow_{\mathsf{i}} I$ and G is redex-unshared, then $\langle G\rangle_{\mathcal{R}} \rightarrow \langle I\rangle_{\mathcal{R}}$.*

Proof. The fact each reduction step starting in G can be mimicked by n reduction steps in $\langle G\rangle_{\mathcal{R}}$ is known from the literature. If G is redex-unshared, then $n = 1$, because no redex in a redex-unshared term graph can be shared. $\square$

As an example, consider the term rewrite rule $a(c, c) \rightarrow c$ and the following term graph, which is not redex-unshared and correspond to $a(a(c, c), a(c, c))$:

$$\begin{tikzcd}
& @ \arrow[d, bend right] \arrow[d, bend left] & \\
& a \arrow[dl] \arrow[dr] & \\
c & & c
\end{tikzcd}$$

The term graph rewrites in *one* step to the following one

$$\begin{tikzcd}
@ \arrow[d, bend right] \arrow[d, bend left] \\
c
\end{tikzcd}$$

while the term $a(a(c, c), a(c, c))$ rewrites to $a(c, c)$ in *two* steps.

Lemma 6. *If $t \rightarrow_{\mathsf{o}}^{n} u$, u is in normal form and $\langle G\rangle_{\mathcal{R}} = t$, then there is $m \leq n$ such that $G \rightarrow_{\mathsf{o}}^{m} I$, where $\langle I\rangle_{\mathcal{R}} = u$.*

Proof. An easy consequence of Lemma 5 and Proposition 1. $\square$

Theorem 2 (Outermost Graph-Reducibility). *For every constructor rewrite system $\mathcal{R}$ over Σ and for every term t over Σ, the following two conditions are equivalent:*

1. *$t \rightarrow_{\mathsf{o}}^{n} u$, where u is in normal form;*

2. $[t]_{\mathcal{G}} \rightarrow_{\mathsf{o}}^{m} G$, *where* G *is in normal form and* $\langle G\rangle_{\mathcal{R}} = u$.
Moreover, $m \leq n$.

Proof. Suppose $t \rightarrow_{\mathsf{o}}^{n} u$, where u is in normal form. Then, by applying Lemma 6, we obtain a normal form G such that $[t]_{\mathcal{G}} \rightarrow_{\mathsf{o}}^{m} G$, where $m \leq n$ and $\langle G\rangle_{\mathcal{R}} = u$. Now, suppose $[t]_{\mathcal{G}} \rightarrow_{\mathsf{o}}^{m} G$ where $\langle G\rangle_{\mathcal{R}} = u$ and G is in normal form. By applying n times Lemma 5, we obtain that $\langle [t]_{\mathcal{G}}\rangle_{\mathcal{R}} \rightarrow^{n} \langle G\rangle_{\mathcal{R}} = u$ where $m \leq n$. But $\langle [t]_{\mathcal{G}}\rangle_{\mathcal{R}} = t$ and u is a normal form by Lemma 4, since G is normal. □

The innermost case can be treated in a similar way:

Lemma 7. *If* $t \rightarrow_{\mathsf{i}}^{n} u$, *$u$ is in normal form and* $\langle G\rangle_{\mathcal{R}} = t$ *and G is redex-unshared, then* $G \rightarrow_{\mathsf{i}}^{n} I$, *where* $\langle I\rangle_{\mathcal{R}} = u$.

Proof. An easy consequence of Lemma 5 and Proposition 1. □

Theorem 3 (Innermost Graph Reducibility). *For every constructor rewrite system $\mathcal{R}$ over Σ and for every term t over Σ, the following two conditions are equivalent:*
1. $t \rightarrow_{\mathsf{i}}^{n} u$, *where u is in normal form;*
2. $[t]_{\mathcal{G}} \rightarrow_{\mathsf{i}}^{n} G$, *where G is in normal form and* $\langle G\rangle_{\mathcal{R}} = u$.

5 Consequences for Complexity Analysis

Theorems 2 and 3 tell us that term graph rewriting faithfully simulates term rewriting, with both outermost and innermost rewriting. In the outermost case, graph rewriting may perform better than term rewriting, because redex can be shared and one graph rewriting step may correspond to more than one term rewriting step. In innermost reduction, on the other hand, every graph step corresponds to exactly one term rewriting step.

But how much does it cost to perform reduction in a graph rewrite system $\mathcal{G}$ corresponding to a term rewrite system $\mathcal{R}$? Let us analyze more closely the combinatorics of graph rewriting, fixing our attention to outermost rewriting for the moment:
- Consider a closed term t and a term graph G such that $[t]_{\mathcal{G}} \rightarrow_{\mathsf{o}}^{*} G$.
- Every graph rewriting step makes the underlying graph bigger by at most the size of the rhs of a rewrite rule. So, if $[t]_{\mathcal{G}} \rightarrow_{\mathsf{o}}^{*} G \rightarrow_{\mathsf{o}} H$, then the difference $|H| - |G|$ cannot be too big: at most a constant k depending on $\mathcal{R}$ but independent on t. As a consequence, if $[t]_{\mathcal{G}} \rightarrow_{\mathsf{o}}^{n} G$ then $|G| \leq nk + |t|$. Here, we exploit in an essential way the possibility of sharing subterms.
- Whenever $[t]_{\mathcal{G}} \rightarrow_{\mathsf{o}}^{n} G$, computing a graph H such that $G \rightarrow H$ takes polynomial time in $|G|$, which is itself polynomially bounded by n and $|t|$.

Exactly the same reasoning can be applied to innermost reduction. Hence:

Theorem 4. *For every orthogonal, constructor term rewriting system $\mathcal{R}$, there is a polynomial* $p : \mathbb{N}^2 \rightarrow \mathbb{N}$ *such that for every term t the normal form of* $[t]_{\mathcal{G}}$ *can be computed in time at most* $p(|t|, \mathit{Time}_{\mathsf{o}}(M))$ *when performing outermost graph reduction and in time* $p(|t|, \mathit{Time}_{\mathsf{i}}(M))$ *when performing innermost graph reduction.*

We close this section by observing explicitly that the normal form of $[t]_{\mathcal{G}}$ is not a direct representation of the normal form of t. It may contain many shared subterms, that have to be "unshared" if one wants to print the normal form of t. As a limit example consider the system we already mentioned in the introduction: $f(0) = c$, $f(n+1) = g(f(n))$, and $g(x) = d(x,x)$. Here $f(n)$ will normalize in $O(n)$ steps with innermost reduction, but the normal form *as a term* is of size $O(2^n)$, being the complete binary tree of height n. We believe this has to be considered a feature of our cost model, allowing to distinguish the time (and space) needed for the computation from the one necessary for the communication of the result.

Despite the succinct representation of data via shared graphs, equality of terms is efficiently computed on graph representatives. We state this as a proposition, being of interest in its own.

Proposition 4. *Given two term graphs G and H, it is decidable in time polynomial in $|G| + |H|$ whether $\langle G\rangle_{\mathcal{R}} = \langle H\rangle_{\mathcal{R}}$.*

Proof. We can give a naive procedure working in quadratic time as follows. More efficient algorithms are available, for instance Paige and Tarjan's one for bisimulation, which runs in time $O(|E|\log|V|)$, where E and V are the sets of edges and vertices, respectively, of the graphs.

The decision procedure will fill a $m_1 \times m_2$ matrix, with m_1 and m_2 the number of nodes of G and H, respectively, using dynamic programming. Any element will contain either $\surd$ (OK) or $\times$ (fail). Start by filling all the elements (c_G, c_H) where c_G is a sink of G and c_H is a sink of H (a sink is a node labeled with a constructor of arity 0). Fill it with $\surd$ if they are the same constructor; with $\times$ otherwise. Now proceed along the inverse of the topological order (that is, go to the nodes pointing to the ones you just considered), and fill any such element (d_G, d_H) with $\surd$, if they are the same constructor *and* all the pairs (c_G, c_H) — with c_G successor in topological order of d_G and c_H successor in topological order of d_H — are marked with $\surd$. Otherwise, fill (d_G, d_H) with $\times$. At the end return $\surd$ iff the two roots are marked with $\surd$. □

6 Context and Related Work

Graph-reducibility of any orthogonal term rewrite system is well known [14]. However, this classical result do not mention anything about the relation between the complexity of term rewriting and the one of graph-rewriting. Quantitative analysis of the involved simulations is outside the scope of the classical results on the subject.

Asymptotic complexity analysis in the context of term rewriting systems has received a lot of attention in the last ten years [13, 6, 3, 9]. In some cases time complexity results are a consequence of an argument taking into account both the number of reduction steps and some other parameter (e.g., the size of intermediate terms [13]), so to bound the actual cost of the computation with an *ad hoc* combination of these two dimensions. In other cases [3, 9], results about the derivational complexity of TRS are kept distinct from other results about actual computation time. This body of research has been the main motivation for our work.

In a recent paper [7] we proved a close correspondence between orthogonal constructor term rewrite systems and weak call-by-value λ-calculus. In particular the two systems can simulate each other with a linear overhead, taking as cost model for both systems the number of reduction steps to normal form, that is the most natural one. This should not confuse the reader who knows that "optimal λ-reduction is not elementary recursive" [2, 1], meaning that there are terms whose normalization requires on a Turing machine a time hyperexponential in the number of optimal beta-reductions (which are a sophisticated form of graph-rewritings with partial sharing). For these results to hold is essential to take *full* beta-reduction, where we are allowed to reduce a redex also inside a λ-abstraction.

Graph rewriting has been considered in this paper as a technical tool to obtain our main result. An interesting research line would be to situate graph rewriting – and its complexity theory – in the context of those other machine models with a dynamically changing configuration structure, like Knuth's "linking automata" [11], Schönage's storage modification machines [15], and especially their common moral ancestor — Kolmogorov and Uspensky's machines [12, 8]. This would be particularly interesting in the study of classes like linear time and real time. Indeed, while the class of polynomial functions is very robust and coincide on all these different models (and on Turing machines, of course), these automata seem to give a better understanding of the lower complexity classes. After some preliminary investigation, the authors are convinced that the task of relating term graph rewriting and pointer machines from a complexity point of view is not trivial. For example, garbage collection is a non-local operation that is implicitly performed as part of any term graph rewriting step, while in pointer machines only the "programmer" is responsible for such (potentially costly) operations.

References

1. A. Asperti, P. Coppola, and S. Martini. (Optimal) duplication is not elementary recursive. *Inf. Comput.*, 193(1):21–56, 2004.
2. A Asperti and H. G. Mairson. Parallel beta reduction is not elementary recursive. *Inf. Comput.*, 170(1):49–80, 2001.
3. Martin Avanzini and Georg Moser. Complexity analysis by rewriting. In *FLOPS*, pages 130–146, 2008.
4. H. Barendregt, M. Eekelen, J. Glauert, J. Kennaway, M. Plasmeijer, and M. Sleep. Term graph rewriting. In J. de Bakker, A. Nijman, and P. Treleaven, editors, *Volume II: Parallel Languages on PARLE: Parallel Architectures and Languages Europe*, pages 141–158. Springer-Verlag, 1986.
5. Erik Barendsen. Term graph rewriting. In Terese (M. Bezem, J.W. Klop, and R. de Vrijer), editors, *Term Rewriting Systems*, chapter 13, pages 712–743. Cambridge Univ. Press, 2003.

6. Guillaume Bonfante, Jean-Yves Marion, and Jean-Yves Moyen. Quasi-interpretations and small space bounds. In *Term Rewriting and Applications*, volume 3467 of *LNCS*, pages 150–164. Springer, 2005.
7. Ugo Dal Lago and Simone Martini. On constructor rewriting systems and the lambda calculus. In *ICALP*, volume 5556 of *LNCS*, pages 163–174. Springer, 2009.
8. Yuri Gurevich. On Kolmogorov machines and related issues. *Bulletin of the European Association for Theoretical Computer Science*, 35:71–82, 1988.
9. Nao Hirokawa and Georg Moser. Automated complexity analysis based on the dependency pair method. In *IJCAR*, pages 364–379, 2008.
10. Neil D. Jones. *Computability and Complexity from a Programming Perspective.* MIT Press, 1997.
11. Donald Knuth. *The Art of Computer Programming, Vol. 1.* Prentice Hall, 1973. Pages 462-463.
12. A.N. Kolmogorov and V. Uspensky. On the definition of algorithm. *Uspekhi Mat. Naut*, 13(4):3–28, 1958. In Russian. English translation in AMS Translations, series 2, vol. 21(1963), 217-245.
13. Jean-Yves Marion and Jean-Yves Moyen. Efficient first order functional program interpreter with time bound certifications. In *Logic for Programming and Automated Reasoning, 7th International Conference, Proceedings*, volume 1955 of *LNCS*, pages 25–42. Springer, 2000.
14. Detlef Plump. Graph-reducible term rewriting systems. In *Graph-Grammars and Their Application to Computer Science*, pages 622–636, 1990.
15. A. Schönage. Storage modification machines. *SIAM J. Comput.*, 9:490–508, 1980.
16. Peter van Emde Boas. Machine models and simulation. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*, pages 1–66. MIT Press, 1990.

Comparing Cost Functions in Resource Analysis

E. Albert[1], P. Arenas[1], S. Genaim[1], I. Herraiz[1] and G. Puebla[2]

[1] DSIC, Complutense University of Madrid, E-28040 Madrid, Spain
[2] CLIP, Technical University of Madrid, E-28660 Boadilla del Monte, Madrid, Spain

Abstract. *Cost functions* provide information about the amount of resources required to execute a program in terms of the sizes of input arguments. They can provide an upper-bound, a lower-bound, or the average-case cost. Motivated by the existence of a number of automatic cost analyzers which produce cost functions, we propose an approach for automatically proving that a cost function is smaller than another one. In all applications of resource analysis, such as resource-usage verification, program synthesis and optimization, etc., it is essential to compare cost functions. This allows choosing an implementation with smaller cost or guaranteeing that the given resource-usage bounds are preserved. Unfortunately, automatically generated cost functions for realistic programs tend to be rather intricate, defined by multiple cases, involving non-linear subexpressions (e.g., exponential, polynomial and logarithmic) and they can contain multiple variables, possibly related by means of constraints. Thus, comparing cost functions is far from trivial. Our approach first syntactically transforms functions into simpler forms and then applies a number of sufficient conditions which guarantee that a set of expressions is smaller than another expression. Our preliminary implementation in the COSTA system indicates that the approach can be useful in practice.

1 Introduction

Cost analysis [12,6] aims at statically predicting the resource consumption of programs. Given a program, cost analysis produces a *cost function* which approximates the resource consumption of the program in terms of the input data sizes. This approximation can be in the form of an upper-bound, a lower-bound, or the average-case resource consumption, depending on the particular analysis and the target application. For instance, upper bounds are required to ensure that a program can run within the resources available; lower bounds are useful for scheduling distributed computations. The seminal cost analysis framework by Wegbreit [12] was already generic on the notion of *cost model*, e.g., it can be used to measure different resources, such as the number of instructions executed, the memory allocated, the number of calls to a certain method, etc. Thus, cost functions can be used to predict any of such resources.

In all applications of resource analysis, such as resource-usage verification, program synthesis and optimization, etc., it is necessary to compare cost functions. This allows choosing an implementation with smaller cost or to guarantee that the given resource-usage bounds are preserved. Essentially, given a method

m, a cost function f_m and a set of linear constraints ϕ_m which impose size restrictions (e.g., that a variable in m is larger than a certain value or that the size of an array is non zero, etc.), we aim at comparing it with another cost function bound $\mathtt{b}$ and corresponding size constraints $\phi_{\mathtt{b}}$. Depending on the application, such functions can be automatically inferred by a resource analyzer (e.g., if we want to choose between two implementations), one of them can be user-defined (e.g., in resource usage verification one tries to verify, i.e., prove or disprove, *assertions* written by the user about the efficiency of the program).

From a mathematical perspective, the problem of cost function comparison is analogous to the problem of proving that the difference of both functions is a decreasing or increasing function, e.g., $\mathtt{b} - f_m \geq 0$ in the context $\phi_{\mathtt{b}} \wedge \phi_m$. This is undecidable and also non-trivial, as cost functions involve non-linear subexpressions (e.g., exponential, polynomial and logarithmic subexpressions) and they can contain multiple variables possibly related by means of constraints in $\phi_{\mathtt{b}}$ and ϕ_m. In order to develop a practical approach to the comparison of cost functions, we take advantage of the form that cost functions originating from the analysis of programs have and of the fact that they evaluate to non-negative values. Essentially, our technique consists in the following steps:

1. Normalizing cost functions to a form which make them amenable to be syntactically compared, e.g., this step includes transforming them to sums of products of basic cost expressions.
2. Defining a series of comparison rules for basic cost expressions and their (approximated) differences, which then allow us to compare two products.
3. Providing sufficient conditions for comparing two sums of products by relying on the product comparison, and enhancing it with a *composite* comparison schema which establishes when a product is larger than a sum of products.

We have implemented our technique in the COSTA system [3], a COSt and Termination Analyzer for Java bytecode. Our experimental results demonstrate that our approach works well in practice, it can deal with cost functions obtained from realistic programs and verifies user-provided upper bounds efficiently.

The rest of the paper is organized as follows. The next section introduces the notion of cost bound function in a generic way. Sect. 3 presents the problem of comparing cost functions and relates it to the problem of checking the inclusion of functions. In Sect. 4, we introduce our approach to prove the inclusion of one cost function into another. Section 5 describes our implementation and how it can be used online. In Sect. 6, we conclude by overviewing other approaches and related work.

2 Cost Functions

Let us introduce some notation. The sets of natural, integer, real, non-zero natural and non-negative real values are denoted by $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$, $\mathbb{N}^+$, and $\mathbb{R}^+$, respectively. We write x, y, and z, to denote variables which range over $\mathbb{Z}$. A *linear* *expression* has the form $v_0 + v_1x_1 + \ldots + v_nx_n$, where $v_i \in \mathbb{Z}$, $0 \leq i \leq n$. Similarly, a *linear constraint* (over $\mathbb{Z}$) has the form $l_1 \leq l_2$, where l_1 and l_2 are linear expressions. For simplicity we write $l_1 = l_2$ instead of $l_1 \leq l_2 \wedge l_2 \leq l_1$, and $l_1 < l_2$ instead of $l_1 + 1 \leq l_2$. Note that constraints with rational coefficients can be always transformed into equivalent constraints with integer coefficients, e.g., $\frac{1}{2}x > y$ is equivalent to $x > 2y$. The notation $\bar{t}$ stands for a sequence of entities $t_1, \ldots, t_n$, for some $n{>}0$. We write φ, ϕ or ψ, to denote sets of linear constraints which should be interpreted as the conjunction of each element in the set. An assignment σ over a tuple of variables $\bar{x}$ is a mapping from $\bar{x}$ to $\mathbb{Z}$. We write $\sigma \models \varphi$ to denote that $\sigma(\varphi)$ is satisfiable.

The following definition presents our notion of *cost expression*, which characterizes syntactically the kind of expressions we deal with.

Definition 1 (cost expression). Cost expressions *are symbolic expressions which can be generated using this grammar:*

$e ::= n \mid \mathsf{nat}(l) \mid e + e \mid e * e \mid \log_a(\mathsf{nat}(l) + 1) \mid \mathsf{nat}(l)^n \mid a^{\mathsf{nat}(l)} \mid \mathsf{max}(S)$

where $n, a \in \mathbb{N}^+$ *and* $a \geq 2$, *l is a linear expression, S is a non empty set of cost expressions,* $\mathsf{nat} : \mathbb{Z} \rightarrow \mathbb{N}$ *is defined as* $\mathsf{nat}(v)=\mathsf{max}(\{v, 0\})$. *Given an assignment σ and a basic cost expression e, $\sigma(e)$ is the result of evaluating e w.r.t. σ.*

Observe that linear expressions are always wrapped by nat, as we will explain below in the example. Logarithmic expressions contain a linear subexpression plus "1" which ensures that they cannot be evaluated to $\log_a(0)$. By ignoring syntactic differences, cost analyzers produce cost expressions in the above form.

It is customary to analyze programs (or methods) w.r.t. some initial *context constraints*. Essentially, given a method $m(\bar{x})$, the considered context constraints φ describe conditions on the (sizes of) initial values of $\bar{x}$. With such information, a cost analyzer outputs a *cost function* $f_m(\bar{x}_s) = \langle e, \varphi \rangle$ where e is a cost expression and $\bar{x}_s$ denotes the data sizes of $\bar{x}$. Thus, f_m is a function of the input data sizes that provides bounds on the resource consumption of executing m for any concrete value of the input data $\bar{x}$ such that their sizes satisfy φ. Note that φ is basically a set of linear constraints over $\bar{x}_s$. We use $\mathcal{CF}$ to denote the set of all possible cost functions. Let us see an example.

Example 1. Figure 1 shows a Java program which we use as running example. It is interesting because it shows the different complexity orders that can be obtained by a cost analyzer. We analyze this program using the COSTA system, and selecting the number of executed *bytecode* instructions as cost model. Each Java instruction is compiled to possibly several corresponding bytecode instructions but, since this is not a concern of this paper, we will skip explanations about the constants in the upper bound function and refer to [2] for details.

Given the context constraint $\{n > 0\}$, the COSTA system outputs the *upper bound* cost function for method m which is shown at the bottom of the figure. Since m contains two recursive calls, the complexity is exponential on n, namely we have a factor $2^{\mathsf{nat}(n)}$. At each recursive call, the method f is invoked and its cost (plus a constant value) is multiplied by $2^{\mathsf{nat}(n)}$. In the code of f, we can observe that the while loop has a logarithmic complexity because the loop

```
void m(int n, int a, int b) {
  if (n > 0) {
    m(n - 1, a, b);
    m(n - 2, a, b);
    f(a, b, n);
  }
  }
```

```
void f(int a, int b, int n) {
  int acc = 0;
  while (n > 0) {
    n = n/2;  acc++;
  }
  for (int i = 0; i < a; i++)
    for (int j = 0; j < b; j++) acc++;
}
```

Upper Bound Cost Function

$$m(n,a,b) = 2^{\mathsf{nat}(n)} \underbrace{*(31+ \underbrace{\underbrace{(8*\log(1+\mathsf{nat}(2*n-1))}_{\text{while loop}} + \underbrace{\mathsf{nat}(a)*(10+6*\mathsf{nat}(b))}_{\text{nested loop}})}_{\text{cost of f}})}_{\text{cost of recursive calls}} + \underbrace{3*2^{\mathsf{nat}(n)}}_{\text{base cases}}$$

Fig. 1. Running example and upper bound obtained by COSTA on the number of executed bytecode instructions.

counter is divided by 2 at each iteration. This cost is accumulated with the cost of the second nested loop, which has a quadratic complexity Finally, the cost introduced by the base cases of m is exponential since, due to the double recursion, there is an exponential number of computations which correspond to base cases. Each such computation requires a maximum of 3 instructions.

The most relevant point in the upper bound is that all variables are wrapped by nat in order to capture that the corresponding cost becomes zero when the expression inside the nat takes a negative value. In the case of $\mathsf{nat}(n)$, the nat is redundant since thanks to the context constraint we know that $n > 0$. However, it is required for variables a and b since, when they take a negative value, the corresponding loops are not executed and thus their costs have to become zero in the formula. Essentially, the use of nat allows having a compact cost function instead of one defined by multiple cases. Some cost analyzers generate cost functions which contain expressions of the form $max(\{Exp, 0\})$, which as mentioned above is equivalent to $\mathsf{nat}(Exp)$. We prefer to keep the max operator separate from the nat operator since that will simplify their handling later. □

3 Comparison of Cost Functions

In this section, we state the problem of comparing two cost functions represented as cost expressions. As we have seen in Ex. 1, a cost function $\langle e, \varphi \rangle$ for a method m is a single cost expression which approximates the cost of any possible execution of m which is consistent with the context constraints φ. This can be done by means of nat subexpressions which encapsulate conditions on the input data sizes in a single cost expression. Besides, cost functions often contain max subexpressions, e.g., $\langle \mathsf{max}(\{\mathsf{nat}(x) * \mathsf{nat}(z), \mathsf{nat}(y) * \mathsf{nat}(z)\}), \mathit{true}\rangle$ which represent the cost of disjunctive branches in the program (e.g., the first sub-expression might correspond to the cost of a then-branch and the second one the cost of the else-branch of a conditional statement).

Though nat and max expressions allow building cost expressions in a compact format, when comparing cost functions it is useful to *expand* cost expressions into sets of simpler expressions which altogether have the same semantics. This, on one hand, allows handling simpler syntactic expressions and, on the other hand, allows exploiting stronger context constraints. This expansion is performed in two steps. In the first one we eliminate all max expressions. In the second one we eliminate all nat expressions. The following definition transforms a cost function into a set of max-free cost functions which cover all possible costs comprised in the original function. We write $e[a \mapsto b]$ to denote the expression obtained from e by replacing all occurrences of subexpression a with b.

Definition 2 (max-free operator). *Let $\langle e, \varphi\rangle$ be a cost function. We define the* max*-free operator $\tau_{\mathsf{max}} : 2^{\mathcal{CF}} \mapsto 2^{\mathcal{CF}}$ as follows: $\tau_{\mathsf{max}}(M) = (M - \{\langle e, \varphi\rangle\}) \cup \{\langle e[\mathsf{max}(S) \mapsto e'], \varphi\rangle, \langle e[\mathsf{max}(S) \mapsto \mathsf{max}(S'), \varphi\rangle\}$, where $\langle e, \varphi\rangle \in M$ contains a subexpression of the form* $\mathsf{max}(S)$*, $e' \in S$ and $S' = S - \{e'\}$.*

In the above definition, each application of τ_{max} takes care of taking out one element e' inside a max subexpression by creating two non-deterministic cost functions, one with the cost of such element e' and another one with the remaining ones. This process is iteratively repeated until the fixed point is reached and there are no more max subexpressions to be transformed. The result of this operation is a max-free cost function, denoted by $\mathit{fp}_{\mathsf{max}}(M)$. An important observation is that the constraints φ are not modified in this transformation.

Once we have removed all max-subexpressions, the following step consists in removing the nat-subexpressions to make two cases explicit. One case in which the subexpression is positive, hence the nat can be safely removed, and another one in which it is negative or zero, hence the subexpression becomes zero. As notation, we use capital letters to denote fresh variables which replace the nat subexpressions.

Definition 3 (nat-free operator). *Let $\langle e, \varphi\rangle$ be a* max*-free cost function. We define the* nat*-free operator $\tau_{\mathsf{nat}} : 2^{\mathcal{CF}} \mapsto 2^{\mathcal{CF}}$ as follows: $\tau_{\mathsf{nat}}(M) = (M - \{\langle e, \varphi\rangle\}) \cup \{\langle e_i, \varphi_i\rangle \mid \varphi \wedge \varphi_i \text{ is satisfiable }, 1 \leq i \leq 2\}$, where $\langle e, \varphi\rangle \in M$ contains a subexpression* $\mathsf{nat}(l)$*, $\varphi_1 = \varphi \cup \{A = l, A > 0\}$, $\varphi_2 = \varphi \cup \{l \leq 0\}$, with A a fresh variable, and $e_1 = e[\mathsf{nat}(l) \mapsto A]$, $e_2 = e[\mathsf{nat}(l) \mapsto 0]$.*

In contrast to the max elimination transformation, the elimination of nat subexpressions modifies the set of linear constraints by adding the new assignments of fresh variables to linear expressions and the fact that the subexpression is greater than zero or when it becomes zero. The above operator τ_{nat} is applied iteratively until there are new terms to transform. The result of this operation is a nat-free cost function, denoted by $\mathit{fp}_{\mathsf{nat}}(M)$. For instance, for the cost function $\langle \mathsf{nat}(x) * \mathsf{nat}(z{-}1), \{x > 0\}\rangle$, $\mathit{fp}_{\mathsf{nat}}$ returns the set composed of the following nat-free cost functions:

$\langle A * B, \{A = x, A > 0, B = z{-}1, B > 0\}\rangle$ and $\langle A * 0, \{A = x, A > 0, z{-}1 \leq 0\}\rangle$

In the following, given a cost function f, we denote by $\tau(f)$ the set $fp_{\mathsf{nat}}(fp_{\mathsf{max}}(\{f\}))$ and we say that each element in $fp_{\mathsf{nat}}(fp_{\mathsf{max}}(\{f\}))$ is a *flat* cost function.

Example 2. Let us consider the cost function in Ex. 1. Since such cost function contains the context constraint $n{>}0$, then the subexpressions $\mathsf{nat}(n)$ and $\mathsf{nat}(2{*}n{-}1)$ are always positive. By assuming that fp_{nat} replaces $\mathsf{nat}(n)$ by A and $\mathsf{nat}(2{*}n{-}1)$ by B, only those linear constraints containing $\varphi = \{n > 0, A = n, A > 0, B = 2{*}n{-}1, B > 0\}$ are satisfiable (the remaining cases are hence not considered). We obtain the following set of flat functions:

(1) $\langle 2^A{*}(31{+}8{*}\log(1{+}B){+}C{*}(10{+}6{*}D)){+}3{*}2^A, \varphi_1 = \varphi \cup \{C{=}a, C > 0, D{=}b, D{>}0\}\rangle$
(2) $\langle 2^A{*}(31{+}8{*}\log(1{+}B)){+}3{*}2^A, \varphi_2 = \varphi \cup \{a{\leq}0, D{=}b, D{>}0\}\rangle$
(3) $\langle 2^A{*}(31{+}8{*}\log(1{+}B){+}C{*}10{+}3{*}2^A, \varphi_3 = \varphi \cup \{C{=}a, C > 0, b{\leq}0\}\rangle$
(4) $\langle 2^A{*}(31{+}8{*}\log(1{+}B)){+}3{*}2^A, \varphi_4 = \varphi \cup \{a{\leq}0, b{\leq}0\}\rangle$ □

In order to compare cost functions, we start by comparing two flat cost functions in Def. 4 below. Then, in Def. 5 we compare a flat function against a general, i.e., non-flat, one. Finally, Def. 6 allows comparing two general functions.

Definition 4 (smaller flat cost function in context). *Given two flat cost functions $\langle e_1, \varphi_1\rangle$ and $\langle e_2, \varphi_2\rangle$, we say that $\langle e_1, \varphi_1\rangle$ is smaller than or equal to $\langle e_2, \varphi_2\rangle$ in the context of φ_2, written $\langle e_1, \varphi_1\rangle \trianglelefteq \langle e_2, \varphi_2\rangle$, if for all assignments σ such that $\sigma \models \varphi_1 \cup \varphi_2$ it holds that $\sigma(e_1) \leq \sigma(e_2)$.*

Observe that the assignments in the above definition must satisfy the conjunction of the constraints in φ_1 and in φ_2. Hence, it discards the values for which the constraints become incompatible. An important point is that Def. 4 allows comparing pairs of flat functions. However, the result of such comparison is weak in the sense that the comparison is only valid in the context of φ_2. In order to determine that a flat function is smaller than a general function for any context we need to introduce Def. 5 below.

Definition 5 (smaller flat cost function). *Given a flat cost function $\langle e_1, \varphi_1\rangle$ and a (possibly non-flat) cost function $\langle e_2, \varphi_2\rangle$, we say that $\langle e_1, \varphi_1\rangle$ is smaller than or equal to $\langle e_2, \varphi_2\rangle$, written $\langle e_1, \varphi_1\rangle \preceq \langle e_2, \varphi_2\rangle$, if $\varphi_1 \models \varphi_2$ and for all $\langle e_i, \varphi_i\rangle \in \tau(\langle e_2, \varphi_2\rangle)$ it holds that $\langle e_1, \varphi_1\rangle \trianglelefteq \langle e_i, \varphi_i\rangle$.*

Note that Def. 5 above is only valid when the context constraint φ_2 is more general, i.e., less restrictive than φ_1. This is required because in order to prove that a function is smaller than another one it must be so for all assignments which are satisfiable according to φ_1. If the context constraint φ_2 is more restrictive than φ_1 then there are valid input values for $\langle e_1, \varphi_1\rangle$ which are undefined for $\langle e_2, \varphi_2\rangle$. For example, if we want to check whether the flat cost function (1) in Ex. 2 is smaller than another one f which has the context constraint $\{n > 4\}$, the comparison will fail. This is because function f is undefined for the input values $0 < n \leq 4$. This condition is also required in Def. 6 below, which can be used on two general cost functions.

Definition 6 (smaller cost function). *Consider two cost functions $\langle e_1, \varphi_1\rangle$ and $\langle e_2, \varphi_2\rangle$ such that $\varphi_1 \models \varphi_2$. We say that $\langle e_1, \varphi_1\rangle$ is smaller than or equal to $\langle e_2, \varphi_2\rangle$ iff for all $\langle e_1', \varphi_1'\rangle \in \tau(\langle e_1, \varphi_1\rangle)$ it holds that $\langle e_1', \varphi_1'\rangle \preceq \langle e_2, \varphi_2\rangle$.*

In several applications of resource usage analysis, we are not only interested in knowing that a function is smaller than or equal than another. Also, if the comparison fails, it is useful to know which are the pairs of flat functions for which we have not been able to prove them being smaller, together with their context constraints. This can be useful in order to strengthen the context constraint of the left hand side function or to weaken that of the right hand side function.

4 Inclusion of Cost Functions

It is clearly not possible to try all assignments of input variables in order to prove that the comparison holds as required by Def. 4 (and transitively by Defs. 5 and 6). In this section, we aim at defining a practical technique to syntactically check that one flat function is smaller or equal than another one for all valid assignments, i.e., the relation $\trianglelefteq$ of Def. 4. The whole approach is defined over flat cost functions since from it one can use Defs. 5 and 6 to apply our techniques on two general functions.

The idea is to first *normalize* cost functions so that they become easier to compare by removing parenthesis, grouping identical terms together, etc. Then, we define a series of *inclusion schemas* which provide sufficient conditions to syntactically detect that a given expression is smaller or equal than another one. An important feature of our approach is that when expressions are syntactically compared we compute an approximated difference (denoted adiff) of the comparison, which is the subexpression that has not been required in order to prove the comparison and, thus, can still be used for subsequent comparisons. The whole comparison is presented as a fixed point transformation in which we remove from cost functions those subexpressions for which the comparison has already been proven until the left hand side expression becomes zero, in which case we succeed to prove that it is smaller or equal than the other, or no more transformations can be applied, in which case we fail to prove that it is smaller. Our approach is safe in the sense that whenever we determine that a function is smaller than another one this is actually the case. However, since the approach is obviously approximate, as the problem is undecidable, there are cases where one function is actually smaller than another one, but we fail to prove so.

4.1 Normalization Step

In the sequel, we use the term *basic cost expression* to refer to expressions of the form n, $\log_a(A{+}1)$, A^n, a^l. Furthermore, we use the letter b, possibly subscripted, to refer to such cost expressions.

Definition 7 (normalized cost expression). *A* normalized *cost expression is of the form $\Sigma_{i=1}^{n} e_i$ such that each e_i is a product of basic cost expressions.*

Note that each cost expression as defined above can be normalized by repeatedly applying the distributive property of multiplication over addition in order to get rid of all parentheses in the expression. We also assume that products which are composed of the same basic expressions (modulo constants) are grouped together in a single expression which adds all constants.

Example 3. Let us consider the cost functions in Ex. 2. Normalization results in the following cost functions:

$(1)_n$ $\langle 34{*}2^A{+}8{*}\log_2(1{+}B){*}2^A{+}10{*}C{*}2^A{+}6{*}C{*}D{*}2^A,$
$\quad\varphi_1 = \{A{=}n, A{>}0, B{=}2{*}n{-}1, B{>}0, C{=}a, C > 0, D{=}b, D{>}0\}\rangle$
$(2)_n$ $\langle 34{*}2^A{+}8{*}\log_2(1{+}B){*}2^A,$
$\quad\varphi_2 = \{A{=}n, A{>}0, B{=}2{*}n{-}1, B{>}0, a{\leq}0, D{=}b, D{>}0\}\rangle$
$(3)_n$ $\langle 34{*}2^A{+}8{*}\log_2(1{+}B){*}2^A{+}10{*}C{*}2^A,$
$\quad\varphi_3 = \{A{=}n, A{>}0, B{=}2{*}n{-}1, B{>}0, C{=}a, C > 0, b{\leq}0\}\rangle$
$(4)_n$ $\langle 34{*}2^A{+}8{*}\log_2(1{+}B){*}2^A,$
$\quad\varphi_4 = \{A{=}n, A{>}0, B{=}2{*}n{-}1, B{>}0, a{\leq}0, b{\leq}0\}\rangle$

□

Since $e_1 * e_2$ and $e_2 * e_1$ are equal, it is convenient to view a *product* as the set of its elements (i.e., basic cost expressions). We use $\mathcal{P}_b$ to denote the set of all products (i.e., sets of basic cost expressions) and $\mathcal{M}$ to refer to one product of $\mathcal{P}_b$. Also, since $\mathcal{M}_1 + \mathcal{M}_2$ and $\mathcal{M}_2 + \mathcal{M}_1$ are equal, it is convenient to view the *sum of products* as the set of its elements (its products). We use $\mathcal{P}_\mathcal{M}$ to denote the set of all sums of products and $\mathcal{S}$ to refer to one sum of products of $\mathcal{P}_\mathcal{M}$. Therefore, *a normalized cost expression* is a set of sets of basic cost expressions.

Example 4. For the normalized cost expressions in Ex. 3, we obtain the following set representation:

$(1)_s$ $\langle\{\{34, 2^A\}, \{8, \log_2(1{+}B), 2^A\}, \{10, C, 2^A\}, \{6, C, D, 2^A\}\},$
$\quad\varphi_1 = \{A{=}n, A{>}0, B{=}2{*}n{-}1, B{>}0, C{=}a, C > 0, D{=}b, D{>}0\}\rangle$
$(2)_s$ $\langle\{\{34, 2^A\}, \{8, \log_2(1{+}B), 2^A\}\},$
$\quad\varphi_2 = \{A{=}n, A{>}0, B{=}2{*}n{-}1, B{>}0, a{\leq}0, D{=}b, D{>}0\}\rangle$
$(3)_s$ $\langle\{\{34, 2^A\}, \{8, \log_2(1{+}B), 2^A\}, \{10, C, 2^A\}\},$
$\quad\varphi_3 = \{A{=}n, A{>}0, B{=}2{*}n{-}1, B{>}0, C{=}a, C > 0, b{\leq}0\}\rangle$
$(4)_s$ $\langle\{\{34, 2^A\}, \{8, \log_2(1{+}B), 2^A\}\},$
$\quad\varphi_4 = \{A{=}n, A{>}0, B{=}2{*}n{-}1, B{>}0, a{\leq}0, b{\leq}0\}\rangle$

□

4.2 Product Comparison

We start by providing sufficient conditions which allow proving the $\trianglelefteq$ relation on the basic cost expressions that will be used later to compare products of basic cost expressions. Given two basic cost expressions e_1 and e_2, the third column in Table 1 specifies sufficient, linear conditions under which e_1 is smaller or equal than e_2 in the context of φ (denoted as $e_1 \leq_\varphi e_2$). Since the conditions under which $\leq_\varphi$ holds are over linear expressions, we can rely on existing linear constraint solving techniques to automatically prove them. Let us explain some of entries in the table. E.g., verifying that $A^n \leq m^l$ is equivalent to verifying

e_1	e_2	$e_1 \leq_\varphi e_2$	adiff
n	n'	$n \leq n'$	1
n	$\log_a(A+1)$	$\varphi \models \{a^n \leq A+1\}$	1
n	A^m	$m > 1 \wedge \varphi \models \{n \leq A\}$	A^{m-1}
n	m^l	$m > 1 \wedge \varphi \models \{n \leq l\}$	m^{l-n}
l_1	l_2	$l_2 \notin \mathbb{N}^+$, $\varphi \models \{l_1 \leq l_2\}$	1
l	A^n	$n > 1 \wedge \varphi \models \{l \leq A\}$	A^{n-1}
l	$n^{l'}$	$n > 1 \wedge \varphi \models \{l \leq l'\}$	$n^{l'-l}$
$\log_a(A{+}1)$	l	$l \notin \mathbb{N}^+$, $\varphi \models \{A+1 \leq l\}$	1
$\log_a(A{+}1)$	$\log_b(B{+}1)$	$a \geq b \wedge \varphi \models \{A \leq B\}$	1
$\log_a(A{+}1)$	B^n	$n > 1 \wedge \varphi \models \{A+1 \leq B\}$	B^{n-1}
$\log_a(A{+}1)$	n^l	$n > 1 \wedge \varphi \models \{l > 0, A+1 \leq l\}$	$n^{l-(A+1)}$
A^n	B^m	$n > 1 \wedge m > 1 \wedge n \leq m \wedge \varphi \models \{A \leq B\}$	B^{m-n}
A^n	m^l	$m > 1 \wedge \varphi \models \{n * A \leq l\}$	m^{l-n*A}
n^l	$m^{l'}$	$n \leq m \wedge \varphi \models \{l \leq l'\}$	$m^{l'-l}$

Table 1. Comparison of basic expressions $e_1 \leq_\varphi e_2$

$\log_m(A^n) \leq \log_m(m^l)$, which in turn is equivalent to verifying that $n{*}\log_m(A) \leq l$ when $m > 1$ (i.e., $m \geq 2$ since m is an integer value). Therefore we can verify a stronger condition $n * A \leq l$ which implies $n * \log_m(A) \leq l$, since $\log_m(A) \leq A$ when $m \geq 2$. As another example, in order to verify that $l \leq n^{l'}$, it is enough to verify that $\log_n(l) \leq l'$ when $n > 1$, which can be guaranteed if $l \leq l'$.

The "part" of e_2 which is not required in order to prove the above relation becomes the *approximated difference* of the comparison operation, denoted $\mathsf{adiff}(e_1, e_2)$. An essential idea in our approach is that adiff is a cost expression in our language and hence we can transitively apply our techniques to it. This requires having an approximated difference instead of the exact one. For instance, when we compare $A \leq 2^B$ in the context $\{A \leq B\}$, the approximated difference is 2^{B-A} instead of the exact one $2^B - A$. The advantage is that we do not introduce the subtraction of expressions, since that would prevent us from transitively applying the same techniques.

When we compare two products $\mathcal{M}_1$, $\mathcal{M}_2$ of basic cost expressions in a context constraint φ, the basic idea is to prove the inclusion relation $\leq_\varphi$ for every basic cost expression in $\mathcal{M}_1$ w.r.t. a different element in $\mathcal{M}_2$ and at each step accumulate the difference in $\mathcal{M}_2$ and use it for future comparisons if needed.

Definition 8 (product comparison operator). *Given* $\langle \mathcal{M}_1, \varphi_1 \rangle, \langle \mathcal{M}_2, \varphi_2 \rangle$ *in* $\mathcal{P}_b$ *we define the product comparison operator* $\tau_* : (\mathcal{P}_b, \mathcal{P}_b) \mapsto (\mathcal{P}_b, \mathcal{P}_b)$ *as follows:* $\tau_*(\mathcal{M}_1, \mathcal{M}_2) = (\mathcal{M}_1 - \{e_1\}, \mathcal{M}_2 - \{e_2\} \cup \{\mathsf{adiff}(e_1, e_2)\})$ *where* $e_1 \in \mathcal{M}_1$, $e_2 \in \mathcal{M}_2$, *and* $e_1 \leq_{\varphi_1 \wedge \varphi_2} e_2$.

In order to compare two products, first we apply the above operator τ_* iteratively until there are no more terms to transform. In each iteration we pick e_1 and e_2 and modify $\mathcal{M}_1$ and $\mathcal{M}_2$ accordingly, and then repeat the process on the new sets. The result of this operation is denoted $fp_*(\mathcal{M}_1, \mathcal{M}_2)$. This process is finite because the size of $\mathcal{M}_1$ strictly decreases at each iteration.

Example 5. Let us consider the product $\{8, \log_2(1{+}B), 2^A\}$ which is part of $(1)_s$ in Ex. 4. We want to prove that this product is smaller or equal than the following one $\{7, 2^{3*B}\}$ in the context $\varphi = \{A \le B{-}1, B{\ge}10\}$. This can be done by applying the τ_* operator three times. In the first iteration, since we know by Table 1 that $\log_2(1{+}B) \le_\varphi 2^{3*B}$ and the adiff is 2^{2*B-1}, we obtain the new sets $\{8, 2^A\}$ and $\{7, 2^{2*B-1}\}$. In the second iteration, we can prove that $2^A \le_\varphi 2^{2*B-1}$, and add as adiff $2^{2*B-A-1}$. Finally, it remains to be checked that $8 \le_\varphi 2^{2*B-A-1}$. This problem is reduced to checking that $\varphi \models 8 \le 2{*}B{-}A{-}1$, which it trivially true. $\square$

The following lemma states that if we succeed to transform $\mathcal{M}_1$ into the empty set, then the comparison holds. This is what we have done in the above example.

Lemma 1. *Given $\langle \mathcal{M}_1, \varphi_1\rangle, \langle \mathcal{M}_2, \varphi_2\rangle$ where $\mathcal{M}_1, \mathcal{M}_2 \in \mathcal{P}_b$ and for all $e \in \mathcal{M}_1$ it holds that $\varphi_1 \models e \ge 1$. If $fp_*(\mathcal{M}_1, \mathcal{M}_2) = (\emptyset, \_)$ then $\langle \mathcal{M}_1, \varphi_1\rangle \trianglelefteq \langle \mathcal{M}_2, \varphi_2\rangle$.*

Note that the above operator is non-deterministic due to the (non-deterministic) choice of e_1 and e_2 in Def. 8. Thus, the computation of $fp_*(\mathcal{M}_1, \mathcal{M}_2)$ might not lead directly to $(\emptyset, \_)$. In such case, we can backtrack in order to explore other choices and, in the limit, all of them can be explored until we find one for which the comparison succeeds.

4.3 Comparison of Sums of Products

We now aim at comparing two sums of products by relying on the product comparison of Sec. 4.2. As for the case of basic cost expressions, we are interested in having a notion of approximated adiff when comparing products. The idea is that when we want to prove $k_1{*}A \le k_2{*}B$ and $A \le B$ and k_1 and k_2 are constant factors, we can leave as approximated difference of the product comparison the product $(k_2 - k_1){*}B$, provided $k_2 - k_1$ is greater or equal than zero. As notation, given a product $\mathcal{M}$, we use $\mathsf{constant}(\mathcal{M})$ to denote the constant factor in $\mathcal{M}$, which is equals to n if there is a constant $n \in \mathcal{M}$ with $n \in \mathbb{N}^+$ and, otherwise, it is 1. We use $\mathsf{adiff}(\mathcal{M}_1, \mathcal{M}_2)$ to denote $\mathsf{constant}(\mathcal{M}_2) - \mathsf{constant}(\mathcal{M}_1)$.

Definition 9 (sum comparison operator). *Given $\langle \mathcal{S}_1, \varphi_1\rangle$ and $\langle \mathcal{S}_2, \varphi_2\rangle$, where $\mathcal{S}_1, \mathcal{S}_2 \in \mathcal{P}_\mathcal{M}$, we define the* sum comparison operator $\tau_+ : (\mathcal{P}_\mathcal{M}, \mathcal{P}_\mathcal{M}) \mapsto (\mathcal{P}_\mathcal{M}, \mathcal{P}_\mathcal{M})$ *as follows: $\tau_+(\mathcal{S}_1, \mathcal{S}_2) = (\mathcal{S}_1 - \{\mathcal{M}_1\}, (\mathcal{S}_2 - \{\mathcal{M}_2\}) \cup \mathcal{A})$ iff $fp_*(\mathcal{M}_1, \mathcal{M}_2) = (\emptyset, \_)$ where:*
- *$\mathcal{A} = \{\ \}$ if $\mathsf{adiff}(\mathcal{M}_1, \mathcal{M}_2) \le 0$;*
- *otherwise, $\mathcal{A} = (\mathcal{M}_2 - \{\mathsf{constant}(\mathcal{M}_2)\}) \cup \{\mathsf{adiff}(\mathcal{M}_1, \mathcal{M}_2)\}$.*

In order to compare sums of products, we apply the above operator τ_+ iteratively until there are no more elements to transform. As for the case of products, this process is finite because the size of $\mathcal{S}_1$ strictly decreases in each iteration. The result of this operation is denoted by $fp_+(\mathcal{S}_1, \mathcal{S}_2)$.

Example 6. Let us consider the sum of products $(3)_s$ in Ex. 4 together with $\mathcal{S} = \{\{50, C, 2^B\}, \{9, D^2, 2^B\}\}$ and the context constraint $\varphi = \{1{+}B{\leq}D\}$. We can prove that $(3)_s \trianglelefteq \mathcal{S}$ by applying τ_+ three times as follows:

1. $\tau_+((3)_s, \mathcal{S}) = ((3)_s - \{\{34, 2^A\}\}, \mathcal{S}')$, where $\mathcal{S}' = \{\{16, C, 2^B\}, \{9, D^2, 2^B\}\}$. This application of the operator is feasible since $fp_*(\{34, 2^A\}, \{50, C, 2^B\}) = (\emptyset, \_)$ in the context $\varphi_3 \wedge \varphi$, and the difference constant part of such comparison is 16.
2. Now, we perform one more iteration of τ_+ and obtain as result $\tau_+((3)_s - \{\{34, 2^A\}\}, \mathcal{S}') = ((3)_s - \{\{34, 2^A\}, \{10, C, 2^A\}\}, \mathcal{S}'')$, where $\mathcal{S}'' = \{\{6, C, 2^B\}, \{9, D^2, 2^B\}\}$. Observe that in this case $fp_*(\{10, C, 2^A\}, \{\{16, C, 2^B\}) = (\emptyset, \_)$.
3. Finally, one more iteration of τ_+ on the above sum of products, gives $(\emptyset, \mathcal{S}''')$ as result, where $\mathcal{S}''' = \{\{6, C, 2^B\}, \{1, D^2, 2^B\}\}$.

In this last iteration we have used the fact that $\{1{+}B{\leq}D\} \in \varphi$ in order to prove that $fp_*(\{8, \log_2(1{+}B), 2^A\}, \{9, D^2, 2^B\}) = (\emptyset, \_)$ within the context $\varphi_3 \wedge \varphi$. $\square$

Theorem 1. *Let $\langle \mathcal{S}_1, \varphi_1 \rangle$, $\langle \mathcal{S}_2, \varphi_2 \rangle$ be two sum of products such that for all $\mathcal{M} \in \mathcal{S}_1$, $e \in \mathcal{M}$ it holds that $\varphi_1 \models e \geq 1$. If $fp_+(\mathcal{S}_1, \mathcal{S}_2) = (\emptyset, \_)$ then $\langle \mathcal{S}_1, \varphi_1 \rangle \trianglelefteq \langle \mathcal{S}_2, \varphi_2 \rangle$.*

Example 7. For the sum of products in Ex. 6, we get $fp_+((3)_s, \mathcal{S}) = (\emptyset, \mathcal{S}''')$. Thus, according to the above theorem, it holds that $\langle (3)_s, \varphi_3 \rangle \trianglelefteq \langle \mathcal{S}, \varphi \rangle$. $\square$

4.4 Composite Comparison of Sums of Products

Clearly the previous schema for comparing sums of products is not complete. There are cases like the comparison of $\{\{A^3\}, \{A^2\}, \{A\}\}$ w.r.t. $\{\{A^6\}\}$ within the context constraint $A > 1$ which cannot be proven by using a one-to-one comparison of products. This is because a single product comparison would consume the whole expression A^6. We try to cover more cases by providing a *composite* comparison schema which establishes when a single product is greater than the addition of several products.

Definition 10 (sum-product comparison operator). *Consider $\langle \mathcal{S}_1, \varphi_1 \rangle$ and $\langle \mathcal{M}_2, \varphi_2 \rangle$, where $\mathcal{S}_1 \in \mathcal{P}_\mathcal{M}$, $\mathcal{M}_2 \in \mathcal{P}_b$ and for all $\mathcal{M} \in \mathcal{S}_1$ it holds that $\varphi_1 \models \mathcal{M} > 1$. Then, we define the* sum-product comparison operator $\tau_{(+,*)} : (\mathcal{P}_\mathcal{M}, \mathcal{P}_b) \mapsto (\mathcal{P}_\mathcal{M}, \mathcal{P}_b)$ *as follows: $\tau_{(+,*)}(\mathcal{S}_1, \mathcal{M}_2) = (\mathcal{S}_1 - \{\mathcal{M}_2'\}, \mathcal{M}_2'')$, where $fp_*(\mathcal{M}_2', \mathcal{M}_2) = (\emptyset, \mathcal{M}_2'')$.*

The above operator $\tau_{(+,*)}$ is applied while there are new terms to transform. Note that the process is finite since the size of S_1 is always decreasing. We denote by $fp_{(+,*)}(\mathcal{S}_1, \mathcal{M}_2)$ the result of iteratively applying $\tau_{(+,*)}$.

Example 8. By using the sum-product operator we can transform the pair $(\{\{A^3\}, \{A^2\}, \{A\}\}, \{A^6\})$ into $(\emptyset, \emptyset)$ in the context constraint $\varphi = \{A > 1\}$. To this end, we apply $\tau_{(+,*)}$ three times. In the first iteration, $fp_*(\{A^3\}, \{A^6\}) = (\emptyset, \{A^3\})$. In the second iteration, $fp_*(\{A^2\}, \{A^3\}) = (\emptyset, \{A\})$. Finally in the third iteration $fp_*(\{A\}, \{A\}) = (\emptyset, \emptyset)$. $\square$

When using the sum-product comparison operator to compare sums of products, we can take advantage of having an approximated difference similar to the one defined in Sec. 4.3. In particular, we define the approximated difference of comparing $\mathcal{S}$ and $\mathcal{M}$, written $\mathsf{adiff}(\mathcal{S},\mathcal{M})$, as $\mathsf{constant}(\mathcal{M}) - \mathsf{constant}(\mathcal{S})$, where $\mathsf{constant}(\mathcal{S}) = \sum_{\mathcal{M}' \in \mathcal{S}} \mathsf{constant}(\mathcal{M}')$. Thus, if we compare $\{\{A^3\},\{A^2\},\{A\}\}$ is smaller or equal than $\{4, A^6\}$, we can have as approximated difference $\{A^6\}$, which is useful to continue comparing further summands. As notation, we use $\mathcal{P}_\mathcal{S}$ to denote the set of all sums of products and $\mathcal{S}_s$ to refer one element.

Definition 11 (general sum comparison operator). *Let us consider* $\langle \mathcal{S}_s, \varphi \rangle$ *and* $\langle \mathcal{S}_2, \varphi' \rangle$*, where* $\mathcal{S}_s \in \mathcal{P}_\mathcal{S}$ *and* $\mathcal{S}_2 \in \mathcal{P}_\mathcal{M}$*. We define the general sum comparison operator* $\mu_+ : (\mathcal{P}_\mathcal{S}, \mathcal{P}_\mathcal{M}) \mapsto (\mathcal{P}_\mathcal{S}, \mathcal{P}_\mathcal{M})$ *as follows:* $\mu_+(\mathcal{S}_s, \mathcal{S}_2) = (\mathcal{S}_s - \{\mathcal{S}_1\}, (\mathcal{S}_2 - \{\mathcal{M}\}) \cup \mathcal{A})$*, where* $fp_{(+,*)}(\mathcal{S}_1, \mathcal{M}) = (\emptyset, \_)$ *and* $\mathcal{A} = \{\ \}$ *if* $\mathsf{adiff}(\mathcal{S}_1, \mathcal{M}) \leq 0$*; otherwise* $\mathcal{A} = (\mathcal{M} - \{\mathsf{constant}(\mathcal{M})\}) \cup \{\mathsf{adiff}(\mathcal{S}_1, \mathcal{M})\}$.

Similarly as we have done in definitions above, the above operator μ_+ is applied iteratively while there are new terms to transform. Since the cardinality of $\mathcal{S}_s$ decreases in each step the process is finite. We denote by $fp^g_+(\mathcal{S}_s, \mathcal{S}_2)$ to the result of applying the above iterator until there are no sets to transform.

Observe that the above operator does not replace the previous sum comparator operator in Def. 9 since it sometimes can be of less applicability since $fp_{(+,*)}$ requires that all elements in the addition are strictly greater than one. Instead, it is used in combination with Def. 9 so that when we fail to prove the comparison by using the one-to-one comparison we attempt with the sum-product comparison operator above.

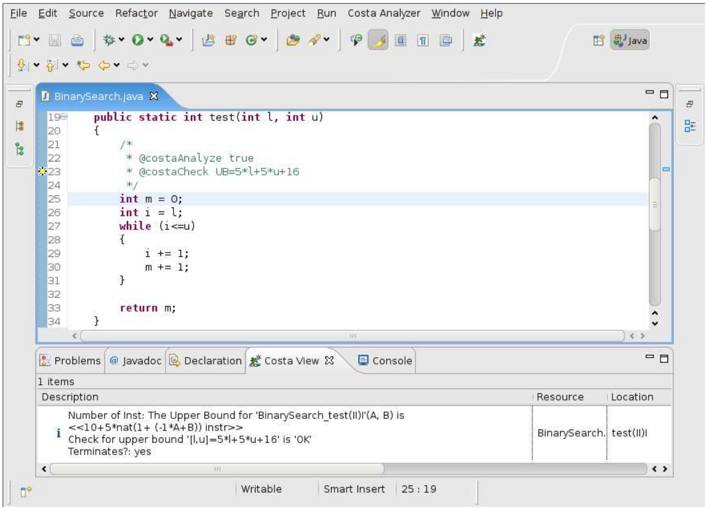
In order to apply the general sum comparison operator, we seek for partitions in the original $\mathcal{S}$ which meet the conditions in the definition above.

Theorem 2 (composite inclusion). *Let* $\langle \mathcal{S}_1, \varphi_1 \rangle$*,* $\langle \mathcal{S}_2, \varphi_2 \rangle$ *be two sum of products such that for all* $\mathcal{M}' \in \mathcal{S}_1$*,* $e \in \mathcal{M}'$ *it holds* $\varphi_1 \models e > 1$*. Let* $\mathcal{S}_s$ *be a partition of* $\mathcal{S}_1$*. If* $fp^g_+(\mathcal{S}_s, \mathcal{S}_2) = (\emptyset, \_)$ *then* $\langle \mathcal{S}_1, \varphi_1 \rangle \trianglelefteq \langle \mathcal{S}_2, \varphi_2 \rangle$.

5 Implementation and Experimental Evaluation

We have implemented our technique and it can be used as a back-end of existing non-asymptotic cost analyzers for average, lower and upper bounds (e.g., [8,2,10,4,5]), and regardless of whether it is based on the approach to cost analysis of [12] or any other. Currently, it is integrated within the COSTA System, and it can be tried out through its web interface which is available from `http://costa.ls.fi.upm.es`.

We first illustrate the application of our method in resource usage verification by showing the working mode of COSTA through its Eclipse plugin. Figure 2 shows a method which has been annotated to be analyzed (indicated by the annotation `@costaAnalyze true`) and its resulting upper bound compared against the cost function written in the assertion `@costaCheck`. The output of COSTA is shown in the *Costa view* (bottom side of the Figure). There, the upper bound



```java
public static int test(int l, int u)
{
    /*
     * @costaAnalyze true
     * @costaCheck UB=5*l+5*u+16
     */
    int m = 0;
    int i = l;
    while (i<=u)
    {
        i += 1;
        m += 1;
    }

    return m;
}
```

1 items

Description	Resource	Location
Number of Inst: The Upper Bound for 'BinarySearch_test(II)I'(A, B) is <<10+5*nat(1+ (-1*A+B)) instr>> Check for upper bound '[l,u]=5*l+5*u+16' is 'OK' Terminates?: yes	BinarySearch.	test(II)I

Fig. 2. Screenshot of the COSTA plugin for Eclipse, showing how annotations are used to interact with COSTA

inferred by COSTA is displayed, together with the result of the comparison with the user's assertion. Besides, the verification of the upper bound is shown in the same line where the annotation is as a marker in the left side of the editor. If the verification fails, a warning marker is shown, instead of the star-like marker of Figure 2. Thus, by annotating the methods of interest with candidate upper bounds, it is possible to verify the resource usage of such methods, and to mark those methods that do not meet their resource usage specification.

In Table 2, we have performed some experiments which aim at providing some information about the accuracy and the efficiency of our technique. The first seven benchmark programs correspond to examples taken from the JOlden benchmark suite [11], the next two ones from the experiments in [1] and the last one is our running example. COSTA infers the upper bound cost functions for them which are shown in the second column of the table. All execution times shown are in milliseconds and have been computed as the average time of ten executions. The environment were the experiments were run was Intel Core2 Duo 1.20 GHz with 2 processors, and 4 GB of RAM.

The first column is the name of the benchmark. The second column is the expresion of the cost function. The next two columns show the time taken by our implementation of the comparison approach presented in the paper in two different experiments which we describe below. The next two columns include the term size of the cost function inferred by COSTA and normalized as explained in Section 4, and the term size of the product of the cost function by itself. The next two columns include the ratio between size and time; those are estimations of the number of terms processed by milisecond in the comparison. We use CF to refer to the cost function computed by COSTA.

Bench.	**Cost Function**	T_1	T_2	**Size**$_1$	**Size**$_2$	$Size/T_1$	$Size/T_2$
`bH`	$128 + 96 * \mathsf{nat}(x)$	0	0.2	6	11	N/A	N/A
`treeAdd`	$4 + (4 * \mathsf{nat}(x) + 1) + 40 * 2^{\mathsf{nat}(y-1)}$	8	18	11	41	1.40	2.28
`biSort`	$16 + (4 * \mathsf{nat}(x) + 1) * \mathsf{nat}(y - 1)$	15	39	9	33	0.60	0.85
`health`	$28 * (4^{\mathsf{nat}(x-1)} - 1)/3 + 28 * 4^{\mathsf{nat}(x-1)}$	7	23	21	115	3.00	5.00
`voronoi`	$20 * \mathsf{nat}(2 * x - 1)$	2	5	3	5	1.50	1.00
`mst`	$\mathsf{max}(12 + 4 * \mathsf{nat}(1 + x)$ $+\mathsf{nat}(1 + x) * (20 + 4 * \mathsf{nat}(1/4 * x))$ $+16 * \mathsf{nat}(1 + x) * \mathsf{nat}(1 + x) + 8 * \mathsf{nat}(1 + x),$ $4 + \mathsf{max}(16 + 4 * \mathsf{nat}(1 + x)$ $+\mathsf{nat}(1 + x) * (20 + 4 * \mathsf{nat}(1/4 * x))$ $+16 * \mathsf{nat}(1 + x) * \mathsf{nat}(1 + x) + 16 * \mathsf{nat}(1 + x),$ $20 + 4 * \mathsf{nat}(1 + x)+$ $+\mathsf{nat}(1 + x) * (20 + 4 * \mathsf{nat}(1/4 * x))+$ $4 * \mathsf{nat}(1/4 * x)))$	96	222	49	241	0.51	1.09
`em3d`	$93 + 4 * \mathsf{nat}(t) + 4 * \mathsf{nat}(y)+$ $\mathsf{nat}(t - 1) * (28 + 4 * \mathsf{nat}(y)) + 4 * \mathsf{nat}(t)+$ $4 * \mathsf{nat}(y) + \mathsf{nat}(t - 1) * (28 + 4 * \mathsf{nat}(y))+$ $4 * \mathsf{nat}(y)$	54	113	19	117	0.35	1.04
`multiply`	$9 + \mathsf{nat}(x) * (16 + 8 * \log_2(1 + \mathsf{nat}(2 * x - 3)))$	10	24	14	55	1.40	2.29
`evenDigits`	$49 + (\mathsf{nat}(z) * (37 + (\mathsf{nat}(y) * (32 + 27 * \mathsf{nat}(y))$ $+27 * \mathsf{nat}(y))) + \mathsf{nat}(y) * (32 + 27 * \mathsf{nat}(y))$ $+27 * \mathsf{nat}(y))$	36	94	29	195	0.81	2.07
`running`	$2^{\mathsf{nat}(x)} * (31 + (8 * \log_2(1 + \mathsf{nat}(2 * x - 1))+$ $+\mathsf{nat}(y) * (10 + 6 * \mathsf{nat}(z)))) + 3 * 2^{\mathsf{nat}(x)}$	40	165	34	212	0.85	1.28

Table 2. Experiments in Cost Function Comparison

T_1 Time taken by the comparison $CF \preceq rev(CF)$, where $rev(CF)$ is just the reversed version of CF. I.e., $rev(x + y + 1) = 1 + x + y$. The size of the expressions involved in the comparison is shown in the fifth column of the table (Size$_1$).

T_2 Time taken by the comparison $CF + CF \preceq CF * CF$, assuming that CF takes at least the value 2 for all input values. In this case, the size of the expression grows considerably and hence the comparison takes a longer time than the previous case. The size of the largest expression in this case is shown in the sixth column of the table (Size$_2$).

In all cases, we have succeeded to prove that the comparison holds. Ignoring the first benchmark, that took a negligible time, the ratio between size and time and falls in a narrow interval (1 or 2 terms processed by milisecond). Interestingly, for each one of the benchmarks (except `voronoi`), that ratio increases with term size, implying that the number of terms processed by milisecond is higher in more complex expressions. However, these performance measurements should be verified with a larger number of case studies, to verify how it varies with the size of the input. We leave that task as further work. In any case, we believe that our preliminary experiments indicate that our approach is sufficiently precise in practice and that the comparison times are acceptable.

6 Other Approaches and Related Work

In this section, we discuss other possible approaches to handle the problem of comparing cost functions. In [7], an approach for inferring non-linear invariants using a linear constraints domain (such as polyhedra) has been introduced. The idea is based on a *saturation* operator, which lifts linear constraints to non-linear ones. For example, the constraint $\Sigma a_i x_i = a$ would impose the constraint $\Sigma a_i Z_{x_i u} = au$ for each variable u. Here $Z_{x_i u}$ is a new variable which corresponds to the multiplication of x_i by u. This technique can be used to compare cost functions, the idea is to start by saturating the constraints and, at the same time, converting the expressions to linear expressions until we can use a linear domain to perform the comparison. For example, when we introduce a variable $Z_{x_i u}$, all occurrences of $x_i u$ in the expressions are replaced by $Z_{x_i u}$. Let us see an example where: in the first step we have the two cost functions to compare; in the second step, we replace the exponential with a fresh variable and add the corresponding constraints; in the third step, we replace the product by another fresh variable and saturate the constraints:

$$\begin{array}{l|l} w \cdot 2^x \geq 2^y & \{x \geq 0, x \geq y, w \geq 0\} \\ w \cdot Z_{2^x} \geq Z_{2^y} & \{x \geq 0, x \geq y, Z_{2^x} \geq Z_{2^y}\} \\ Z_{w \cdot 2^x} \geq Z_{2^y} & \{x \geq 0, x \geq y, Z_{2^x} \geq Z_{2^y}, Z_{w \cdot 2^x} \geq Z_{2^y}\} \end{array}$$

Now, by using a linear constraint domain, the comparison can be proved. We believe that the saturation operation is very expensive compared to our technique while it does not seem to add significant precision.

Another approach for checking that $e_1 \preceq e_2$ in the context of a given context constraint φ is to encode the comparison $e_1 \preceq e_2$ as a Boolean formula that simulates the behavior of the underlying machine architecture. The unsatisfiability of the Boolean formula can be checked using SAT solvers and implies that $e_1 \preceq e_2$. The drawback of this approach is that it requires fixing a maximum number of bits for representing the value of each variable in e_i and the values of intermediate calculations. Therefore, the result is guaranteed to be sound only for the range of numbers that can be represented using such bits. On the positive side, the approach is complete for this range. In the case of variables that correspond to integer program variables, the maximum number of bits can be easily derived from the one of the underlying architecture. Thus, we expect the method to be precise. However, in the case of variables that correspond to the size of data-structures, the maximum number of bits is more difficult to estimate.

Another approach for this problem is based on numerical methods since our problem is analogous to proving whether $0 \preceq b - f_m$ in the context ϕ_b. There are at least two numerical approaches to this problem. The first one is to find the roots of $b - f_m$, and check whether those roots satisfy the constraints ϕ_b. If they do not, a single point check is enough to solve the problem. This is because, if the equation is verified at one point, the expressions are continuous, and there is no sign change since the roots are outside the region defined by ϕ_b, then we can ensure that the equation holds for all possible values satisfying ϕ_b. However, the problem of finding the roots with multiple variables is hard in general and often not solvable. The second approach is based on the observation that there is no need to compute the actual values of the roots. It is enough to know whether there are roots in the region defined by ϕ_b. This can be done by finding the minimum values of expression $b - f_m$, a problem that is more affordable using numerical methods [9] . If the minimum values in the region defined by ϕ_b are greater than zero, then there are no roots in that region. Even if those minimum values are out of the region defined by ϕ_b or smaller than zero, it is not necessary to continue trying to find their values. If the algorithm starts to converge to values out of the region of interest, the comparison can be proven to be false. One of the open issues about using numerical methods to solve our problem is whether or not they will be able to handle cost functions output from realistic programs and their performance. We have not explored these issues yet and they remain as subject of future work.

7 Conclusions

In conclusion, we have proposed a novel approach to comparing cost functions which is relatively efficient and powerful enough for performing useful comparisons of cost functions. Making such comparisons automatically and efficiently is essential for any application of automatic cost analysis. Our approach could be combined with more heavyweight techniques, such as those based on numerical methods, in those cases where our approach is not sufficiently precise.

References

1. E. Albert, P. Arenas, S. Genaim, and G. Puebla. Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis. In *SAS*, LNCS 5079, 2008.
2. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In *ESOP*, LNCS 4421, pages 157–172. Springer, 2007.
3. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. COSTA: Design and Implementation of a Cost and Termination Analyzer for Java Bytecode. In *Post-proceedings of Formal Methods for Components and Objects (FMCO'07)*, number 5382 in LNCS, pages 113–133. Springer-Verlag, October 2008.
4. V. Braberman, F. Fernández, D. Garbervetsky, and S. Yovine. Parametric Prediction of Heap Memory Requirements. In *ISMM*. ACM Press, 2008.
5. W-N. Chin, H.H. Nguyen, C. Popeea, and S. Qin. Analysing Memory Resource Bounds for Low-Level Programs. In *ISMM*. ACM Press, 2008.
6. P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL'77*, pages 238–252. ACM, 1977.
7. B. S. Gulavani and S. Gulwani. A Numerical Abstract Domain Based on Expression Abstraction and Max Operator with Application in Timing Analysis. In *CAV*, LNCS 5123, pages 370–384. Springer, 2008.
8. S. Gulwani, K. K. Mehra, and T. M. Chilimbi. Speed: precise and efficient static estimation of program computational complexity. In *POPL*, pages 127–139. ACM, 2009.
9. S. Kirkpatrick, Jr. C.D. Gelatt, and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.
10. J. Navas, M. Méndez-Lojo, and M. Hermenegildo. User-Definable Resource Usage Bounds Analysis for Java Bytecode. In *BYTECODE*. Elsevier, 2009.
11. JOlden Suite. `http://www-ali.cs.umass.edu/DaCapo/benchmarks.html`.
12. B. Wegbreit. Mechanical Program Analysis. *Comm. of the ACM*, 18(9), 1975.

Characterising Effective Resource Analyses for Parallel and Distributed Coordination

P. W. Trinder[1], M. I. Cole[2], H-W. Loidl[3], and G. J. Michaelson[1]

[1] School of Mathematical and Computer Sciences,
Heriot-Watt University, Riccarton, Edinburgh EH14 4AS, U.K.
{trinder,greg}@macs.hw.ac.uk
[2] School of Informatics
The University of Edinburgh, 10 Crichton Street Edinburgh, EH8 9AB
mic@inf.ed.ac.uk
[3] Ludwig-Maximilians-Universität München,
Institut für Informatik, D 80538 München, Germany,
hwloidl@informatik.uni-muenchen.de

Abstract. An important application of resource analysis is to improve the performance of parallel and distributed programs. In this context key resources are time, space and communication. Given the spectrum of cost models and associated analysis techniques, what combination should be selected for a specific parallel or distributed context?
We address the question as follows. We outline a continuum of coordination cost models and a range of analysis techniques. We consider six representative parallel/distributed applications of resource analysis techniques, and aim to extract general principles governing why the combination of techniques is effective in its context.

1 Introduction

Parallel and distributed programs must specify both the computation to be performed, and how this is to be *coordinated* across multiple locations. Effective resource analyses enable better coordination, for example scheduling can be improved with accurate estimates of the computational cost for units of work. The resource analyses need to build on realistic cost models to reflect the resource consumption incurred during execution. Furthermore, an appropriate analysis technique must be used to predict resource consumption to the required accuracy. Finally, there are many possible uses of such resource information.

Section 2 classifies the cost models, focusing on the level of abstraction over the hardware that is provided. The PRAM model is extremely simple and abstract. More refined models use a fixed structure of the execution of the code to produce an accurate cost model: Bulk Synchronous Processes (BSP) is one such example. Finally, there is a rich class of models that take hardware details such as caches and processor pipelines into account to produce a very accurate model: for example the processor model used by AbsInt's aiT analysis.

Section 3 classifies the analysis techniques. We start by outlining several representative systems that use resource bounds to improve coordination: design-time cost analysis through the use of structured program notations such as the Bird-Meertens-Formalism (BMF) or Bulk Synchronous Processes (BSP); compile-time cost analysis through the use of type inference, abstract interpretation, or constraint system solving; run-time cost analysis through the (abstract) execution of (abstracted) input based on a costed semantics.

Section 4 outlines six applications of resource analysis techniques in a parallel/distributed context. The applications are selected to be effective, i.e. they improve parallel/distributed coordination, and representative. The applications are representative in utilising a range of analyses from the most abstract to the most concrete, and in using the resource information for a range of coordination purposes including resource-safe execution, compiler optimisations, optimising parallel execution and enabling mobility.

Section 5 investigates in the impact of the choices made for the cost model and analysis technique on the concrete application domain. Section 6 summarises the general principles governing why combinations of cost model and analysis technique are effective, and speculates on future trends for resource analysis in the parallel/distributed arena.

2 Coordination Cost Models

Coordination cost models provide tractable abstractions of the performance of real parallel or distributed systems. They cover a well populated continuum from the most simple and abstract, through to the highly detailed and concrete. For our purposes we note three well-known examples that are used in the analysis methodologies presented subsequently. As we shall see in Section 4, classical sequential cost models are also useful, e.g. using predicted execution time for tasks to inform scheduling.

2.1 PRAM

The Parallel Random Access Machine (PRAM) model [17] is the most abstract parallel cost model. PRAM is the fundamental parallel machine and cost model within the parallel algorithms and complexity research community. In its simplest form, it models stepwise synchronous, but otherwise unrestricted access to a shared memory by a set of conventional sequential processors. At each synchronous step, each processor performs one operation from a simple, conventional instruction set. Each such step is costed at unit time, irrespective of the operations involved, and in particular, irrespective of which shared memory locations are accessed. It therefore ignores more practical issues such as contention, memory hierarchy, underlying communication infrastructure and all processor internal issues. Nevertheless, it has provided a durable and sound basis for at least the initial phases of asymptotically analysed parallel algorithm design. A plethora of variants aim to introduce more pragmatic cost issues.

2.2 BSP

The Bulk Synchronous Parallel (BSP) model [7] occupies a less abstract position in the cost model spectrum than PRAM. In contrast to PRAM it recognises that synchronisation is not free, that sharing of data involves communication (whether explicitly or implicitly), and that the cost of this communication, both absolutely and relative to that of processor-local computation can be highly machine dependent. To tame this complexity BSP introduces a constrained operational model. Thus, a BSP computer consists of processors connected by a communication network. Each processor has a fast local memory, and may follow its own thread of computation. A BSP computation proceeds in a series of *supersteps* comprising three stages:

- Independent concurrent computation on each processor using only local values.
- Communication: in which each processor exchanges data with every other processor.
- Barrier synchronisation: where all processes wait until all other processes have finished their communication actions.

The BSP cost model has two parts: one to estimate the cost of a superstep, and another to estimate the cost of the program as the sum of the costs of the supersteps. The cost of a superstep is the sum of the cost of the longest running local computation, the cost of the maximum communication between the processes, and the cost of the barrier synchronisation. The costs are computed in terms of four abstract parameters which respectively model the number of processors, the cost of global synchronisation, the global bandwidth of the network and the raw computational speed of the processors. The constrained computation model allows BSP implementations to provide a benchmark suite which derives concrete, machine-specific values for the four BSP parameters. These can then be inserted into the abstract (architecture independent) cost already derived for a given program, to predict its true performance.

While BSP makes no attempt to account for processor internals or memory hierarchy (other than indirectly through benchmarking) or specific communication patterns (indeed, classical BSP implementations rely on randomisation to deliberately *obliterate* patterns in the interests of predictability), a considerable literature testifies to the pragmatic success of the approach [7].

2.3 Cost Semantics

Cost semantics occupy a more concrete position in the cost model spectrum than BSP. Cost semantics use a non-standard semantics with a cost assigned to primitive operations, and language constructs combine the costs of their subterms, e.g. the cost of a multiplication might be 1 plus the cost of evaluating the left and right operands. The costs of coordination operations like communication latency, or synchronisation, are accounted as some architecture dependent value.

Concrete cost semantics aim to provide precise cost estimates by assigning accurate costs to terms, often based on careful profiling of a target architecture [9]. In contrast an *abstract* cost semantics assigns unit costs to terms and hence is both simpler and architecture independent.

2.4 Accurate Hardware Models

Accurate hardware models occupy the most concrete position in the cost model spectrum. These models provide precise cost information of low-level code, for example by providing time information in clock cycles for each machine instruction. Such level of detail is required for industry strength worst-case execution time (WCET) analyses. These analyses must be safe in the sense of always producing upper bounds. They also have to be precise to be of any practical use. One example of a WCET analysis that combines these features is AbsInt's aiT tool [16]. It is based on abstract interpretation, operates on machine code for a range of embedded systems processors, and produces WCET bounds. In order to obtain accurate bounds, the analysis models details of the hardware architecture, in particular the cache behaviour and the possible pipeline structure of the processor. Thus, the accurate cost model is complemented by an accurate hardware model at the analysis stage.

3 Resource Analyses

3.1 Design Time Analysis

Abstract cost models based around PRAM, BSP and Bird-Meertens-Formalism (BMF) enable the programmer to reason about costs during program design. The models often require that the program is expressed using a specific structure, e.g. as a sequence of supersteps for BSP analysis. A significant advantage of these techniques is that, guided by the model, the programmer can relatively cheaply transform the program design to reduce the consumption of a specific resource, before committing to an implementation.

3.2 Compile Time Analysis

In the area of compile time analyses many techniques have been developed to statically infer information of the behaviour of the program at runtime. The best known techniques are type inference, abstract interpretation, and constraint system solving, and they may be used in combination.

Type Inference: Based on the observation that type inference can be separated into two phases, collecting constraints on type/resource variables and solving these constraints [28], several type-inference based analyses have been developed that extend Hindley-Milner type inference to collect constraints on resources.

Abstract Interpretation: Abstract interpretation [12] defines an abstract domain of values, which is typically very small and is often used to provide only qualitative information. For example in strictness analysis the interesting distinction is only whether an expression is strict or not. By using a richer abstract domain quantitative information can be modelled, too. Functions are mapped to abstracted functions that operate over the abstract domain. The analysis then proceeds by executing these abstract functions, and in particular finding fixpoints for recursive functions. Many practically useful techniques have been developed for this process, and therefore well-developed inference engines exists that can be used for cost analysis.

Constraint System Solving: This approach is related to the type inference approach. In the former, constraints are collected during type inference and then solved separately. In the (pure) constraint system solving approach the collection of constraints is not tied to type inference. An example of this approach is control flow analysis [36].

3.3 Run-Time Analysis

Run-time cost analysis typically entails the abstract execution with some abstracted input. It differs from profiling in that the execution and resources are abstract rather than real. It is often used in conjunction with a static resource analysis, e.g. to approximate the sizes of key data structures, e.g. [23, 32].

4 Parallel/Distributed Resource Analysis Applications

This section outlines six representative parallel/distributed applications of resource analysis techniques. The applications are ordered from those applying the most abstract analysis (BMF-PRAM) to the most concrete (Type-based Analysis with a Precise Model).

It is well known that performance analysis within conventional programming models is undecidable. Pragmatic progress can be made by relaxing the extent to which we hope for computable analysis for example by requiring oracular insight from the programmer and/or by constraining the programming model in some way.

For each system we outline how resource information is obtained and applied. Each representative model is effective, i.e. the cost information improves coordination and hence performance.

4.1 BMF-PRAM

The Bird-Meertens Formalism (BMF) [5] is a calculus for deriving functional programs from specifications, with an emphasis on the use of bulk operations across collective data-structures such as arrays, lists and trees. While independent of any explicit reference to parallelism, many of its operations clearly admit potentially efficient parallel implementation. A number of projects have attempted to exploit this opportunity by adding parallel cost analyses to BMF inspired programming models.

In [37] Cai and Skillicorn present an informal PRAM based cost model for BMF across list-structured data. Each operation is provided with a cost, parameterised by the costs of the applied operations (for example, the element-wise cost of an operation to be mapped across all elements of a list) and data structure sizes, and rules are provided for composing costs across sequential and concurrent compositions. The paper concludes with a sample program derivation for the *maximum segment sum* problem. In conventional BMF program-calculation style, an initially "obviously" correct but inefficient specification is transformed by the programmer into a much more efficient final form.

In [25], Jay et al. build a formal cost calculus for a small BMF-like language using PRAM as the underlying cost model. In order to aid implementation, the language is further constrained to be *shapely*, meaning that the size of intermediate bulk data-structures can be statically inferred from the size of inputs. The approach is demonstrated by automated application to simple matrix-vector operations.

These approaches can be characterised as being of relatively low accuracy (a property inherited from their PRAM foundation), offering a quite rich, though structurally constrained source language, being entirely static in nature and with varying degrees of formality and support.

4.2 BMF-BSP

Building on Jay and Skillicorn's seminal work, a number of projects have sought to inject more realism into the costing of BMF inspired parallel programming frameworks. The primary vehicle to this end was the substitution of BSP for PRAM as the foundational cost model [25, 20]. In particular, [20] defines and implements a BMF-BSP calculus and compares the accuracy of its predictions with the runtime of equivalent (but hand-translated) BSP programs. Using maximum segment as a case study, the predictions exhibit good accuracy and would lead to the correct decision at each stage of the program derivation.

Meanwhile, in a more informal setting reflecting the approach of [37], [6] reports upon a BSP based, extended BMF derivation of a program for the solution of tridiagonal systems of linear equations. Once again good correlation between (hand generated) predictions and real implementation is reported, with no more than 12% error across a range of problem sizes.

These developments can be characterized as offering enhanced accuracy (and for the first time, experimentally validated), while retaining similarly structured models and support. As a by-product of the use of BSP, analyses are now target architecture specific, once instantiated with the machine's BSP constants, though still static with respect to any particular instance.

4.3 Skeleton-based Approaches

The *skeleton* based approach to parallel programming [11] advocates that commonly occuring patterns of parallel computation and interaction be abstracted as library or language constructs. These may be tied to data-parallel bulk operations, in the style of BMF, or used to capture more task oriented process networks (for example, pipelines). Various projects have sought to tie cost models to skeletons and to use these, either explicitly or implicitly to guide program development and implementation.

For example, based around a simple model of message passing costs, [19] uses meta-programming to build cost equations for a variety of skeleton implementations into an Eden skeleton library, allowing the most appropriate implementation to be chosen at compile-time given instantiation of target machine specific parameters (i.e. in the style of, but distinct in detail from, BSP). Discrimination between four possible variants of a farm skeleton, used to implement a Mandelbrot visualisation is reported.

Meanwhile, [18] describes an attempt to embed the BSP model directly into a functional programming language, ML. At the level of parallelism, the programming model is thus constrained to follow the BSP superstep constraints (which might be viewed as relatively loose skeleton), while computation within a superstep is otherwise unconstrained. Analysis is informal, in the conventional BSP style, but the language itself has a robust parallel and distributed implementation. A reported implementation of an N-body solver once again demonstrates close correlation between predicted and actual execution times.

The approach proposed by [40] presents the programmer with imperative skeletons, each with an associated parallel cost model. The models are defined in a performance enhanced process algebra [22], parameterised by a small number constants derived by running benchmark code fragments. As in [19] models of competing implementation strategies are evaluated and the best selected. In a novel extension, designed to cater for systems in which architectural performance characteristics may vary dynamically, the chosen model is periodically validated against actual performance. Where a significant discrepancy is found, the computation can be halted, re-evaluated and rescheduled.

These approaches are strong in terms of language support, offering essentially a two-layer model in which parallelism is constrained by the available skeleton functions but local computation is free and powerful. The cost foundations are of middling accuracy, sometimes augmented by the use of real code profiling. They employ a range of static and dynamic analysis.

4.4 Using Statically Inferred Granularity Information for Parallel Coordination

This section outlines several systems that apply cost information in the context of parallel computation to decide whether a parallel thread should be generated for some computation. In particular, it should become possible to identify very small pieces of computation, for which the overhead of generating a parallel thread is higher than the actual computation itself. Hence the characteristic feature of the cost information here is that while it must be accurate for small computations, it can be far less accurate for larger computations. Potentially all computations beyond a certain threshold can be mapped to an abstract value of infinity.

Static Dependent Costs: Reistad and Gifford [35] define the notion of static dependent costs for the analysis of a strict, higher-order functional language with imperative features. These costs describe the execution time of a function in terms of the size of its input by attaching cost information to the type of a function. Thereby it becomes possible to propagate cost information from the definition of a function to its use, enabling the static, type-based analysis of higher-order programs. The static inference of cost expressions is combined with runtime calculation that instantiate cost expressions for concrete sizes to gain concrete estimates. Runtime measurements of the system show that their cost estimates are usually within a factor of three of the real costs. This information is used in a dynamic profitability analysis, that compares the cost estimate of an expression with the thread creation overhead, and generates parallelism only if it is profitable. A game of life program, based on a parallel map operation exploiting this profitability analysis, achieved a speedup of more than two compared to a naive version of a parallel map on a four processor SGI shared-memory machine.

Dynamic Granularity Estimation: Another instance of this approach is [23], where a technique of dynamic granularity estimation for strict, list-based, higher-order languages is developed. This technique consists of two components:

– A compile-time (static) component, based on abstract interpretation to identify components whose complexity depends on the size of a data structure.
– A run-time (dynamic) component, for approximating sizes of the data structures at run-time.

Based on the results of the static component, the compiler inserts code for checking the size of parameters at certain points. At runtime the result of these checks determine whether a parallel task is created or not. The dynamic component is implemented on a Sequent Symmetry shared-memory machine on top of a parallel SML/NJ implementation. It is stated that the runtime overhead for keeping track of approximations (one additional word per cons cell) is very low. For the quicksort example an efficiency improvement of 23% has been reported.

Sized Time Systems: The sized time system in [30] develops a type-based inference of execution time and combines it with sized types [24], a static type system for inferring bounds on the size of data structures. Thus, in contrast to the previous systems, no run-time analysis is required. As in the previous systems, the analysis of time information is restricted to non-recursive functions. As traditional, the inference is separated into a phase of collecting constraints, inequalities over natural numbers, and a separate phase of solving these constraints. A simulator for the parallel execution of Haskell programs has been used to implement several scheduling algorithms that make use of granularity information. In its most naive instance all potential parallelism below a fixed granularity threshold is discarded. In a second variant, granularity information is used to always pick the largest item when generating new parallelism. In a final version, granularity information is used by the scheduler to favour the largest thread upon re-scheduling. The results with these three version showed [29][Chapter 5], that the most naive version of a fixed granularity threshold performed best, since the management overhead of the more refined policies dominated the gains in execution time.

In summary, all three *systems* discussed here are based on an abstract, architecture-independent cost model, and use a static, type-based cost analysis to determine size-dependent bounds. Two of the three systems combine these with a very simple run-time analysis, which mainly supplies size information. The *languages* covered are predominantly functional, although the static dependent cost system also covers imperative constructs. The *run-time techniques* that use the provided cost information are very simple: in most cases a binary decision on the profitability of a potential parallel thread is made. Arguably the use of the cost information is *a priori* limited by the choice of an abstract cost model, which cannot provide precise bounds. However, measurements of the system show that even with the abstract cost model, cost predictions, where possible, are reasonably accurate.

4.5 Abstract Cost Semantics: Autonomous Mobile Programs

Autonomous mobile programs (AMPs) periodically use a cost model to decide where to execute in a network [15]. The key decision is whether the predicted time to complete on the current location is greater than the time to communicate to the best available location and complete there.

The AMP cost model is an abstract cost semantics for a core subset of the Jocaml mobile programming language including iterating higher-order functions like `map`. Rather than predicting the time to evaluate a term the model predicts the *continuation* cost of every subterm within a term. This information is used to estimate the time to complete the program from the current point in the execution.

The AMP continuation cost model is generated statically, and is then parameterised dynamically to determine movement behaviour. Key dynamic parameters include the current input size, execution speed on the current location, predicted execution speeds of alternative locations.

In summary the AMP abstract costed operational semantics is applied to a core mobile functional language with higher-order functions. The model is statically generated but dynamically parameterised. While such an abstract cost semantics provides low accuracy, empirical results show that the information adequately informs mobility decisions [15].

4.6 Type-based Analysis (Precise Model): Resource-safe Execution in Hume

The goal of resource-safe execution is to statically guarantee that available resources are never exhausted. This is highly desirable in many contexts, e.g. to provide resource guarantees for mobile code, or in embedded systems where resources are highly constrained.

With multi-core architectures entering the main-stream of computer architectures, embedded system designers are looking into exploiting the parallelism provided on such platforms. Thus, the new challenge is to combine resource-safe execution with a model for parallel execution that can effectively, and safely exploit the parallelism. One aspect to this challenge is to best use the special nature of the resource bounds, required for resource-safe execution, to guide parallel execution.

In order to meet safety requirements on embedded systems, the resource predictions, and hence the cost model, have to be *upper bounds* on the concrete resource consumption rather than simple predictions. These bounds don't necessarily have to be precise, however they must be concrete enough to assure that no concrete evaluation exceeds them. Furthermore, formal guarantees of the validity for these bounds are highly desirable.

The resource analysis for Hume [26], together with the infrastructure for executing Hume code on embedded systems, is an instance of such resource-safe execution. The source language, Hume, has two layers. The box layer defines a network of boxes that communicate along single-buffer one-to-one wires. The expression layer is a *strict, higher-order functional language*. The resource analysis is a *static, type-based analysis*, building on the concept of amortised costs. It produces, where possible, linear bounds on the resource consumption. Some supported resources are heap- and stack-space consumption, and worst case execution time.

The underlying cost model is an accurate hardware model obtained by performing machine-code-level worst-case execution time analysis on the operations of the underlying abstract machine. Thus, it is a *concrete cost model*, taking into account hardware characteristics such as cache and pipeline structure. It is a *safe cost model* in the sense that all costs are upper bounds. The results of the resource analyses for space and time have been validated against measurements obtained on real embedded systems hardware for a range of applications [27].

The Hume compiler currently uses the resource information only in determining sizes of buffers etc, needed to assure resource-safe, single processor execution. In the longer term this information will also be used in other components of the system, for example in the scheduler on the box layer. The decomposition of the program into boxes provides a natural model for parallel execution on multi-core machines. In this context, the number of threads, namely boxes, is statically fixed. The main usage of the resource information is therefore in statically mapping the threads to the available cores and in dynamically deciding which thread to execute next. Since on box layer the execution of a program is an alternating sequence of compute- and communicate-steps, the mapping process is akin to the process of developing a parallel program in a BSP model.

5 Cost Model & Analysis Critique

Given the spectrum of cost models and associated analysis techniques, what combination should be selected for a specific parallel or distributed application? This section investigates why a specific cost model and analysis technique proves effective in the specific parallel/distributed context.

5.1 BMF-PRAM

In common with their PRAM base, the techniques discussed in Section 4.1 are most appropriate in the early phases of algorithm design, rather than detailed program development. The techniques enable the designer to quickly compare coarse performance estimates of alternative approaches. An informal, even asymptotic flavour predominates.

5.2 BMF-BSP

The BMF-BSP approaches discussed in Section 4.2 are more appropriate when a reasonably detailed algorithm already exists, allowing more refined, machine-sensitive cost modelling as a concrete program is refined. They are most appropriate in (indeed, almost constrained to) contexts which provide a BSP library implementation, with its associated benchmark suite.

5.3 Skeleton-based Approaches

Since the skeleton techniques outlined in Section 4.3 largely aim to absolve the programmer of responsibility for the detailed expression and exploitation of parallelism, resource analysis techniques are typically exploited in the library implementation itself, both statically and even dynamically. With the exception of the work in [18], the programmer is unaware of the cost model's existence.

5.4 Using Statically Inferred Granularity Information for Parallel Coordination

The three granularity estimation systems outlined in Section 4.4 share the following notable features.

- The inference engine is simple and cheap, but limited to non-recursive functions.
- Most of the information is inferred statically, but in some cases a light-weight run-time analysis is applied, too.

– The inferred quantitative information is mostly used in a qualitative way (whether a thread is profitable, i.e. large enough to be evaluated in parallel).

For an application domain where imprecise, mostly qualitative information is sought, *ad hoc* techniques or light-weight, type-inference based techniques work very well. The mostly static nature of the analysis avoids run-time overhead.

5.5 Abstract Cost Semantics: Autonomous Mobile Programs

The rather simple Abstract Costed Operational Semantics used by AMPs is effective for a combination of reasons.

– It compares the *relative* cost of completing at the current location with the cost of completing at an alternative location.
– It requires only *coarse grain execution time estimates*. That is, rather than attempting to predict the execution time of small computational units, it compares the time to complete the entire program on the current location with the time to complete on an alternative location.
– It *incorporates dynamic information* into the static model, i.e. parameterising the model with current performance.

5.6 Type-based Analysis (Precise Model): Resource-safe Execution in Hume

The following characteristics of the resource analysis for Hume make it an effective tool for delivering guaranteed resource information.

– The analysis is purely static and thus resource-safe execution can be guaranteed before executing the code.
– To deliver such guarantees, the type-based analysis builds on strong formal foundations and the type system is proven sound.
– Through its tight integration of resource information into the type system, using numeric annotations to types, it is natural to base the static analysis on a type inference engine.
– To guarantee that the analysis delivers bounds, we must start with a precise and safe cost model, itself representing upper bounds.
– To facilitate tight upper bounds the analysis uses an accurate hardware model.

The key requirement in this application domain is safety, and thus the emphasis is on the formal aspects of the analysis. Beyond these aspects the following practical aspects contribute to the usability of the inferred resource information.

– Through the generic treatment of resources, the analysis can be easily retargeted for other (quantitative) resources.
– By using a standard linear program solver in the constraint solving stage, we achieve an efficient analysis.

6 Discussion

We have outlined a continuum of coordination cost models and a range of analysis techniques for parallel/distributed programs. By critiquing six representative resource analysis applications we identify the following general principles governing why the combination of techniques is effective in its context.

- Predominantly, the effective parallel/distributed resource analyses have been carefully designed to deliver the right information for the specific coordination purposes, and this has a number of aspects.
    - The analysis must deliver information that is sufficiently accurate. Often a surprisingly simple cost model is effective, e.g. for the AMPs in Section 5.5.
    - The analysis must combine static and dynamic components appropriately. For some applications purely static information suffices, where others require at least some dynamic information (Section 4.4).
    - In many cases it is sufficient for the analysis to produce qualitative predictions, e.g. is it worth creating a thread to evaluate an expression. However in some scenarios, such as resource-safe execution, the analysis must produce (upper) bounds (Section 4.6).
- Highly abstract resource analyses like BMF-PRAM are informative even at early phases of parallel algorithm design (Section 5.1).
- More refined, architecture dependant analyses can be utilised during parallel program development (Section 5.2).
- Improving reusable coordination abstractions like algorithmic skeletons can have a significant impact and resource analyses are commonly applied within skeleton libraries (Section 5.3).
- Even partial cost information can prove very useful, for example in deciding whether to generate parallelism (Section 4.4).
- Often the inferred quantitative information is mostly used in a qualitative way. Therefore, imprecise or relative resource information is sufficient (Section 4.4).

Clearly resource analysis will remain an important tool for parallel/distributed systems, and we trust that the principles above will assist in the design of future systems. We anticipate that these systems will be able to exploit the rapidly-improving resource analysis technologies. Indeed recent advances have already widened the range of programs for which static information can be provided (a detailed survey of WCET analyses is given in [39]) and caused a shift from run-time to compile-time techniques (Section 3). Some important trends that we anticipate in the near future are as follows.

In the area of static analyses there is a general trend to type-based analysis and to enriching basic type systems with information on resource consumption. The standard type-inference machinery has proven to be a very flexible engine that can be re-used to infer resource information.

Static analyses are getting increasingly complex and therefore more error-prone. At the same time automated theorem proving techniques increasingly mature. The combination of both, for example through formalised soundness proofs of the analysis, is desirable in particular in safety-critical systems. Alternatively, proof-carrying-code [33] or abstraction carrying code, avoid the (complex) soundness proof in general, and perform (formal) certificate validation on each program instead.

Hardware, and hence precise cost models, are becoming increasingly complex. This will push existing, low-level resource analysis to their limits and significantly worsen the WCET bounds that are achievable. For these reasons, probabilistic cost models are of increasing interest to the WCET community [34]. In the context of parallel and distributed execution, where predictions rather than bounds are sufficient, this trend will be even more relevant, but is currently not explored.

With respect to programming models, increasing interest in structured and constrained approaches [14] can be seen to bring benefits in terms of simplification, when coupled with correspondingly structured cost models. Constraining the patterns of parallelism available to the programmer facilitates the construction of tractable cost models, parameterised by the costs of the composed sequential fragments.

References

1. R. Bird and O. de Moor. *Algebra of Programming.* Prentice Hall, 1997.
2. H. Bischof, S. Gorlatch, and E. Kitzelmann. Cost Optimality and Predictability of Parallel Programming with Skeletons. In *Euro-Par 2003 Parallel Processing*, LNCS 2790, pages 682–693. Springer-Verlag, 2003.
3. Rob Bisseling. *Parallel Scientific Computation: A Structured Approach using BSP and MPI.* Oxford University Press, 2004.
4. A. Bonenfant, C. Ferdinand, K. Hammond, and R. Heckmann. Worst-Case Execution Times for a Purely Functional Language. In *Implementation of Functional Languages (IFL 2006)*, LNCS 4449, pages 235–252, Budapest, Hungary, September 4–6, 2006. Springer.
5. Murray Cole. *Algorithmic skeletons: structured management of parallel computation.* MIT Press, 1989.
6. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL'77 — Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, January 1977.
7. Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.
8. Xiao Yan Deng, Phil Trinder, and Greg Michaelson. Cost-Driven Autonomous Mobility. *Computer Languages, Systems and Structures.*
9. C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for a real-life processor. In *Intl Workshop on Embedded Software (EMSOFT'01)*, LNCS 2211, pages 469–485, Tahoe City, USA, October 8–10, 2001. Springer.
10. Steven Fortune and James Wyllie. Parallelism in random access machines. In *STOC '78: Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 114–118. ACM, 1978.

11. Frédéric Gava. Bsp functional programming: Examples of a cost based methodology. In *ICCS '08: Proceedings of the 8th international conference on Computational Science, Part I*, pages 375–385. Springer-Verlag, 2008.
12. K. Hammond, R. Loogen, and J. Berhold. Automatic Skeletons in Template Haskell. In *Proceedings of 2003 Workshop on High Level Parallel Programming, Paris, France*, June 2003.
13. Yasushi Hayashi and Murray Cole. Automated cost analysis of a parallel maximum segment sum program derivation. *Parallel Processing Letters*, 12(1):95–111, 2002.
14. Jane Hillston. *A compositional approach to performance modelling.* Cambridge University Press, New York, NY, USA, 1996.
15. Lorenz Huelsbergen, James R. Larus, and Alexander Aiken. Using the run-time sizes of data structures to guide parallel-thread creation. In *LFP '94: Proceedings of the 1994 ACM conference on LISP and functional programming*, pages 79–90, New York, NY, USA, 1994. ACM.
16. R.J.M. Hughes, L. Pareto, and A. Sabry. Proving the Correctness of Reactive Systems Using Sized Types. In *POPL'96 — Symposium on Principles of Programming Languages*, St. Petersburg Beach, Florida, January 1996. ACM.
17. C. B. Jay, M. I. Cole, M. Sekanina, and P. Steckler. A monadic calculus for parallel costing of a functional language of arrays. In *Euro-Par'97 Parallel Processing, volume 1300 of Lecture Notes in Computer Science*, pages 650–661. Springer, 1997.
18. S. Jost, H-W. Loidl, K. Hammond, N. Scaife, and M. Hofmann. "Carbon Credits" for Resource-Bounded Computations using Amortised Analysis. In *Intl. Symp. on Formal Methods (FM '09)*, LNCS. Springer, November 2009. to appear.
19. S. Jost, H-W. Loidl, N. Scaife, K. Hammond, G. Michaelson, and M. Hofmann. Worst-Case Execution Time Analysis through Types. In *Proc. of Euromicro Conference on Real-Time Systems (ECRTS'09)*, pages 13–17, Dublin, Ireland, July 1–3, 2009. Work-in-Progress Session.
20. T-M. Kuo and P. Mishra. Strictness Analysis: a New Perspective Based on Type Inference. In *FPCA'89 — Conference on Functional Programming Languages and Computer Architecture*, pages 260–272, Imperial College, London, UK, September 11–13, 1989. ACM Press.
21. H-W. Loidl. *Granularity in Large-Scale Parallel Functional Programming.* PhD thesis, Department of Computing Science, University of Glasgow, March 1998.
22. H-W. Loidl and K. Hammond. A Sized Time System for a Parallel Functional Language. In *Glasgow Workshop on Functional Programming*, Ullapool, Scotland, July 8–10, 1996.
23. E. Mera, P. López-García, G. Puebla, M. Carro, and M. Hermenegildo. Combining Static Analysis and Profiling for Estimating Execution Times. In *Intl. Symp. on Practical Aspects of Declarative Languages (PADL'07)*, LNCS 4354, pages 140–154. Springer, January 2007.
24. G. Necula. Proof-carrying-code. In *Symposium on Principles of Programming Languages (POPL'97)*, pages 106–116, Paris, France, January 15–17, 1997.
25. E. Quiñnones, E.D. Berger, G. Bernat, and F.J. Cazorla. Using Randomized Caches in Probabilistic Real-Time Systems. In *Proc. of Euromicro Conference on Real-Time Systems (ECRTS'09)*, pages 129–138, Dublin, Ireland, July 1–3, 2009. IEEE.
26. Brian Reistad and David K. Gifford. Static dependent costs for estimating execution time. In *LFP '94: Proceedings of the 1994 ACM conference on LISP and functional programming*, pages 65–78, New York, NY, USA, 1994. ACM.
27. Olin Shivers. *Control-Flow Analysis of Higher-Order Languages.* PhD thesis, School of Computer Science, Carnegie Mellon University, May 1991.

28. D. B. Skillicorn and W. Cai. A cost calculus for parallel functional programming. *J. Parallel Distrib. Comput.*, 28(1):65–83, 1995.
29. R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The Worst-Case Execution-Time Problem—Overview of Methods and Survey of Tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):1–53, 2008.
30. Gagarine Yaikhom, Murray Cole, and Stephen Gilmore. Combining measurement and stochastic modelling to enhance scheduling decisions for a parallel mean value analysis algorithm. In *International Conference on Computational Science (2)*, pages 929–936, 2006.

Global and local space properties of stream programs

Marco Gaboardi[1] and Romain Péchoux[2]

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

Abstract. In this paper, we study semantic interpretation criteria in order to ensure safety and complexity properties of first order Haskell like programs on streams. We study global and local upper bounds properties of both theoretical and practical interests guaranteeing that the size of each output stream element is bounded by a function in the maximal size of the input stream elements.

1 Introduction

A wider interest for infinite data structures and, particularly, streams has emerged in the last past two decades. Indeed, advances in computer networking combined with the creation of modern operating systems have made streaming practical and affordable for ordinary computers, thus leading streaming to become one of the most used network technologies.
This technological jump has coincided with a renewal of interest in theoretical infinitary models and studies. Several formal frameworks have been designed for the manipulation of infinite objects including infinitary rewriting [1], infinitary lambda-calculus [2] and computable analysis, which provides several models of computation over real numbers [3]. Several properties of these models such as infinitary weak and strong normalizations and complexity classes definitions and characterizations, among others, have been deeply studied in the literature.
An interesting approach to deal with infinite data is the use of laziness in functional programming languages [4]. In languages like Haskell, streams are list expressions whose elements are evaluated on demand. In this way streams can be treated by finitary means.
In parallel, several studies have emerged on the underlying theories. Many effort have been made on studying tools and techniques, as co-induction and bisimulation, to prove stream program equivalence [5, 6]. Other studies have been made in developing techniques to ensure productivity, a notion introduced in [7]. A stream definition is productive if it can be effectively evaluated in a unique constructor infinite normal form. Productivity is in general undecidable, so, many restricted languages and restricting criteria have been studied to ensure it [8–12].
Besides program equivalence and productivity, other stream properties surprisingly have received little attention. Some interesting considerations about buffering and overflow in stream programs have been made in [10, 13] emphasising that

complexity aspects of streams are of real practical interest. Moreover, the fact that usual tools of complexity theory, well behaving on inductive data types, cannot be directly adapted to streams suggests that an extensive study of theoretical tools is necessary.
In [14], we considered a small stream Haskell-like first order language and we started the study of some *space properties*, i.e. properties about the size of stream elements. We presented a new method that use semantic interpretations in order to ensure I/O and synchrony upper bounds properties for functions working on streams. The semantic interpretations used there are extensions of the notions of quasi-interpretation [15] and sup-interpretation [16], introduced to obtain upper bounds on finitary term rewriting systems.
The method introduced in [14] is promising and is well adapted to purely operational reasoning, nevertheless the properties studied there are limited to properties about *functions* working on input streams and they do not consider *definitions* of streams, i.e. functions that do not have streams in input. In particular, such properties do not hold even for simple examples of stream programs like

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

In the present work, we generalize the method of [14], in order to study *properties of stream programs*, i.e. properties of both functions working on streams and stream definitions. In particular, we study space properties of streams like `nats` or `ones`. We design criteria to ensure a *global* and a *local* space upper bound properties.
Consider the following stream program:

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

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 k, i.e. the maximal natural number $\underline{\texttt{n}}$ 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 bound by a constant k. In order to generalize this property, we study a *Global Upper Bound* (GUB) property ensuring that the size of stream elements is bound by a function in the maximal size of the input elements.
Analogously, consider the following stream program:

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

Clearly, every stream `s` built using `nats` and `sad` is not globally bound. Nevertheless it is easy to verify that for every such an `s` there exists a function f such that every element `a` of `s` in the *local* position n has a size bound by $f(n)$. In order to generalize this property, we study a *Local Upper Bound* (LUB) property ensuring that the size of the n-th evaluated element of a stream is bounded by

a function in its index n and the maximal size of the input.
The above properties, very natural from a complexity theory point of view, are also of practical interest since they can be used to prove upper bounds on classical stream examples. For this reason, we study two distinct criteria guaranteeing them.
From the technical point of view, in order to ensure the global upper bound property, we need to simply adapt the tools developed in [14]. The result is simple but gives an interesting insight on the way global properties of infinite data types can be given in a finitary way. Conversely, in order to ensure the local upper bound we need an extension of the usual semantic interpretation. In particular, we introduce the new notion of *parametrized semantic interpretation*, i.e. semantic interpretation where functions depend on external parameters. Parametrized semantic interpretations allow us to ensure the local upper bound and seem to be a pertinent tool for future developments.

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

2 Preliminaries

2.1 Syntax of the first order sHask language

We consider a first order Haskell-like language `sHask` computing on stream data presented in [14]. Consider three disjoint sets $\mathcal{X}$, $\mathcal{C}$ and $\mathcal{F}$ representing the set of *variables*, the set of *constructor symbols* and, respectively, the set of *function symbols*. A `sHask` program consists in a set of definitions described in the the grammar of Table 1, where $\mathtt{x} \in \mathcal{X}$, $\mathbf{c} \in \mathcal{C}$ and $\mathtt{f} \in \mathcal{F}$. Throughout the paper, we use the identifier `t` to represent a symbol in $\mathcal{C} \cup \mathcal{F}$ and the notation $\overline{\mathtt{d}}$, for some identifier `d`, to represent the sequence $\mathtt{d}_1, \ldots, \mathtt{d}_n$, whenever n is clear from the context. We will also use the notation $\mathtt{t}\ \overrightarrow{\mathtt{e}}$ as a short for the application $\mathtt{t}\ \mathtt{e}_1\ \cdots\ \mathtt{e}_n$, whenever `t` is a symbol of arity n. Notice that, as usual, application associates to the left. Moreover, we distinguish a special *error symbol* **Err** in $\mathcal{C}$ of arity 0 corresponding to pattern matching failure.
The language `sHask` includes a `Case` operator to carry out pattern matching and first order program definitions. Note that `Case` operator can appear only in definitions. In this, our grammar presentation differs from the one in [14], but there we also have some additional conditions that turn the definitions to be exactly the same considered here. We will use set of definitions $\mathtt{f}\ \overrightarrow{\mathtt{p_1}} = \mathtt{e}_1, \ldots, \mathtt{f}\ \overrightarrow{\mathtt{p}_k} = \mathtt{e}_k$ as *syntactic sugar* for an expression of the shape $\mathtt{f}\ \overrightarrow{\mathtt{x}} = \mathtt{Case}\ \overline{\mathtt{x}}\ \mathtt{of}\ \overline{\mathtt{p_1}} \rightarrow \mathtt{e}_1, \ldots, \overline{\mathtt{p}_k} \rightarrow \mathtt{e}_k$.

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

Table 1. `sHask` syntax

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

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

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

Table 2. `sHask` type system

that do not contain other lists. We assure this property by a typing restriction similar to the one of [13].

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

$$
\begin{array}{rll}
\sigma, \tau ::= & \alpha \mid \mathtt{Nat} \mid \sigma \times \sigma & \texttt{(source types)} \\
\mathtt{A} ::= & a \mid \sigma \mid \mathtt{A} \times \mathtt{A} \mid [\sigma] & \texttt{(base types)}
\end{array}
$$

where α is a source type variable*, a is a* base type variable*,* `Nat` *is a* constant type *representing natural numbers, $\times$ and $[\ ]$ are* type constructors*.*

Notice that the above definition can be extended to standard algebraic data types. In the sequel, we use α, β to denote source type variables, a, b to denote base type variables, σ, τ to denote source types and $\mathtt{A}, \mathtt{B}$ to denote base types. As in Haskell, we allow restricted polymorphism, i.e. a source type variable α and a base type variable a represent every source type and respectively every base type. As usual, $\to$ associates to the right. For notational convenience, we will use $\overrightarrow{\mathtt{A}} \to \mathtt{B}$ as an abbreviation for $\mathtt{A}_1 \to \cdots \to \mathtt{A}_n \to \mathtt{B}$.
Every function and constructor symbol $\mathtt{t}$ of arity n come equipped with a type $\mathtt{A}_1 \to \cdots \to \mathtt{A}_n \to \mathtt{A}$. Well typed symbols, patterns and expressions are defined using the type system in Table 2. Note that the symbol **Err** can be typed with

every base type `A` in order to get type preservation in the evaluation mechanism. Moreover, it is worth noting that the type system, in order to allow only first order function definitions, assigns functional types to constructors and function symbols, but only base types to expressions. Typing judgments are of the shape $\mathtt{t} :: \overrightarrow{\mathtt{A}} \to \mathtt{B}$, for some symbol `t` and some type $\overrightarrow{\mathtt{A}} \to \mathtt{B}$. In our examples, we will only consider three standard data types: numerals, lists and pairs, encoded by the constructor symbols 0 and infix $+1$, `nil` and infix : and, respectively, $(\_,\_)$. In this work,we are specifically interested in studying stream properties. So we pay attention to particular classes programs working on $[\alpha]$, the type of both finite and infinite lists of type α. In what follows we use a terminology slightly different from the one used in [14]. A function symbol `f` is called a *stream function* if $\mathtt{f} :: [\sigma_1] \to \cdots \to [\sigma_n] \to \overrightarrow{\tau} \to [\sigma]$ with $n > 0$. In the case where $\mathtt{f} :: \overrightarrow{\tau} \to [\sigma]$, the function symbol `f` is called a *stream constructor*. Given a definition $\mathtt{f}\,\overrightarrow{\mathtt{p}} = \mathtt{e}$ we say that it is a *function definition* if `f` is a stream function, otherwise if `f` is a stream constructor we say that it is a *stream definition*. In what follows, we will in general talk about properties of function symbols to stress that such properties holds both for functions and stream definitions.

Example 1. Consider the following program:

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

`zip` is a stream function and `nats` is a stream constructor.

2.2 `sHask` lazy operational semantics

The `sHask` language has a standard lazy operational semantics, where sharing is not considered. The semantics is described by the rules of Table 3 using substitutions, where a substitution σ (sometimes denoted $\{\mathtt{e}_1/x_1, \ldots, \mathtt{e}_n/x_n\}$) is a mapping from variables to expressions. The computational domain is the set of `Values` defined in Table 1. `Values` are either particular expressions with a constructor symbol at the outermost position or the symbol **Err** corresponding to pattern matching errors. Note that the intended meaning of the notation $\mathtt{e} \Downarrow \mathtt{v}$ is that the expression `e` *evaluates* to the value $\mathtt{v} \in \mathtt{Values}$. As usual in lazy semantics the evaluation does not explore the entire results but stop once the requested information is found.

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

2.3 Preliminary notions

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

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

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

$$\frac{\mathtt{Case}\ \mathtt{e}^1\ \mathtt{of}\ \mathtt{p}_1^1 \to \ldots \to \mathtt{Case}\ \mathtt{e}^m\ \mathtt{of}\ \mathtt{p}_1^m \to \mathtt{d}_1 \Downarrow \mathbf{Err} \quad \mathtt{Case}\ \overline{\mathtt{e}}\ \mathtt{of}\ \overline{\mathtt{p}_\mathbf{2}} \to \mathtt{d}_\mathbf{2}, \ldots, \overline{\mathtt{p}_\mathbf{n}} \to \mathtt{d}_\mathbf{n} \Downarrow \mathtt{v}}{\mathtt{Case}\ \overline{\mathtt{e}}\ \mathtt{of}\ \overline{\mathtt{p}_1} \to \mathtt{d}_1, \ldots, \overline{\mathtt{p}_n} \to \mathtt{d}_n \Downarrow \mathtt{v}}\ (\text{c})$$

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

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

Table 3. `sHask` lazy operational semantics

$$\begin{aligned}
&!!\ ::\ [\alpha] \to \mathtt{Nat} \to \alpha \\
&(\mathtt{x} : \mathtt{xs})\ !!\ \ 0 \ \ = \ \ \mathtt{x} \\
&(\mathtt{x} : \mathtt{xs})\ !!\ (\mathtt{y}+1) = \mathtt{xs}\ !!\ \mathtt{y}
\end{aligned}$$

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

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

where $\hat{\mathtt{C}}$ is a function symbol representing the *strict* version of the primitive constructor $\mathbf{c}$. For example in the case where $\mathbf{c}$ is $+1$ we can define $\hat{\mathtt{C}}$ as the program $\mathtt{succ} :: \mathtt{Nat} \to \mathtt{Nat}$ defined by $\mathtt{succ}\ 0 = 0+1$ and $\mathtt{succ}\ (\mathtt{x}+1) = (\mathtt{x}+1)+1$. When we want to stress that an expression $\mathtt{e}$ is completely evaluated (i.e. evaluated to a constructor value) we denote it by $\underline{\mathtt{e}}$. A relevant set of completely evaluated expressions is the set $\mathtt{N} = \{\underline{\mathtt{n}} \mid \underline{\mathtt{n}} :: \mathtt{Nat}\}$ of *canonical numerals.* In general we write $\underline{0}, \underline{1}, \ldots$ for concrete instances of canonical numerals. Finally we introduce a notion of *size* for expressions.

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

$$\begin{aligned}
|\mathtt{e}| &= 0 && \textit{if } \mathtt{e} \textit{ is a variable or a symbol of arity } 0 \\
|\mathtt{e}| &= \sum_{i \in \{1,\ldots,n\}} |\mathtt{e}_i| + 1 && \textit{if } e = \mathtt{t}\ \mathtt{e}_1\ \cdots\ \mathtt{e}_n,\ \mathtt{t} \in \mathcal{C} \cup \mathcal{F}.
\end{aligned}$$

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

3 Interpretations

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

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

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

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

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

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

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

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

Additive assignments have the following interesting property.

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

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

Proof. Assume that for every $\boldsymbol{c}$ of arity n whenever $n > 0$ then $(\!|\boldsymbol{c}|\!)(X_1, \cdots, X_n) = \sum_{i=1}^{n} X_i + \alpha_{\boldsymbol{c}}$, with $\alpha_{\boldsymbol{c}} \geq 1$, otherwise $(\!|\boldsymbol{c}|\!) = 0$. If $\underline{\mathtt{v}}$ is a constructor symbol of arity 0 then $|\underline{\mathtt{v}}| = (\!|\underline{\mathtt{v}}|\!) = 0$, else $\underline{\mathtt{v}} = \mathbf{c}\ \underline{\mathtt{v}}_1 \ldots\ \underline{\mathtt{v}}_n$ and we show the result by induction on the size. Take $\alpha = \max_{\mathbf{c} \in \mathcal{C}}(\alpha_{\mathbf{c}})$ and suppose that $|\underline{\mathtt{v}}_i| \leq (\!|\underline{\mathtt{v}}_i|\!) \leq \alpha \times |\underline{\mathtt{v}}_i|$:

$$\begin{aligned} |\mathbf{c}\ \underline{\mathtt{v}}_1 \ldots\ \underline{\mathtt{v}}_n| &= \sum_{i=1}^{n} |\underline{\mathtt{v}}_i| + 1 \leq \sum_{i=1}^{n} (\!|\underline{\mathtt{v}}_i|\!) + \alpha_{\mathbf{c}} = (\!|\mathbf{c}\ \underline{\mathtt{v}}_1 \ldots\ \underline{\mathtt{v}}_n|\!) \\ &\leq \sum_{i=1}^{n} \alpha \times |\underline{\mathtt{v}}_i| + \alpha = \alpha \times |\mathbf{c}\ \underline{\mathtt{v}}_1 \ldots\ \underline{\mathtt{v}}_n| \end{aligned}$$

The first inequality is a consequence of the combination of induction hypothesis and the fact that $\alpha_{\mathbf{c}} \geq 1$. The second inequality is a consequence of the combination of induction hypothesis and the fact that $\alpha_{\mathbf{c}} \leq \alpha$. □

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

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

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

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

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

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

Proof.

1. By induction on the derivation length and can be found in [17].
2. It is a direct consequence of point 1 of this proposition because the lazy semantics is just a particular rewrite strategy.
3. By induction on the size of constructor values using point 2 of this proposition and monotonicity. □

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

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

Proof. By induction on the size of expressions, it can be found in [14]. □

In the following sections, we study global and local stream properties related to constructor value size upper bounds. Moreover, we introduce criteria that use interpretations to ensure them. Although these properties are mostly undecidable, the criteria we discuss are decidable when considering restricted classes of functions, for example polynomials over real numbers of bounded degree and coefficients, see [18, 19] for a more detailed discussion.

4 Global upper bound (GUB)

In studing space properties of programs it is natural to relate the output element size with respect to the input size. Ensuring this provides interesting information about the complexity of functions computed by the corresponding program. However, recognize the true complexity measure in order to express interesting bound properties, when lazy languages and infinite data are involved, is not so obvious. In particular, streams are infinite data and it would be nonsense to consider their whole size as input. One solution is to take the maximal size of a stream element as a parameter and to bound the maximal output element size by a function in the maximal input size. This is what we call a global upper bound because it bounds the size of output elements globally. Notice that, in general, such a definition has a meaning if the input stream has a maximal element size. This trivially holds when there is no input stream.

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

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

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

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

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

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

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

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

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

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

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

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

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

We show a more involved example.

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

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

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

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

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

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

5 Local upper bound (LUB)

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

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

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

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

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

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

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

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

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

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

A parametrized assignment is monotonic if it is monotonic with respect to each of its arguments, including the parameter $L \in \mathbb{R}$.
We extend the notion of additivity to parametrized interpretation so that an additive symbol **c** *of arity n, for some $k \geq 1$, has the interpretation*

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

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

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

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

Proof.

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

: in the context $\mathsf{C}[\diamond]$). By definition of parametrized interpretations, we also know that for all L (and in particular for r'), $(\!|l\sigma|\!)_L \geq (\!|r\sigma|\!)_L$. So we have $(\!|\mathsf{C}|\!)_r((\!|l\sigma|\!)_{r'}) \geq (\!|\mathsf{C}|\!)_r((\!|r\sigma|\!)_{r'}) = (\!|\mathsf{d}|\!)_r$, by monotonicity.

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

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

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

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

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

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

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

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

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

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

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

It is a parametrized interpretation and, consequently, `nats` is LUB.

Now, we show an intermediate result for parametrized interpretations.

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

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

Proof. We proceed by induction on $\underline{\mathtt{n}} \in \mathtt{N}$.
Let $\underline{\mathtt{n}} = \mathtt{0}$ and $\mathtt{e}\ !!\ \mathtt{0} \Downarrow \mathtt{v}$. It is easy to verify that necessarily $\mathtt{e} \Downarrow \mathtt{hd : tl}$. By definition of $(\!|:|\!)$ and by Proposition 2.2 we have $(\!|\mathtt{e}|\!)_0 \geq (\!|\mathtt{hd : tl}|\!)_0 \geq (\!|\mathtt{hd}|\!)_0$. By Proposition 2.3, if $\mathtt{eval\ hd} \Downarrow \mathtt{v}$ then $(\!|\mathtt{hd}|\!)_0 \geq (\!|\mathtt{v}|\!)_0$. So we have $(\!|\mathtt{e}|\!)_0 \geq (\!|\mathtt{v}|\!)_0$ and then the conclusion.
Consider the case $\underline{\mathtt{n}} = \underline{\mathtt{n}}'{+}1$ and $\mathtt{e}\ !!\ (\underline{\mathtt{n}}'{+}1) \Downarrow \mathtt{v}$. It is easy to verify that necessarily $\mathtt{e} \Downarrow \mathtt{hd : tl}$, hence we have $(\!|\mathtt{e}|\!)_n \geq (\!|\mathtt{hd : tl}|\!)_n = \max((\!|\mathtt{hd}|\!)_n, (\!|\mathtt{tl}|\!)_{n-1}) \geq (\!|\mathtt{tl}|\!)_{n-1}$, by Proposition 2.2. Moreover $\mathtt{e}\ !!\ (\underline{\mathtt{n}}'{+}1) \Downarrow \mathtt{v}$ implies by definition that $\mathtt{tl}\ !!\ \underline{\mathtt{n}}' \Downarrow \mathtt{v}$ and by induction hypothesis we have $(\!|\mathtt{tl}|\!)_{n-1} \geq (\!|\mathtt{v}|\!)_0$. So we have $(\!|\mathtt{e}|\!)_n \geq (\!|\mathtt{v}|\!)_0$ and then the conclusion. □

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

Proof. It suffices to show that every function symbol has a local upper bound. For simplicity, we consider the case where for each function symbol we have only one definition. The general case with several definitions follows directly. Consider a stream function symbol $\mathtt{f} :: \overrightarrow{[\sigma]} \to \overrightarrow{\tau} \to [\sigma]$ defined by $\mathtt{f}\ \overrightarrow{\mathtt{p}_s}\ \overrightarrow{\mathtt{p}_b} = \mathtt{e}$.
Let $\underline{\mathtt{n}} \in \mathtt{N}$ and σ be a substitution and suppose $\mathtt{eval}((\mathtt{f}\ \overrightarrow{\mathtt{p}_s}\sigma\ \overrightarrow{\mathtt{p}_b}\sigma)\ !!\ \underline{\mathtt{n}}) \Downarrow \underline{\mathtt{v}}$. It is easy to verify that $(\mathtt{f}\ \overrightarrow{\mathtt{p}_s}\sigma\ \overrightarrow{\mathtt{p}_b}\sigma)\ !!\ \underline{\mathtt{n}} \Downarrow \mathtt{v}$, for some $\mathtt{v}$ such that $\mathtt{eval\ v} \Downarrow \underline{\mathtt{v}}$. By Lemma 5, $(\!|\mathtt{f}\ \overrightarrow{\mathtt{p}_s}\sigma\ \overrightarrow{\mathtt{p}_b}\sigma|\!)_n \geq (\!|\mathtt{v}|\!)_0$. By Proposition 2.3 and Lemma 1, $(\!|\mathtt{v}|\!)_0 \geq (\!|\underline{\mathtt{v}}|\!)_0 \geq |\underline{\mathtt{v}}|$. Notice that Lemma 1 still holds because if $(\!|-|\!)_L$ is an additive parametrized assignment then $(\!|-|\!)_0$ is an additive assignment. Hence we can conclude $(\!|\mathtt{f}\ \overrightarrow{\mathtt{p}_s}\sigma\ \overrightarrow{\mathtt{p}_b}\sigma|\!)_n \geq |\underline{\mathtt{v}}|$. By Lemma 4 and monotonicity:

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

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

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

We show a more involved example.

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

$$\begin{array}{ll}
\texttt{fib} :: [\texttt{Nat}] & \texttt{add} :: \texttt{Nat} \to \texttt{Nat} \to \texttt{Nat} \\
\texttt{fib} = \underline{0} : (\underline{1} : (\texttt{sad fib (tail fib)})) & \texttt{add}\ (\texttt{x}+1)\ (\texttt{y}+1) = ((\texttt{add x y})+1)+1 \\
 & \texttt{add}\ (\texttt{x}+1)\ \ \texttt{0}\ = \texttt{x}+1 \\
\texttt{tail} :: [\alpha] & \texttt{add}\ \ \texttt{0}\ \ (\texttt{y}+1) = \texttt{y}+1 \\
\texttt{tail x : xs} = \texttt{xs} &
\end{array}$$

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

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

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

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

6 Combining the criteria

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

Example 11. This program computes the componentwise positive minus between two streams of numerals:

$$\begin{array}{ll}
\texttt{min} :: \texttt{Nat} \to \texttt{Nat} \to \texttt{Nat} & \texttt{smin} :: [\texttt{Nat}] \to [\texttt{Nat}] \to [\texttt{Nat}] \\
\texttt{min}\ \ \texttt{0}\ \ \ \texttt{x}\ = \ \texttt{0} & \texttt{smin}\ (\texttt{x} : \texttt{xs})\ (\texttt{y} : \texttt{ys}) = (\texttt{min x y}) : (\texttt{smin xs ys}) \\
\texttt{min}\ (\texttt{x}+1)\ \ \texttt{0}\ = \texttt{x}+1 & \\
\texttt{min}\ (\texttt{x}+1)\ (\texttt{y}+1) = \texttt{min x y} &
\end{array}$$

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

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

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

References

1. Kennaway, J., Klop, J., Sleep, M., de Vries, F.: Transfinite Reductions in Orthogonal Term Rewriting Systems. LNCS **488** (1989) 1–12
2. Kennaway, J., Klop, J., Sleep, M., de Vries, F.: Infinitary lambda calculus. TCS **175**(1) (1997) 93–125
3. Weihrauch, K.: A foundation for computable analysis. LNCS **1337** (1997) 185–200
4. Henderson, P., Morris, J.: A lazy evaluator. Proceedings of POPL'76 (1976) 95–103
5. Pitts, A.M.: Operationally-based theories of program equivalence. In: Semantics and Logics of Computation. Cambridge University Press (1997)
6. Gordon, A.: Bisimilarity as a theory of functional programming. TCS **228** (1999)
7. Dijkstra, E.W.: On the productivity of recursive definitions. EWD749 (1980)
8. Sijtsma, B.: On the productivity of recursive list definitions. ACM TOPLAS **11**(4) (1989) 633–649
9. Coquand, T.: Infinite objects in type theory. LNCS **806** (1994) 62–78
10. Hughes, J., Pareto, L., Sabry, A.: Proving the correctness of reactive systems using sized types. Proceedings of ACM POPL'96 (1996) 410–423
11. Buchholz, W.: A term calculus for (co-) recursive definitions on streamlike data structures. Annals of Pure and Applied Logic **136**(1-2) (2005) 75–90
12. Endrullis, J., Grabmayer, C., Hendriks, D., Isihara, A., Klop, J.: Productivity of Stream Definitions. LNCS **4639** (2007) 274
13. Frankau, S., Mycroft, A.: Stream processing hardware from functional language specifications. Proceeding of IEEE HICSS-36 (2003)
14. Gaboardi, M., Péchoux, R.: Upper bounds on stream I/O using semantic interpretations. In: CSL '09. (2009) To appear.
15. Bonfante, G., Marion, J.Y., Moyen, J.Y.: Quasi-interpretations, a way to control resources. TCS - Accepted.
16. Marion, J.Y., Péchoux, R.: Resource analysis by sup-interpretation. In: FLOPS. Volume 3945 of LNCS. (2006) 163–176
17. Marion, J.Y., Péchoux, R.: Characterizations of polynomial complexity classes with a better intensionality. Proceedings ACM PPDP'08 (2008) 79–88
18. Amadio, R.: Synthesis of max-plus quasi-interpretations. Fundamenta Informaticae, 65(1–2), 2005.
19. Bonfante, G., Marion, J.Y., Moyen, J.Y., Péchoux, R.: Synthesis of quasi-interpretations. LCC2005, LICS Workshop (2005) `http://hal.inria.fr/`.

Improvements to a Resource Analysis for Hume

Hans-Wolfgang Loidl and Steffen Jost[⋆]

School of Computer Science, University of St Andrews,
St Andrews KY16 9SX, Scotland, UK
{hwloidl,jost}@cs.st-andrews.ac.uk

Abstract. The core of our resource analysis for the embedded systems language Hume is a resource-generic, type-based inference engine that employs the concept of amortised costs to statically infer resource bounds. In this paper we present extensions and improvements of this resource analysis in several ways. We develop and assess a call count analysis, as a specific instance of our inference engine. We address usability aspects and discuss an improved presentation of the inferred resource bounds together with the possibility of interactively tuning these bounds. Finally, we demonstrate improvements in the performance of our analysis.

1 Introduction

In the past [19] we have developed an amortised cost based resource analysis for a higher-order, strict functional language, namely expression-level Hume. Salient features of this analysis are its strong formal foundations, building on amortised program complexity and type systems, high performance due to employing efficient linear program solvers, and the possibility to express not only size-dependent but also data-dependent bounds on (generic) resource consumption. This analysis has been successfully used to infer upper bounds on the heap- and stack-space consumption and on the worst-case execution time of expression-level Hume programs.

One of the main strengths of our analysis is its flexible design, which permits easy adaptation to model other quantitative resources. In essence, only a cost table, mapping abstract machine instructions to basic costs, needs to be modified. We use this flexibility to develop a call count analysis for (higher-order) programs. The bounds inferred by our analysis are in general data-dependent and we demonstrate this strength on a standard textbook example of insertion into a red-black tree, which is discussed in context of our automatic amortised resource analysis herein for the first time.

This paper also addresses pragmatic limitations in the practical application and acceptance of our type-based analysis.

We have found that the presentation of resource bounds in the form of numeric annotations to the types is difficult to understand for the non-expert user.

[⋆] We acknowledge financial support by EU Framework VI projects IST-510255 (EmBounded) and IST-26133 (SCIEnce), and by EPSRC grant EP/F030657/1 (Islay).

We therefore produced an elaboration module, which translates the annotated types, produced by our analysis, into more easily digestable closed form expressions. Furthermore, the bounds shown to the user are usually just *one* of many possible solutions, picked by an heuristic. All solutions to the linear program yield valid cost bounds and there is no "best" solution. Hence we now allow the user to interactively explore the solution space, starting with the solution presented by our heuristic, which we will describe here for the first time.

While our analysis was designed from the start to be highly efficient, we identified several possibilities of further improving its performance. These issues cover the tuning of the Haskell implementation as well as more tightly integrating the constraint solving phase into the overall analysis. As a result we achieve a speedup factor of up to 1.36.

The main *contributions* of this paper are:

– the development and assessment of a function call count analysis, as an instance of our resource-generic amortised-cost-based resource analysis;
– concrete evidence of enhanced resource bounds due to the data-dependence, rather than only size-dependence, of our analyses;
– the development of an elaboration module providing interactive resource bounds phrased in natural-language as opposed to special type annotations;
– and the development and assessment of several improvements to the performance of the analysis.

The structure of the paper is as follows. In Section 2 we present a new call count analysis and apply it to several example programs. In Section 3 we discuss improvements made to the usability of our analysis. In particular, we present an elaboration module in Section 3.1, which translates enriched types, encoding resource consumption, into a human readable, closed-form expression. Furthermore, we demonstrate how the user can explore the solution space of the analysis in Section 3.2. In Section 4 we discuss and quantify improvements made to the performance of the analysis. Section 5 reviews related work. Finally, Section 6 summarises our results.

2 Call Count Analysis

In this section we use the flexibility of our resource analysis to instantiate a *call count analysis*. The goal of this analysis is to determine an upper bound on the number of function calls made in the program, possibly restricting the count to explicitly named functions. This metric is of particular interest for higher-order programs, where determining a bound on the number of calls to a certain function requires an inter-procedural analysis, since a call one function may trigger further calls to itself or other functions.

Beyond being of just theoretical interest, call count information is of relevance for example on mobile devices, where the function of interest may be the transmission of a message, which is charged for by the mobile network provider. In this scenario the "costs" of a function call are very real and measurable in £. For this reason, this particular example has been studied in the Mobius project [3], where Java bytecode has been analysed in order to deliver such call count information.

Our cost table for the call count metric therefore features three parameters: a boolean indicating whether or not calls to built-in functions (like vector operations) should be counted; a list of function identifiers to be ignored; and a list of function identifiers to be counted exclusively. Note that the latter two are mutually exclusive, since either all function calls except for the named ones are counted, or conversely, all function calls are ignored except for calls to the named functions. These parameters are useful in the above depicted usage scenario, where only certain functions use a chargeable service.

The resulting cost table is relatively simple,[1] with almost all entries in the cost table being zero, except for three: true function applications, built-in function application and closure creation overhead. Recall that the higher-order Hume language features under- and over-application, but not a general lambda abstraction[2]. We therefore distinguish between calling a true top-level function and a closure, since those generally have different costs associated with it. For the call count metric, the cost parameter for the application of closures is zero, since the actual function called depends on how the closure was created. Therefore at the time of closure creation, an overhead parameter is added to the cost of applying the created closure later, which thus accounts for each use of that closure. As a concrete example, we want to count the number of calls to the `add` function in the following definition of `sum`:

```
add :: num -> num -> num;
add x y =  x + y;

fold :: (num -> num -> num) -> num -> [num] -> num;
fold f n []     = n;
fold f n (x:xs) = fold f (f x n) xs;

sum :: [num] -> num;
sum xs = fold add 0 xs;
```

Since we count only `add`, the type of the `add` closure created by under-application in the body of `sum` shows a cost of one per application of the closure. When folding this closure over a list, a cost proportional to the length of the list will be inferred. We get as a result the type

```
ARTHUR3 typing for Call Count add:   (list[C<1>:int,#|N]) -(0/0)-> int
```

which encodes a cost of 1 for each cons-cell of the input list,[3] i.e. a bound of n, where n is the length of the list.

[1] See [19] for a detailed cost table featuring 15 entries; the actual implementation has much more entries, roughly two per syntax construct and built-in operator.

[2] However, our prototype implementation also features lambda abstraction directly.

[3] The annotated function type A`-(`x/y`)->`B means that execution requires at most x resource units, of which y are free for reuse afterwards. Any constructor followed by `<`n`>` within type A means that up to n resources may be additionally required for *each* occurrence of that constructor within the input. Also see Section 3.1.

Table 1. Results from the Resource Analyses

Program	Cost model	Analysis (N=10)		Cost model	Analysis (N=10)	
		absolute	ratio		absolute	ratio
Call Count Analysis				*Heap Space Analysis*		
`sum`	22	22	1.00	88	88	1.00
`zipWith`	21	21	1.00	190	192	1.01
`repmin`	60	60	1.00	179	179	1.00
`rbInsert`	10	20	2.00	208	294	1.41
WCET Analysis				*Stack Space Analysis*		
`sum`	16926	21711	1.28	34	34	1.00
`zipWith`	27812	32212	1.16	139	140	1.01
`repmin`	47512	58759	1.24	81	222	2.74
`rbInsert`	27425	43087	1.57	82	155	1.89

Table 1 presents analysis results for call counts, heap- and stack-space consumption, and worst-case execution time. The cost model results have been obtained from an instrumented version of the Hume abstract machine [14]. The `sum` example computes the sum over a list of integers, using a higher-order `fold` function, as shown above. The `zipWith` example implements a variant of the `zip` function parameterised with the function to apply to each pair of elements (which is `add` in the tested code). The `repmin` example replaces all leaves in a binary tree by its minimal element, using map and fold functions over trees. Finally, the `rbInsert` function inserts an element into a red-black tree, possibly re-balancing the resulting tree. The test input lists and trees had a size of 10.

The results for heap- and stack-space consumption in Table 1 show generally good results. The tree-based programs, `repmin` and `rbInsert`, deliver poorer bounds for stack, since our analysis cannot express bounds in terms of the depth of a data-structure, which would be the accurate bound in this case. This problem is most pronounced for `repmin`, which performs 2 tree traversals. The `rbInsert` example will be discussed in more detail below. The time bounds are necessarily less accurate, since the costs for the basic machine instructions are already worst-case bounds, which we obtained through analysis of the generated machine code with the **aiT** tool [11]. In general we aim for bounds within 30% of the observed costs, which might not be the worst case. We achieve this goal for three of the four test programs. The results for the call counts show an exact match for the `sum`, `zipWith` and the `repmin` examples, all of which use higher-order operations.

In the following we take a closer look on the `rbInsert` example, with the source code given in Figure 1. This example is directly taken from Okasaki's text-book [20]. A red-black tree is a binary search tree, in which nodes are coloured red or black. With the help of these colours, invariants can be formulated that guarantee that a tree is roughly balanced. The invariants are that on each path no red node is followed by another red node, and that the number of black nodes is the same on all paths. These invariants guarantee that the lengths of

```
type num = int 16;
data colour = Red | Black;
data tree = Leaf | Node colour tree num tree;

balance :: colour -> tree -> num -> tree -> tree;
balance Black (Node Red (Node Red a x b) y c) z d =
   Node Red (Node Black a x b) y (Node Black c z d);
balance Black (Node Red a x (Node Red b y c)) z d =
   Node Red (Node Black a x b) y (Node Black c z d);
balance Black a x (Node Red (Node Red b y c) z d) =
   Node Red (Node Black a x b) y (Node Black c z d);
balance Black a x (Node Red b y (Node Red c z d)) =
   Node Red (Node Black a x b) y (Node Black c z d);
balance c a x b = Node c a x b;

ins :: num -> tree -> tree;
ins x Leaf = Node Red Leaf x Leaf;
ins x (Node col a y b) = if (x<y)
                           then balance col (ins x a) y b
                         else if (x>y)
                           then balance col a y (ins x b)
                         else (Node col a y b);

rbInsert :: num -> tree -> tree;
rbInsert x t = case ins x t of
                (Node _ a y b) -> Node Black a y b;
```

Fig. 1. Example `rbInsert`: insertion into a red-black tree

any two paths in the tree differs by at most a factor of two. This loose balancing constraint has the benefit that all balancing operations in the tree can be done locally. The `balance` function only has to look at the local pattern and restructure the tree if a red-red violation is found. The `rbInsert` function in Figure 1 performs the usual binary tree search, finally inserting the node as a red node in the tree, if it does not already exist in the tree, and balancing all trees in the path down to the inserted node.

The heap bound for the `rbInsert` function, inferred by our analysis is:

```
ARTHUR3 typing for HumeHeapBoxed:
 (int,tree[Leaf|Node<10>:colour[Red|Black<18>],#,int,#]) -(20/0)->
                                tree[Leaf|Node:colour[Red|Black],#,int,#]
```

This bound expresses that the total heap consumption of the function is $10n + 18b + 20$, where the n is the number of nodes in the tree, and b is the number of black nodes in the tree. The latter demonstrates how our analysis is able to produce data-dependent bounds by attaching annotations to constructors of the input structure. This gives a more precise formula compared to one that only refers to the size of the input structure. In this example the $18b$ part of the formula reflects the costs of applying the `balance` function, which restructures a sub-tree with a black root in the case of a red-red violation. The analysis assumes a worst-case, where every black node is affected by a balancing operation. Note that, due to the above invariants, this cannot occur for a well-formed red-black tree: any insertion into the tree will trigger at most 2 balancing operations (see [9][Chapter 13]). As expected, capturing these (semantic) constraints is beyond the power of our type system.

Similarly the upper bound on the number of clock cycles required to compute the `rbInsert` function is associated with the black nodes in the input tree:

```
ARTHUR3 typing for Time:
(int,tree[Leaf|Node<2889>:colour[Red|Black<1901>],#,int,#]) -(2712/0)->
                                  tree[Leaf|Node:colour[Red|Black],#,int,#]
```

Finally, the call count analysis for `rbInsert` yields:

```
ARTHUR3 typing for Call Count:
 (int,tree[Leaf|Node<2>:colour[Red|Black],#,int,#]) -(1/0)->
                                  tree[Leaf|Node:colour[Red|Black],#,int,#]
```

This type encodes a bound of $2n+1$, where n is the number of nodes. By attaching costs to the constructors of the input it is possible to distinguish between nodes and leaves. However, it is currently not possible to express the fact that in the tree traversal the number of nodes visited on each path is at most $\log\, n$. In the extension of the amortised cost based analysis, developed by Campbell [6], such information on the depth of data structures is available, and his system is able to infer logarithmic bounds on space consumption for such examples.

3 Usability Improvements

3.1 Elaboration Module

Input dependent bounds on resource usage of programs are only useful if they easily allow one to distinguish large classes of inputs having roughly the same resource usage. Consider having a black box for a program that can compute the precise execution cost for any particular input. Even if this black box computes very fast, one still needs to examine all inputs one by one in order to determine the worst case or to establish an overview of the general cost-behaviour. Since the number of concrete inputs may be large or even infinite, this is generally infeasible.

The original amortised analysis technique as proposed by Tarjan [24], being very powerful, may generally produce such a precise "black box" cost oracle. This is not a hindrance for a *manual* technique, as the mathematician performing the method has direct control over the complexity and behaviour of the "black box" that is created. However, for an automated analysis we must ensure that the outcome is always simple enough to be understood and useful. The cost bounds produced by our automated version of the amortised analysis technique are always simple. Namely, they are *linear in the sizes* of the input. This restriction to linearly dependent bounds is our chosen trade-off to obtain an *automated inference* for the amortised analysis.[4] This design guarantees that we can easily divide all possible inputs into large classes having a similar cost. For example, for a program processing two lists we might learn instantly from the result of our efficient analysis that the execution cost can be bounded by a constant times the length of the second list, thereby throwing all inputs which only differ in the first argument into the same cost class. Furthermore we immediately know the execution cost for infinitely many such input classes.

We now exploit this even further to produce cost bounds expressed in natural language. Previously, the cost bound had only been communicated to the user using type annotations. While these allowed a concise and comprehensive presentation of the derived bounds, they also required a fair amount of expertise to understand, despite most derived bounds being actually rather simple. The new elaboration module helps to interpret the annotated types by ignoring irrelevant information, summing up weights in equivalent positions and producing a commented cost-formula, parameterised over a program's input.

We now revisit the results for the red-black tree insertion function from Section 2. We use the option `--speak` to immediately obtain:

```
ARTHUR3 typing for HumeHeapBoxed:
(int,tree[Leaf<20>|Node<18>:colour[Red|Black<10>],#,int,#]) -(0/0)->
                               tree[Leaf|Node:colour[Red|Black],#,int,#]

Worst-case Heap-units required to call rbInsert in relation to its input:
  20*X1 + 18*X2 + 10*X3
    where
        X1 = number of "Leaf" nodes at 1. position
        X2 = number of "Node" nodes at 1. position
        X3 = number of "Black" nodes at 1. position
```

This makes it easy to see that the number of black nodes is significant for the cost formula. Furthermore the cost formula $20X_1 + 18X_2 + 10X_3$ is obviously much more compact and easy to understand. We are directly told that X_1 corresponds to the number of leaves in the first tree argument (there is only one tree argument here); that X_2 corresponds to the number of all nodes and that X_3 corresponds to the number of black nodes. Note that this bound is inferior to the one shown in Section 2, and we will address this in the following subsection.

A further simplification implemented in our elaboration module is the recognition of list types and list-like types, i.e. all constructors are single recursive (e.g. `Cons`), except for precisely one constructor being non-recursive (e.g. `Nil`). In this case it is clear that each element of such a type must have precisely one such terminating constructor. Therefore the weight attached to the terminal constructor may be moved outwards.

[4] Recent (yet unpublished) research by Hofmann and Hoffmann show how the restriction of the inference to linear bounds may be lifted.

For example, consider the annotated type of a function that receives a list of integer lists as its input:[5]

```
(list[Nil<1>|Cons<2>:list[Nil<3>|Cons<4>:int,#],#]) -(5/0)-> int]

Worst-case Heap-units required to call foo in relation to its input:
  6 + 5*X1 + 4*X2
    where
        X1 = number of "Cons" nodes at 1. position
        X2 = number of "Cons" nodes at 2. position
```

We see that the cost formula is much simpler than the annotated type, which features 5 non-zero annotations, whereas the cost formula has only three parameters. However, this useful simplification naturally incurs a slight loss of information. The annotated type expresses that 3 resource units are only needed once the end of the inner list is reached. If the program may *sometimes* choose to abort processing a list to the very end, those 3 resource units are not needed. While our analysis must produce a guarantee on the resource usage for all cases, this simplification means no quality loss for the bound produced. Nevertheless it is conceivable that a programmer might sometimes make use of this additional knowledge about the resource behaviour of the program. However, we believe that the majority of cases benefit from the simplification.

3.2 Interactive Solution Space Exploration

Programs often admit several possible resource bounds and it is in general not clear which bound is preferable. For a simple example, we consider the standard list zipping, such as adding two lists of numerical values. Using a Haskell-style syntax we have:

$$\begin{array}{lll}
\texttt{zipWith add}\,[\,] & [10,20] & = [\,] \\
\texttt{zipWith add}\,[1,2,3,4] & [10,20] & = [11,22] \\
\texttt{zipWith add}\,[1,2,3,4] & [10,20,30,40,50,60] & = [11,22,33,44]
\end{array}$$

We immediately see that the resource consumption, be it time or space, depends on the length of the *shorter* input list. Therefore, we have the following admissible annotated types for the closure created by `zipWith add`:

```
(list[C<6>:int,#|N<2>],list[C<0>:int,#|N<2>]) -(0/0)-> list[C:int,#|N]
Worst-case Heap-units required to call zipWith add in relation to input:
  2 + 6*X1
     where  X1 = number of "C" nodes at 1. position

(list[C<0>:int,#|N<2>],list[C<6>:int,#|N<2>]) -(0/0)-> list[C:int,#|N]
Worst-case Heap-units required to call zipWith add in relation to input:
  2 + 6*X1
     where  X1 = number of "C" nodes at 2. position
```

[5] The output was simplified to ease understanding. Our prototype implementation requires monomorphisation, so each list type would require unique constructors.

The first type says that the cost is proportional to six times the length of the first input list plus two whereas the latter type says that the cost is proportional to six times the length of the second input list plus two. Both bounds are equally useful, and it depends on external knowledge which one is preferable.

Our analysis is capable of expressing this choice within the constraints generated for the program. In fact, if we were to run the prototype analysis on a program involving the function `zipWith`, where the input for `zipWith` is generated in another part of the analysed program and in such a manner that one input list is often significantly shorter than the other one, the analysis would pick the type that admits the overall lower cost bound.

The problem here lies in communicating this choice to the user, when we analyse function `zipWith` all on its own. The analysis cannot guess which type would be preferable to the user, based on the intended use of the function. On the other hand, the overall meaning of a set of constraints is generally incomprehensible to a human due to sheer size, even after extensive simplification. A set of constraints describes an n-dimensional polytope, where n is the number of input sizes[6], i.e. the number of constructors per position in the input and output plus two for the annotations on the function arrows. Hence for the simple `zipWith add` example, we already have a solution space of (at least) 8 dimensions. So even the possibility of printing all the vertices of the polytope seems impractical.

We resolve this dilemma through interaction. The analysis first presents a solution as usual. The user may then increase or decrease the "penalty" attached to a resource variable in the type. The constraints are then re-solved with an adjusted objective function, in which the modified penalties may cause another solution to be produced. This re-solving can be done almost instantaneously, thanks to the improvements described in Section 4, most notably due to keeping the pre-solved constraints and solution in memory. The new solution is printed on the screen again, and the user may then specify another cost variable to be altered, until the cost bound is satisfactory. Step-by-step, the user can thus explore the entire solution space for the analysed program.

Note that our implementation of the analysis has always employed a heuristic that was able to guess the "desired" result for many program examples right away. However, allowing the user to tweak the solver's priorities is also a good way of understanding the overall resource behaviour of a program. So even in the many cases that are already properly resolved by the heuristic guessing a suitable objective function, this interaction may offer valuable insights.

Optimising the bound for red-black tree insertion. We again revisit the red-black tree example from Section 2, this time to show how the interactive optimisation of the solution works. Invoking our analysis for the heap space

[6] Note the solution space generally has a much higher dimension due to the necessary introduction of intermediate variables. Furthermore our experience showed that eliminating these intermediate variables is either best left to the LP solver, which is far more efficient at this task, or rather omitted entirely, since the intermediate variables have actually proven useful for the heuristic to pick a "good" solution.

metric as before, but adding the command-line option for interaction, we obtain a prompt after the solution.

```
ARTHUR3 typing for HumeHeapBoxed:
(int,tree[Leaf<20>|Node<18>:colour[Red|Black<10>],#,int,#]) -(0/0)->
                                tree[Leaf|Node:colour[Red|Black],#,int,#]
Worst-case Heap-units required in relation to input:
20*X1 + 18*X2 + 10*X3
    where
        X1 = number of "Leaf" nodes at 1. position
        X2 = number of "Node" nodes at 1. position
        X3 = number of "Black" nodes at 1. position

Enter variable for weight adjustment or "label","obj" for more info:
```

We are unhappy with the high cost associated with the leaves of the tree, since it seems unreasonable to require such a high cost for processing an empty leaf. Therefore we ask the analysis to lower this value considerably, by increasing the penalty of the associated resource variable from 6 to 36.

```
Enter variable for weight adjustment or "label","obj" for more info: X1
Old objective weight: 6.0 Enter relative weight change: 30
Setting CVar '351' to weight '36.0' in objective function.

(int,tree[Leaf|Node<10>:colour[Red|Black<18>],#,int,#]) -(20/0)->
                                tree[Leaf|Node:colour[Red|Black],#,int,#]
Worst-case Heap-units required in relation to input:
20 + 10*X1 + 18*X2
    where
        X1 = number of "Node" nodes at 1. position
        X2 = number of "Black" nodes at 1. position
```

This already results in the desired solution. The fixed costs increase from 0 to 20, the costs associated with all leaves drop from 20 to 0, and the cost of each red node decreases by 8. Since every tree contains at least one leaf, this bound is clearly better for all inputs.

So here we have an example where the implemented heuristic for choosing *a* solution picked unfortunately a clearly inferior one. However, recall that both solutions represent guaranteed upper bounds on the resource consumption, so it could be that the first solution was already precise enough. Furthermore, if we analyse a program that also constructs a red-black tree as input for the `rbInsert` function, then the LP-Solver automatically chooses the second solution in order to minimise the overall cost, which includes the cost of creating the input and all the potential associated with the input data-structure.

It is important to note that each and every function application will choose the most appropriate admissible annotated type for the function, albeit each function is analysed only once. This is achieved by simply copying the constraints associated with a function for each of its applications, using fresh variable names throughout. Since the generated LPs are sparse and proven to be easily solvable, this blow-up of constraints is of little concern. More information on this mechanism for resource parametricitry can be found in [18]. This once more illustrates that the result for analysing a function is the set of all admissible annotations, rather than any single annotation.

4 Performance Improvements

The combined Hume prototype analyses delegated the solving of the generated linear programming (LP) problem to the LP-solver lp_solve [4], which is available under the GNU Lesser General Public Licence.

Technically this was done by writing all constraints in a human readable format to a file and then calling lp_solve to solve that file. The solution was then read via a Unix pipe and also recorded in a text file. This solution had the advantage that the generated LP was directly tangible. The file contained various comments, in particular the line and column of the source code that ultimately had triggered the generation of that particular constraint. This yielded very high transparency and was very useful in developing and validating the resource analysis. Furthermore, one could alter the LP by hand for experimentation and feed it to the solver again without any difficulties.

However, this solution also had several drawbacks, namely:

1. Communicating large data-structures, such as linear programming problems, via files on the hard-disk of a computer is very slow.
2. Altering the constraints just slightly, requires the full, slow repetition of transmitting the entire LP and solving it from scratch.
3. Running the analysis requires the user to install and maintain the lp_solve command-line tool separately.
4. lp_solve only allows very limited floating point precision when using file communication, causing rounding errors of minor significance.

We have thus added the option of calling the lp_solve library, written in C, directly through the foreign function interface (FFI) of the Glasgow Haskell Compiler (GHC) [12]. This solution now resolves all of the above issues. The library is now linked into the combined Hume prototype analyses' executable file, producing an easy to use stand-alone tool. Furthermore, eliminating the first two problems was a direct prerequisite for realising the interactive solution space exploration described in Section 3.2. Interaction can only work if the time the user is required to wait between each step is very small. The solver lp_solve supports this by fast incremental solving, where the last solution and the pre-solved constraints are kept in the memory and can be adjusted for subsequent solving. Therefore in all program examples examined thus far on our contemporary hardware, *re-solving* the linear program could be done within a fraction of a second, for example less than 0.02 seconds for the biquad filter program example, as opposed to 0.418 seconds required for first-time solving as shown in Table 2.

Table 2. Run-time for Analysis and for LP-solving

Program	Constraints		Run-time non-FFI		Run-time FFI		Speedup	
	Number	Variables	Total	LP-solve	Total	LP-solve	Total	LP-solve
biquad	2956	5756	1.94s	1.335s	1.43s	0.418s	1.36	3.20
cycab	3043	6029	2.81s	2.132s	2.75s	1.385s	1.02	1.54
gravdragdemo	2692	5591	2.16s	1.605s	2.14s	1.065s	1.01	1.51
matmult	21485	36638	104.88s	101.308s	84.17s	21.878s	1.25	4.63
meanshift	8110	15005	11.32s	9.851s	11.01s	6.414s	1.03	1.54
pendulum	1115	2214	0.76s	0.479s	0.67s	0.260s	1.13	1.84

Solving the linear programming problem via the foreign function interface is therefore the default setting now. However, the previous mechanism of calling lp_solve via the command-line is still available through option `--noapisolve`, since this is still quite useful when transparency is desired more than performance, which is often the case when studying the combined Hume analysis itself by applying it to small program examples.

Table 2 summarises the run-times[7] of both versions of the combined Hume resource analysis on some program examples: the *non-FFI version* using option `--noapisolve`, which uses lp_solve via the command-line to solve the constraint set and an *FFI version* using option `--apisolve`, which calls the lp_solve library through the foreign-function-interface. For each version we show the total run-time of the analysis as well as the run-time for just the LP-solving component (both in seconds). The final two columns show the speedup of the FFI version over the non-FFI version.

The applications used in Table 2 to compare the run-times of the analysis are as follows. The `biquad` application is a biquadratic filter application. The `gravdragdemo` application is a simple, textbook satellite tracking program using a Kalman filter, developed in preparation for the `biquad` application. The `cycab` application is the messaging component of the cycab application. The `pendulum` application balances an inverted pendulum on a robot arm with one degree of freedom. The `meanshift` computer vision application is a simple lane tracking program. Finally, `matmult` is a function for matrix multiplication, which was automatically generated from low-level, C-like code. The generated program makes heavy use of higher-order functions and of vectors for modelling the state space, due to the design of the automatic transformation. This results in a high number of constraints and therefore in a compute-intensive analysis phase.

We see that the speedup for the LP-solving part is quite impressive (51–363%). However, one should recall that the command-line version (non-FFI) is required to build the C data-structures holding the constraint set, whereas in the library version (FFI), this task is performed by our prototype analysis, delivering

[7] The performance measurements in Table 2 have been performed on a 1.73GHz Intel Pentium M with 2MB cache and 1GB main memory.

the constraints ready-to-use. This also explains why the overall run-time does not decrease by the same amount as the time spent on LP solving.

The overall speedup is largely varying for our program examples (1–36%), but with the overall run-time being just around 1–3 seconds, it is hard to judge which is the dominating factor in processing. For the large `matmult` example, the only one where the analysis is actually working for a noticeable time, the overall run-time could be reduced by an impressive 25%, or roughly 20 seconds.

We conclude that calling the lp_solve library through the FFI is also beneficial for programs where LP solving actually requires a significant amount of time, in addition to making interactive solving possible at all, as mentioned earlier.

5 Related Work

Type-based Resource Analysis: Using type inference to statically determine quantifiable costs of a program execution has a long history. Most systems use the basic type inference engine to separately infer information on resource consumption. In contrast, our analysis uses a tight integration of resource information into the type, by associating numeric values to constructors in the type. These values are the factors in a linear formula expressing resource consumption. Another notable system, which uses such a tight integration of resources into the type system, is the sized type system by Hughes et al. [17], which attaches bounds on data structure sizes to types. The main difference to our work is that sized types express bounds on the size of the underlying data structure, whereas our weights are factors of the corresponding sizes, which may remain unknown. The original work was limited to type checking, but subsequent work has developed inference mechanisms [7,26]. Vasconcelos' PhD thesis [25] extends these previous approaches by using abstract interpretation techniques to automatically infer linear approximations of the sizes of recursive data types and the stack and heap costs of recursive functions. A combination of sized types and regions is also being developed by Peña and Segura [21], building on information provided by ancillary analyses on termination and safe destruction.

Amortised Costs: The concept of amortised costs has first been developed in the context of complexity analysis by Tarjan [24]. Hofmann and Jost were the first to develop an automatic amortised analysis for heap consumption [15], exploiting a difference metric similar to that used by Crary and Weirich [10]. The latter work, however, only *checks* bounds, and does not *infer* them, as we do. Apart from inference, a notable difference from our work to the work of Tarjan [24] is that credits are associated on a *per-reference* basis instead of the pure layout of data within the memory. Okasaki [20] also noted this as a problem, resorting to the use of *lazy evaluation*. In contrast, per-reference credits can be directly applied to strict evaluation. Hofmann and Jost [16] have extended their method to cover a comprehensive subset of Java, including imperative updates, inheritance and type casts. Shkaravaska et al. [22] subsequently considered heap consumption inference for first-order polymorphic lists, and are currently studying extensions to non-linear bounds. Campbell [6] has developed the ideas of depth-based and temporary credit uses, so giving better results for stack usage.

Other Resource Analyses: Another system that is generic over the resource being analysed is the COSTA system [2]. Its inference engine is based on abstract interpretation. In a first phase a set of recurrence relations are generated, which are solved in a second phase by a recurrence solver that is tailored for the use of resource analysis and as such produces demonstrably better results than general recurrence solvers [1]. Gómez and Liu [13] have constructed an abstract interpretation of determining time bounds on higher-order programs.

Several recent systems aim specifically at the static prediction of heap space consumption. Braberman et al. [5] infer polynomial bounds on the live heap usage for a Java-like language with automatic memory management. However, unlike our system, they do not cover general recursive methods. Chin et al. [8] present a heap and stack analysis for a low-level (assembler) language with explicit (de-)allocation. Their system, like ours, is restricted to linear bounds. Taha's work on staged compilation [23] emphasises the distinction between computation on the development and deployment platforms. In the first stage (development platform) computations can have arbitrary resource consumption, but in the second stage no new heap allocations are allowed.

WCET Analysis: Calculating bounds on worst-case execution time (WCET) is a very active field of research, and we refer to [27] for a detailed survey.

6 Summary

This paper presented extensions and improvements of our amortised cost based resource analysis for Hume [19]. By instantiating our resource inference to the new cost metric of call counts, we obtain information on the number of (possibly specific) function calls in higher-order programs. While initial results from an early call count analysis where presented in [18], we here give the first discussion of the analysis itself and assess it for a range of example programs. In particular, we demonstrate for a standard textbook example of insertion into a red-black tree that the inferred bounds are in general data-dependent and therefore more accurate than bounds that are only size-dependent.

Furthermore, we presented improvements of our analysis in terms of usability, performance, and quality of the bounds. As an important new feature for the acceptance of our type based analysis, the resource bounds are now translated into closed-form cost formulae. Based on feedback from developers of Hume code in interpreting the resource bounds, encoded in annotated types, we consider the improvement in usability through the elaboration module as the biggest step in making our analysis available to a wider community. Although this improvement is the most difficult one to quantify, we believe that such presentation of resource bounds as closed-form formulae is essential for the acceptance of a type-based inference approach.

We have also reported on significant improvements made to the performance of the analysis. For the example programs used in this paper, we observe a speedup factor of up to 1.36, mainly due to a tighter integration of the linear program solving through the FFI interface provided by GHC.

As future work we plan to investigate whether combining our approach with a sized-type analysis might also allow the inference of super-linear bounds, while still using efficient LP-solver technology, possibly multiple times. One challenge for the analysis will be to capture all future code optimisations that might be added to the Hume compiler. We are experimenting with approaches where resource usage is exposed in the form of explicit annotations to a high-level intermediate form. In this way, we may be able to retain the advantage of close correlation of the analysis with the source language, while being able to model a much wider range of compiler optimisation, and still maintain the advantage of easy resource targeting.

The prototype implementation of our amortised analysis is available online at `http://www.embounded.org/software/cost/cost.cgi`. Several example Hume programs are provided, and arbitrary programs may be submitted through the web interface.

References

1. E. Albert, P. Arenas, S. Genaim, and G. Puebla. Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis. In *Intl. Symp. on Static Analysis (SAS'08)*, LNCS 5079, pages 221–237, Valencia, Spain, July 15–17, 2008. Springer.
2. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. COSTA: Design and Implementation of a Cost and Termination Analyzer for Java Bytecode. In *Intl. Symp. on Formal Methods for Components and Objects (FMCO'07)*, LNCS 5382, pages 113–132, Amsterdam, The Netherlands, October 24–26, 2007. Springer.
3. L. Beringer, M. Hofmann, and M. Pavlova. Certification Using the Mobius Base Logic. In *Intl. Symp. on Formal Methods for Components and Objects (FMCO'07)*, LNCS 5382, pages 25–51, Amsterdam, The Netherlands, October 24–26, 2007.
4. M. Berkelaar, K. Eikland, and P. Notebaert. lp_solve: Open source (mixed-integer) linear programming system. GNU LGPL (Lesser General Public Licence). `http://lpsolve.sourceforge.net/5.5`.
5. V. Braberman, F. Fernández, D. Garbervetsky, and S. Yovine. Parametric Prediction of Heap Memory Requirements. In *Intl. Symp. on Memory Management (ISMM'08)*, New York, USA, 2008.
6. B. Campbell. *Stack Usage Analysis.* PhD thesis, Edinburgh Univ., 2008.
7. W.-N. Chin and S.-C. Khoo. Calculating Sized Types. *Higher-Order and Symbolic Computing*, 14(2,3):261–300, 2001.
8. W-N. Chin, H.H. Nguyen, C. Popeea, and S. Qin. Analysing Memory Resource Bounds for Low-level Programs. In *Intl. Symp. on Memory Management (ISMM'08)*, New York, USA, 2008.
9. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Cliff Stein. *Introduction to Algorithms.* MIT Press and McGraw-Hill, second edition, 2001.
10. K. Crary and S. Weirich. Resource Bound Certification. In *Symp. on Principles of Prog. Langs. (POPL'00)*, pages 184–198, 2000.

11. C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and Precise WCET Determination for a Real-Life Processor. In *Intl Workshop on Embedded Software (EMSOFT'01)*, LNCS 2211, pages 469–485, Tahoe City, USA, October 8–10, 2001. Springer.
12. The Glasgow Haskell Compiler. `http://haskell.org/ghc`.
13. G. Gomez and Y.A. Liu. Automatic Time-Bound Analysis for a Higher-Order Language. In *Conf. on Languages, Compilers and Tools for Embedded Systems (LCTES'98)*, LNCS 1474, pages 31–40, Montreal, Canada, June 1998. Springer.
14. K. Hammond. Exploiting Purely Functional Programming to Obtain Bounded Resource Behaviour: the Hume Approach. In *First Central European Summer School (CEFP'05), Revised Selected Lectures*, LNCS 4164, pages 100–134, Budapest, Hungary, July 4–15, 2005. Springer.
15. M. Hofmann and S. Jost. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *Symp. on Principles of Prog. Langs. (POPL '03)*, pages 185–197, January 2003.
16. M. Hofmann and S. Jost. Type-Based Amortised Heap-Space Analysis. In *European Symposium on Programming (ESOP'06)*, LNCS 3924, pages 22–37. Springer, 2006.
17. R.J.M. Hughes, L. Pareto, and A. Sabry. Proving the Correctness of Reactive Systems Using Sized Types. In *Symp. on Principles of Prog. Langs. (POPL '96)*, pages 410–423, St. Petersburg Beach, Florida, January 1996. ACM.
18. S. Jost, K. Hammond, H-W. Loidl, and M. Hofmann. Static Determination of Quantitative Resource Usage for Higher-Order Programs. In *Symp. on Principles of Prog. Langs. (POPL '10)*, Madrid, Spain, January 2010. to appear.
19. S. Jost, H-W. Loidl, K. Hammond, N. Scaife, and M. Hofmann. "Carbon Credits" for Resource-Bounded Computations using Amortised Analysis. In *Intl. Symp. on Formal Methods (FM '09)*, LNCS. Springer, November 2009. to appear.
20. C. Okasaki. *Purely Functional Data Structures.* Cambridge University Press, 1998. ISBN 0521663504.
21. R. Peña and C. Segura. A First-Order Functl. Lang. for Reasoning about Heap Consumption. In *Draft Proc. Int'l Workshop on Impl. and Appl. of Functl. Langs. (IFL '04)*, pages 64–80, 2004.
22. O. Shkaravska, Ron van Kesteren, and Marko van Eekelen. Polynomial Size Analysis of First-Order Functions. In *Typed Lambda Calculi and Applications (TLCA 2007)*, Paris, France, June 26–28, 2007.
23. W. Taha. Resource-Aware Programming. In *Intl. Conf. on Embedded Software and Systems (EMSOFT '04)*, pages 38–43, Dec. 2004.
24. R.E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, April 1985.
25. P.B. Vasconcelos. *Cost Inference and Analysis for Recursive Functional Programs.* PhD thesis, University of St Andrews, 2008.
26. P.B. Vasconcelos and K. Hammond. Inferring Costs for Recursive, Polymorphic and Higher-Order Functional Programs. In *Intl. Workshop on Impl. of Functional Languages (IFL '03)*, pages 86–101. Springer, 2004.
27. R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The Worst-Case Execution-Time Problem—Overview of Methods and Survey of Tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):1–53, 2008.

Compositional Analysis of Hume Box Iterations

Christoph A. Herrmann[1] and Kevin Hammond[1]

University of St Andrews, Scotland KY16 9SX, UK,
{ch,kh}@cs.st-andrews.ac.uk,
WWW home page: http://www.cs.st-andrews.ac.uk/~ch

Abstract. We present work in progress about our recent reaction time analysis of Hume box compositions in the presence of free control parameters for iteration. Hume is a language for resource-critical systems, in which asynchronous boxes controlled by a scheduler exchange data via a static network.
The analysis is carried out by a symbolic simulation in Haskell with an appropriate encoding for optional availability of wire values, a feature of Hume to mimic asynchronous execution.
We explain our model for the symbolic execution of Hume programs, present two realistic examples, and discuss the challenges involved in the analysis of programs with a free control parameter. We present a heuristics for simple cases and suggest the use of a loop template in case that the heuristics fails.

1 Introduction

We present a symbolic timing analysis for compositions of concurrent boxes in Hume [8], a language for resource-critical systems. Although the current implementation is tailored towards the pecularities of Hume, we believe that the challenges that we are tackling are of importance to program analysis for a wide range of systems which consist of components communicating in a restricted asynchronous fashion. The challenges are component executions based on availability of data at their inputs and certain patterns, the treatment of unknown parameters whose actual value decides about program branches taken and numbers of iterations, and a timing analysis which accounts for special program branches.

Hume programs consist of compositions of boxes, which are activated by a scheduler and exchange data via a static network of wires. Each wire connects an output port of a box with an input port of another or the same box, and can buffer at most one data object. If a box cannot release results because one of the values on the wires has not been consumed yet, these results remain in the box heap and the box is blocked from execution until the values are released. If a box is not blocked, it can execute if at least one particular input pattern matches, thereby arguments from input ports can be ignored, and the availability of such ignored values is not required for a successful match and box execution. The patterns are specified in the program part for the box, using the symbol `*` for ignored inputs. Each pattern forms the left-hand side of a purely-functional computational rule, and when it matches, the values on the input ports are assigned to program variables and used in the evaluation of the rule's right-hand side, which determines the values of the output ports. The programmer can use the symbol `*` to specify that no value is produced for a particular output port. The right-hand side expressions can apply pure functions, which are potentially recursive, potentially higher-order. The programmer can define her own data types and constructors which are then associated with individual execution time costs by an amortised analysis of the box execution [9]. The analysis of Hume box compositions presented here builds on this amortised analysis, i.e., assumes that we already can obtain the execution time of each rule of each box. In particular, we are interested in the execution time for a particular class of tasks, starting from a set of input events from the environment, involving several box executions, and finishing with the creation of the last output event.

The paper is organised as follows. In the next section, we describe our model for the symbolic execution of Hume programs and abstraction of Hume programs that we want to analyse in Haskell [14]. Section 3 presents two realistic examples, one in which a box is executed in two different modes, and one in which a box is executed repeatedly, controlled by an iteration counter passed around. Section 4 discusses the case that the initial values of iteration counters are unknown. Section 5 suggests a loop template to simplify the analysis. Section 6 discusses related work and Section 7 concludes.

2 The Model for a Hume Superstep

Several alternative scheduling orders are possible for Hume boxes, e.g., to allow for efficient parallelisation. However, any legitimate schedule be consistent with the denotational Hume semantics. One order which can be easily understood from an operational perspective is the *superstep scheduling* mechanism. The idea of a superstep in Hume is similar to that in the bulk-synchronous parallel programming (BSP) model [18]: within each superstep each box is executed at most once and the data that a box produces is not available to the consumer before the next superstep. It follows that within a superstep any execution order and any potential parallelisation of box executions leads to the same behaviour. However, this only holds within a single superstep, and not across several supersteps. This means that we can view all box executions within a step as being semantically independent of each other, i.e., forming part of a function which maps the system state at the beginning of a superstep to the state at the end.

2.1 Simple Artificial Hume Example

Figure 1 depicts a system with two Hume boxes named A and B and three wires labeled x, y and z. Without further knowledge of the program for the boxes, Box B can react to either or both available inputs on y and z and Box A can be blocked from further execution as long as it wants to release a value onto wire y but cannot do so because B has not consumed the previous value on y yet.

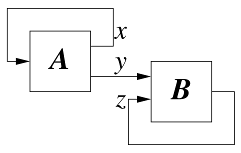


Fig. 1. Two Boxes

We model availability of wire values by a `Maybe` type constructor in Haskell. Figure 2 shows the instance of our model for a scheduling cycle of the two box composition. Because of the superstep semantics assertion of outputs does not have an impact on executions in the same step. In the model, this is taken into account by a distinction into Phase 1 and Phase 2.

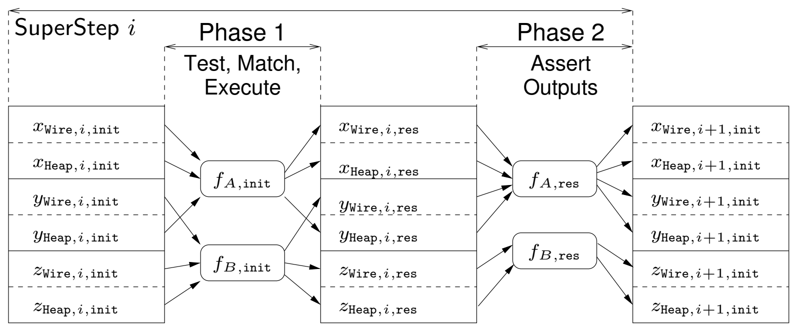

SuperStep i
Phase 1
Test, Match, Execute
Phase 2
Assert Outputs
$x_{\texttt{Wire},i,\texttt{init}}$
$x_{\texttt{Heap},i,\texttt{init}}$
$y_{\texttt{Wire},i,\texttt{init}}$
$y_{\texttt{Heap},i,\texttt{init}}$
$z_{\texttt{Wire},i,\texttt{init}}$
$z_{\texttt{Heap},i,\texttt{init}}$
$f_{A,\texttt{init}}$
$f_{B,\texttt{init}}$
$x_{\texttt{Wire},i,\texttt{res}}$
$x_{\texttt{Heap},i,\texttt{res}}$
$y_{\texttt{Wire},i,\texttt{res}}$
$y_{\texttt{Heap},i,\texttt{res}}$
$z_{\texttt{Wire},i,\texttt{res}}$
$z_{\texttt{Heap},i,\texttt{res}}$
$f_{A,\texttt{res}}$
$f_{B,\texttt{res}}$
$x_{\texttt{Wire},i+1,\texttt{init}}$
$x_{\texttt{Heap},i+1,\texttt{init}}$
$y_{\texttt{Wire},i+1,\texttt{init}}$
$y_{\texttt{Heap},i+1,\texttt{init}}$
$z_{\texttt{Wire},i+1,\texttt{init}}$
$z_{\texttt{Heap},i+1,\texttt{init}}$

Fig. 2. Hume Superstep as a Mapping Between Wire Vectors

We view each scheduling cycle as a function mapping a vector of wire values onto itself. These values are indexed by the location (`Wire`/`Heap`), the scheduling cycle (i / $i+1$) and the phase (`init`/`res`). Each kind of box contributes to the entire function with a single function for each of the two phases, e.g. $f_{A,\texttt{init}}$ and $f_{A,\texttt{res}}$ for Box A. Non-consumption of a value is modelled by copying back the wire value of the `init` vector to the same wire value of the `res` vector, e.g., when Box B does not consume y, function $f_{B,\texttt{init}}$ copies the value $y_{\texttt{Wire},i,\texttt{init}}$ to $y_{\texttt{Wire},i,\texttt{res}}$ where it tells $f_{A,\texttt{res}}$ not to assert the heap value $y_{\texttt{Heap},i,\texttt{res}}$ to $y_{\texttt{Wire},i+1,\texttt{init}}$. In the purely functional style, the absence of a value on a wire is expressed by assigning the value `Nothing` in the model. Note that we do not need extra state information for keeping track on whether a box is blocked; this can be deduced from the state of the box heap. However, other issues which we do not deal with in this paper, like rule reordering to achieve fair merging of inputs could be regarded by adding information about the current rule ordering of each box into the wire vector to form a more general state vector.

2.2 Symbolic Analysis

We could now describe the semantics of either functions (`init` / `res`) of each box in Haskell by matching the actual situation of the state of the wires and the heaps. This gives us a simulation of the behaviour of the Hume program including the execution time for a particular set of input data. But, because we are interested in the worst-case behaviour, we have to take into account all possible situations, not just a subset. If we only had a few wires carrying values of limited bitsize, it might be useful to regard the possibility of an exhaustive simulation, but in our setting this would be practically infeasible. We need to represent information in a compact form and also regard that the program can do arbitrary computations on them; this rules out approaches to do the mapping between wire vectors by a kind of matrix/vector multiplication and looking for characteristic properties of powers of such matrices (as sequences of supersteps).

We address generality by a complete symbolic setting which can deal with all possible situations that can occur due to different values of unknown parameters. However, we apply a restriction on the kind of wire values we can currently handle, but this restriction is purely a lack of features to construct and destruct data types in our symbolic language, and we aim to overcome this restriction as soon as we have found a convenient representation. In particular, in our current setting wires can only carry natural numbers. We embed these into rational numbers to use the value -1 to encode an unavailable wire value and to have the same type for our mathematical expressions in which we need fractions as coefficients of polynomials, serving as closed forms of recurrence equations. With this, we can represent iteration counters directly, but for data structures like lists we can only use a proper abstraction, e.g. the size.

We define symbolic expressions in Haskell by a generalised algebraic data type (GADT) which we name `Sym` and which has one parameter, the type of the expression, e.g. `Rational` or `Bool`. Atomic expressions are constants (`C`) and variables (`V`), and expressions can be composed using arithmetic (`:+:`,`:*:`,`:^:`), comparison (`:=:`,`:<:`) and logical (`:&:`,`:|:`) operators as well as several other constructs to build polynomials and logic formulae on them.

```
data Sym :: * -> * where
 C     :: !Rational -> Sym Rational
 V     :: !Var -> Sym Rational
 (:+:) :: !(Sym Rational) -> !(Sym Rational) -> Sym Rational
 ...
 (:<:) :: !(Sym Rational) -> !(Sym Rational) -> Sym Bool
 ...
 (:&:) :: !(Sym Bool) -> !(Sym Bool) -> Sym Bool
 If    :: !(Sym Bool) -> !(Sym Rational) -> !(Sym Rational) -> Sym Rational
 ...
```

The question is whether the symbolic treatment of `If` is sound when dealing with entire sets of possibilities for values, in some of which the predicate is true and in others of which it is false. This can easily be understood by viewing these symbolic expressions as functions from a concrete environment, i.e., in terms of the values of the variables (`V`) to boolean or rational values. In a concrete environment, each predicate is either true or false. E.g., the interpretation $\mathcal{I}$ of an `If` in an environment ε is defined by:

$$\mathcal{I}(\texttt{If}\ cond\ y\ n)\varepsilon = \begin{cases} \mathcal{I}(y)\varepsilon & \text{if } \mathcal{I}(cond)\varepsilon \ = \ \text{true} \\ \mathcal{I}(n)\varepsilon & \text{otherwise} \end{cases}$$

Stepping back from the point-wise view by universally quantifying over all possible environments is equivalent to combining all single results using a set union. E.g., if we apply a logical *and* (`:&:`) then the expression will be true for the intersection of those environments for which both operand expressions are true.

We can express the behaviour of a box in both phases in the symbolic setting by function `boxStep`, separated for each of the two phases given by the first argument. The second argument is the static information about the box and the third argument is the information for the symbolic simulation containing the wire vector values and the current execution time. The static information contains a (symbolic) function telling the condition under which at least one of the input patterns of the box matches, the (symbolic) function and the indices of the input and output wires. These indices are concrete numbers and can be used for indexing the wire vector in combination with a combined selector for heap/wire and first/second phase, e.g. `arr!(i,HeapA)` means that for wire `i` we refer to the value which resides in the box heap before the first phase. We do not distinguish between different supersteps in the symbolic simulation; the separation between the phases is enough to warrant independence of the function applications within each phase.

```
boxStep :: Ix a => Phase ->  BoxInfo a ->   TArr a Wire -> TArr a Wire
boxStep Phase_1 (matchesInput,boxFun,inwires,outwires) (arr,time)
 = let blockedVars = [ arr!(i,HeapA) | i<-outwires ]
       inputs = [ arr!(i,WireA) | i<- inwires ]
       blocked = anyAvail blockedVars
       runnable = (Not blocked) :&: matchesInput inputs
       (outputs, usedInputs, addTime) = boxFun inputs
       usedInWire rel = usedInputs!!rel
   in (arr // [ ((i,WireB), If (runnable :&: usedInWire rel)
                               noValue (arr!(i,WireA)))
              | (i,rel) <- zip inwires [0..] ]
           // [ ((o,HeapB), If runnable v (arr!(o,HeapA)))
              | (o,v) <- zip outwires outputs ],
       time :+: If blocked (V (Var "Tbl"))
                   (If (matchesInput inputs) addTime (V (Var "Tma"))))
boxStep Phase_2 (matchesInput,boxFun,inwires,outwires) (arr,time)
 = let wireAvail o = (notAvail (arr!(o,HeapB))) :|:
                     (notAvail (arr!(o,WireB)))
       canAssert = foldl (\c o -> c :&: wireAvail o) TT outwires
       doAssert o = canAssert :&: avail (arr!(o,HeapB))
   in (arr // concat [ [ ((o,WireA), If (doAssert o) (arr!(o,HeapB))
                                                     (arr!(o,WireB))),
                         ((o,HeapA), If (doAssert o) noValue
                                                     (arr!(o,HeapB))) ]
```

```
                    | o<-outwires ],
      time :+: (V (Var "Tass")))
```

The function boxFun computes a triple with the abstract values of the outputs, the inputs that have been consumed and the execution time depending on the input situation. This time is only used if the box is actually executed, otherwise the constant Tbl for attempting to process a blocked box or Tma for an unsuccessful match.

3 Realistic Examples

We sketch a couple of realistic examples just to give an impression of the advantages of the symbolic simulation. Both examples assume that we know a bound on the maximum number of scheduling cycles required. Overestimation will not harm because overrun cycles will occur in the resulting expression guarded by a predicate which is false. The expressions can be simplified at the end of each simulation cycle and partially evaluated as soon as concrete data for the variables are available. Of practical use are especially expressions parameterised in only a few parameters open for strategic decisions, like the number of objects a system is supposed to process.

3.1 Sensor control, execution in two different modes

Let us assume we have a Hume box designed for operating a sensor. Because the time that the sensor requires will be orders of magnitude larger than the execution time of a box, it makes sense to have a control box for the sensor which is executed twice; once to process a request which will involve to generate concrete actions for the sensor, and a second time to fetch the results as far as available. The time for the sensor then becomes explicit in the application program and it is it's responsibility to ensure that the sensor has had enough time or to repeat the fetching attempt later. The timing analysis sees two different activations of the box, one for the request and one to fetch the results. The symbolic formula for the execution time of the entire task will contain the time for both box activations, in idealised form something like `... :+: V (VAR "prepare_actions") :+: ... :+: V (VAR "fetch_results") :+: ....`

3.2 Nearest neighbour iteration

Although the Hume expression layer permits sophisticated recursive functions, using them might not always be desirable. Since box executions are currently atomic, it is important that their duration is short enough such that the system can process urgent interrupts quickly. Moving iterations into the coordination layer, i.e., dedicating each iteration a separate activation of a box would permit interrupts to be processed between iterations.

The example shown in Figure 3 is an algorithm computing the k nearest neighbours of an object in two- or three-dimensional space. First, we create a

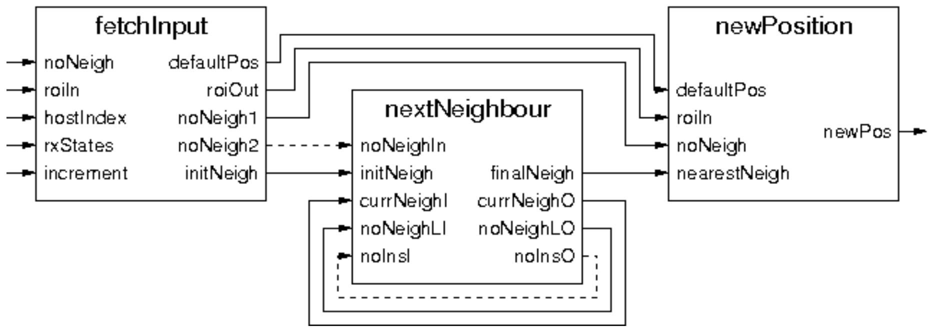


Fig. 3. Iterative calculation of nearest neighbours

list of pairs which consist of the index of the other object and the distance to it. In the first iteration, we are looking for the one with the smallest distance and move it to the front. Then, we continue successively with the rest of the list and decrement k by one until it reaches zero. The box which performs the iteration has among other wires one feedback wire for k and one feedback wire for the current state of the list of other objects. The box performs a case distinction on k; as long as it is greater than zero it sends new data into the feedback wire; if it reaches zero it sends the result list to another box.

The box `nextNeighbour` is linear in the number of objects, and we iterate it as often as how many nearest neighbours we would like to use in the calculation. This number is kept by the loop (`noNeighLO`/`noNeighLI`) and initialised by `noNeighIn`. The total time will then be a product of the number of objects and the number of nearest neighbours, i.e., expressed by a non-linear formula. The task of a single activation of the box `nextNeighbour` is to take a list `currNeighI` (initialised with `initNeigh` and fed back by `currNeighO`) with the object positions in which the first `noInsI` elements are already the closest neighbours and permute the remaining elements such that in the result the first (`noInsI+1`) elements are the closest neighbours. Since the scheduling of the boxes is transparent to the Hume program, we need to maintain a counter value for the number of box iterations which is initialised by `noNeighIn`, incremented in each iteration and fed back in the loop `noInsO`/`noInsI`. This value, depicted with dashed lines in Figure 3, plays a major role in the analysis. We show a small part of the Hume program below: a reduced specification of the box `nextNeighbour`.

```
box nextNeighbour in (...) out(...)
match
  (*,*,currNeighI,noNeighLI,noInsI)
    -> if noNeighLI>noInsI
          then (*, next currNeighI noInsI,noNeighLI, noIns+1)
          else (take noInsI currNeighI,*, *, *)
 |(noNeighIn,initNeigh,*,*,*)
    -> if noNeighIn>0
          then (*, next initN 0, noN,1)
          else ([],*, *,*);
```

The type declarations for input and output ports have been abbreviated by (...). An asterisk (*) denotes, as already mentioned, unavailable data if it appears at an input position and not generated output at an output position. Two rules are specified, separated by |. In the example program, the first rule matches if the first two input ports (appearing at the left-hand side of ->) do not provide data but the other three do; we name this branch loop because it carries values between iterations. The three values come from the previous iteration of this box. The other rule (named init) deals with the first execution of the box when the first two input ports carry data and the other three do not.

The abstract version which we use for the analysis is shown below. The first lambda abstraction in the body states when at least one pattern matches, the second contains functions for the output ports in vectorised form (lists of length 4 combined elementwise according to predicates), conditions when inputs are consumed ([cond1,cond1,...]) and the time, simplified to whether case [1] or [2] has occurred.

```
box_nextNeighbour ins outs
 = (\ [noNeighIn,initNeigh,currNeighI,noNeighLI,noInsI]
      ->     (avail noNeighIn :&: avail initNeigh)
         :|: (avail currNeighI :&: avail noNeighLI :&: avail noInsI),
    \ [noNeighIn,initNeigh,currNeighI,noNeighLI,noInsI]
      -> (let branch1a = [noValue, dc, noNeighLI, noInsI :+: C 1]
              branch1b = [dc, noValue, noValue, noValue]
              branch2a = [noValue, dc, noNeighIn, C 1]
              branch2b = [dc, noValue, noValue, noValue]
              branch1 = zipWith (If (noInsI :<: noNeighLI))
                                branch1a branch1b
              branch2 = zipWith (If (C 0 :<: noNeighIn))
                                branch2a branch2b
          in zipWith (If (avail noNeighIn :&: avail initNeigh))
                     branch2 branch1,
          let cond1 = avail noNeighIn :&: avail initNeigh
              cond2 = avail currNeighI :&: avail noNeighLI :&: avail noInsI
          in [cond1,cond1,cond2,cond2,cond2],
          If (avail noNeighIn :&: avail initNeigh)
             (V (Var "nextNeighbour[1]"))
             (V (Var "nextNeighbour[2]"))),
    ins, outs)
```

The result of the analysis significantly differs depending on whether the value for the k (in the program named `noNeighIn`) is a constant (say `C 5`) or a variable (say `V (VAR "k")`). In the first case, the analysis returns with a concrete number if the simulation is carried out for enough superstep cycles such that the result is output. In the second case, the analysis delivers a case distinction for each potential value of k that can be observed within the given number of supersteps; i.e., it says in a complicated way: `if` $k = 0$ `then` ... `else if` $k = 1$ `then` ... `else` ... However, because generating this finite case distinction, even with incremental simplification after each superstep takes a huge amount of time, it is simpler to do the analysis for all finite k by a separate simulation. What the user probably would want is an inductively defined solution in k which can then be turned into a closed form. This is discussed in the next section.

4 Iterations Depending on an Unknown Control Parameter

Now, let the system perform repetitions which are controlled by a counter or the size of a data structure. For simplicity, let us introduce a derived, abstract control parameter, which is a natural number that is only decremented, always reaches the value zero and then leads the system to deliver the expected response.

Our heuristics works as follows:

1. Find a function $\mathit{steps} : \mathbb{N} \to \mathbb{N}$ which tells for each value of the control parameter how many superstep cycles are required, for some small $k \in \mathbb{N}\backslash\{0\}$ in the form $\forall n \in \mathbb{N}, r \in \{0..k-1\} : \mathit{steps}(n \cdot k + r) = a \cdot n + b_r$. This is motivated by the fact that the system can show a cyclic behaviour already in the absence of any control parameter, as we observed in several examples, e.g., a traffic light controller.
2. Perform a symbolic simulation with control parameter $((n+1) \cdot k)$ as a symbolic expression in the variable n and the constant k for a number of a scheduling cycles. The heuristic fails if the time is not a polynomial in n; otherwise we can apply polynomial series summation to gain a polynomial for the time consumption in case the control parameter is $n \cdot k$. We can furthermore bound any time for $n \cdot k + r$ with $r < k$ by $(n + 1) \cdot k$, or precisely gain the additional cost for each r by simulation.

How can we find such a function *steps* in case it can be represented in this form? We need two additional user-defined functions working on wire vectors: one that initialises the start condition for a given number of the control parameter and one that tells when the response is output. Then, we simulate with many different constant values of the control parameter and obtain a table for the values of function *steps* at the points $0..m$. We interpolate with a set of reasonable functions, e.g. $f_k(n) = a \cdot \lfloor n/k \rfloor + b_{(n \mod k)}$, i.e., we state a linear constraint system for the coefficients a and b_i and try to solve it. We start with $k = 1$; if the system does not have a solution increase the value of k successively.

If the value of k is higher than a reasonable value (e.g., the number of all subsets of rules in the Hume program) we report a failure. Otherwise we still need to verify that the function is correct and do this as follows:

1. Verify $\forall n \in \mathbb{N} : step(k \cdot n + k) - step(k \cdot n) = a$ by simulating over a steps and verifying that at the last step the control parameter has decreased by k, when starting with the symbolic value $k \cdot n + k$ for the control parameter.
2. Verify $\forall n \in \mathbb{N}, r \in \{0..k{-}1\} : step(k \cdot n + r) - step(k \cdot n) = b_r$ by simulating over b_r steps and verifying that the control parameter has decreased by r when starting with the symbolic value $k \cdot n + r$ for the control parameter.

5 Loop Box: A Template for Iterations

The heuristics described in the previous section requires some manual assistance and has some additional assumptions. It would be useful if we could establish these assumptions and avoid the need for manual interaction by harnessing the iteration, much like `for`-loops are used in structured programming instead of `goto`'s. In this section, we present a so-called loop box which can be used to state iteration at the coordination level explicitly.

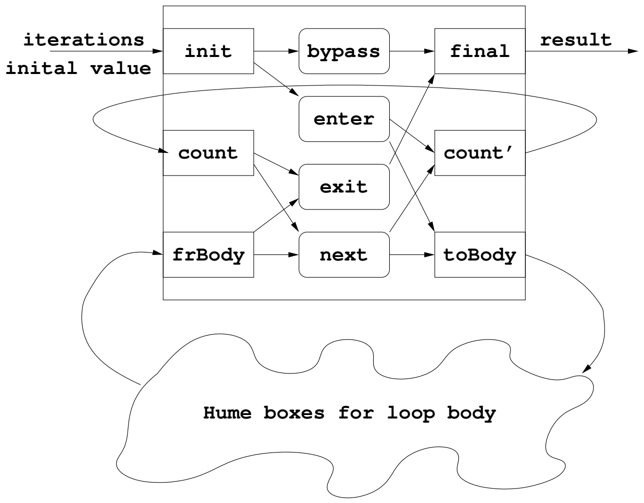


Fig. 4. Use of a loop box

Figure 4 sketches the structure of the loop box. It has three input ports named `init`, `count` and `frBody` and three output ports named `final`, `count'` and `toBody`. Port `count` is connected to `count'` by a feedback wire and thus involves the same index in the wire vector. The box is entered with a pair of the iteration counter and some initial data at the `init` port. If the counter is zero, no iteration takes place and the `bypass` function is applied. Otherwise the `enter` function initialises the iteration counter and initialises the wire to the Hume part which implements the body. When the body returns a result, the decremented loop counter is checked: if it is zero the `exit` function delivers the final result, otherwise the `next` function decrements the loop counter and provides the data for the next entry of the body.

This structure is interesting for analysis because the loop body does not share any variables in the wire vector with the rest of the Hume program, so it can be analysed in isolation. If we have an analysis result for the body, we can combine it to a result of the entire loop.

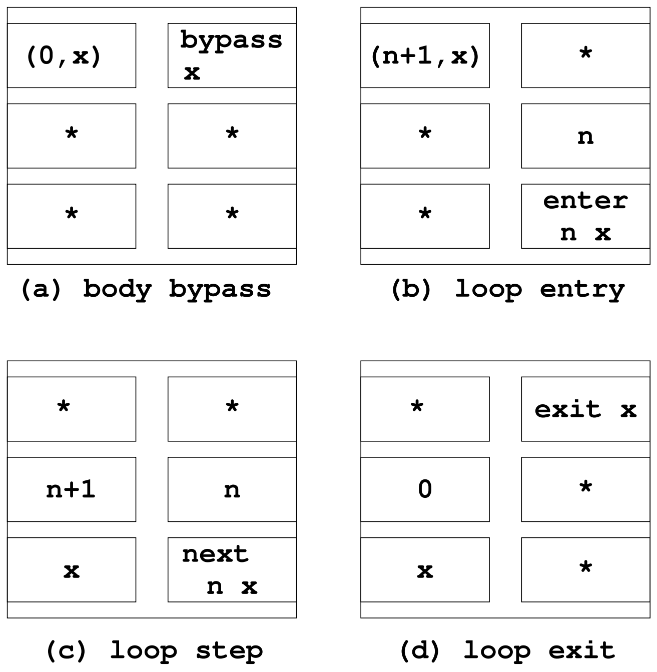


Fig. 5. Inductive patterns of a loop box

Figure 5 shows the four cases we have to distinguish. If the time for the body only depends on the iteration counter and not on the other data, then the time can be given roughly by the following generic formula.

$$t(n) = \begin{cases} t_{by} & \text{if } n = 0 \\ t_{enter} + n \cdot t_{next} + t_{exit} + \sum_{i=0}^{n-1} t_{body}(i) & \text{otherwise} \end{cases}$$

A nested iteration can easily be implemented by using two loop boxes and connecting `toBody`/`frBody` of the outer with `init`/`final` of the inner loop box.

In the analysis, we would substitute the term $t_{body}(i)$ by the time for the inner loop, obtaining likely a polynomial of degree 2 in n.

If we did not have a special part of the analysis which deals with loop boxes, then the abstract version of the outer loop box would look as follows, where we simplified the cost such that all functions have the same cost `V (Var "O")`. Since we have no means for encoding tuples, the input pair is represented as two separate wires which are supposed to be both either available or not. The inner box would have more wires due to additional tupling to provide the outer loop counter to the body.

```
box_outer ins outs
 = (\ [n,init,count,frBody]
        -> ((avail n :&: avail init) :|:
             (avail count :&: avail frBody)),
    \ [n,init,count,frBody]
        -> let brByPass = [C 0,noValue,noValue,noValue]
               brEntry  = [noValue,n,n, C 0]
               brNext   = [noValue,count :+: (C (-1)),
                           count , frBody]
               brExit   = [frBody,noValue,noValue,noValue]
               sel4 by en ne ex = If (avail n :&: avail init)
                                     (If (C 0 :<: n) en by)
                                     (If (C 0 :<: count) ne ex)
           in (zipWith4 sel4 brByPass brEntry brNext brExit,
               [avail n :&: avail init,
                avail n :&: avail init,
                avail count :&: avail frBody,
                avail count :&: avail frBody], V (Var "O")),
    ins, outs)
```

The simulation with increasing values for n delivers a series of scheduling cycles starting with 1, 9, 15, 22, 30, ... which is quadratic in n. Note that the first difference is 8 before it goes back to 6 and then starts to increment. Exact interpolation with a quadratic polynomial would thus fail except that the first point is excluded; in general there might be an entire sequence of initial points to be excluded. It is also not guaranteed that we will have a cyclic behaviour in the case of nested loops, such that we can perform a simulation over a constant number of superstep cycles.

The case distinction in the time formula for the loop box suggests a bottom-up algebraic approach in this case.

6 Related Work

Compositional cost analysis of functional programs was first considered by Wegbreit [20]. An important idea was to derive closed-form cost expressions for functions which can then be reused for the analysis of other functions. Le Métayer used a large extensible library of patterns to find closed forms for recursive cost functions in his **ACE** system [13]. Rosendahl [16] developed an abstract interpretation technique from a step-counting function. Our approach started with a similar idea of profiling by performing a simulation of the boxes using additionally abstract program values. However, we have developed this further by symbolic representation of entire sets of potential program values.

Benzinger [2] developed an analysis prototype for calculating the time complexity of programs generated by the interactive program synthesizer **Nuprl**. The user has to specify the parameters in which the complexity has to be expressed. Simplification and recurrence solving is programmed in Nuprl's ML and Mathematica's functional programming language.

If the analysis results for individual components are to be reused, and these results differ according to some parameters in which the component is used, then if we are to obtain a tight timing analysis, it is important to abstract the result over these parameters [10]. We will therefore normally express the timing as a symbolic expression. In some cases, it is possible to partition the parameter space to obtain clusters of component activations for particular ranges, in order to present a concrete timing value for each range, for example [6]. A variety of academic and commercial tools exist that can give bounds on worst-case execution time [21].

Lisper [12] describes a promising approach that uses parametric integer programming to automatically deal with parameters such as the ones that describe numbers of iterations in a flow graph, provided they are linear. Coffman et al. [4] have similarly derived polynomial time bounds for nested loop programs.

There has been relatively little work on compositional analysis. Reistad and Gifford [15] calculate execution time (not guaranteed worst-case execution times) as symbolic expressions in terms of abstract values of the input data, e.g., its size. Compositional verification of real-time systems is done by Håkansson and Pettersson [7], who apply model-checking without calculation of abstract program values to distinguish cases in the compositional verification of real-time systems. We are not aware of other related work computing situation-dependent time in a compositional fashion.

In the concurrent programming language **Toc** [11], the programmer specifies the time of each task. This provides abstract information similar to the information we have for our boxes, that needs to be verified within each task and that can be used for derivation of an earliest-deadline first schedule. The main difference to our approach is that we distinguish different execution times for different branches in each box.

This paper builds on our previous work presented at the ERCIM/DECOS workshop [9]. In the further development we took much benefit from approaches applying polynomial size functions and their extensions. Kesteren et al. [19] have suggested a restriction to polynomial size functions and to increase the degree of the class of polynomials as long as the test fails. Shkaravska et al. [17] show how to approximate multivalued size functions by a family of piecewise polynomials. Another work aiming for upper bounds where exact approximation is inconvenient is the Java bytecode analysis by Albert et al. [1] implemented in Ciao Prolog.

7 Conclusions

We have seen that symbolic simulation can up to a certain degree be applied successfully for the analysis of Hume box compositions. If the required number of superstep cycles is known it advance and constant, the main task is a good simplification of the symbolic expressions. If we know a relation between a control parameter and the number of cycles, we can use this to create a table for a series of parameter values of interest. The situation gets tricky when we are aiming for a general solution in control parameters; but possible in certain circumstances.

The use of Haskell as a prototyping language for the abstract behaviour of Hume boxes permits manual assistance by exposing the control parameter and other symbolic information of interest to the analysis.

In the case of nested iterations the analysis becomes difficult. We recommend structured programming at the coordination level, using templates of known behaviour to implement iterative patterns. Then, the analysis of the Hume program can be carried out really compositionally, in a bottom-up fashion along the composition of the templates.

We are also investigating dependent types for size verification of inductively defined specification, e.g., the size preservation of quicksort [3]. Related work was done by Danielsson [5] who applied the dependently typed language Agda for timing verification at the level of functional expressions. However, the proofs required inside a dependently typed framework will, for a considerable time, require even more manual interaction than we did here.

Acknowledgements

This work was generously funded by the Systems Engineering for Autonomous Systems (SEAS) Defence Technology Centre (DTC) established by the British Ministry of Defence. We would like to thank very much Olha Shkaravska for providing us with recent material about size analysis with polynomial upper bounds, Martin Hofmann for insight into amortised analysis with non-linear bounds and Edwin Brady for a critical discussion about the benefit of dependent types.

References

1. Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. Automatic inference of upper bounds for recurrence relations in cost analysis. In *The 15th International Static Analysis Symposium (SAS'08)*, LNCS 5079, pages 221–237. Springer-Verlag, 2008.
2. Ralph Benzinger. Automated complexity analysis of Nuprl extracted programs. *Journal of Functional Programming*, 11(1):3–31, 2001.

3. Edwin C. Brady, Christoph A. Herrmann, and Kevin Hammond. Lightweight invariants with full dependent types. In *Trends in Functional Programming*. Intellect, 2008.
4. Joel Coffman, Christopher Healy, Frank Mueller, and David Whalley. Generalizing parametric timing analysis. In *LCTES'07*, pages 152–154, 2007.
5. Nils Anders Danielsson. Lightweight semiformal time complexity analysis for purely functional data structures. In *Principles of programming languages (POPL)*, 2008.
6. Johan Fredriksson, Thomas Nolte, Andreas Ermedahl, and Mikael Nolin. Clustering worst-case execution times for software components. In Christine Rochange, editor, *7th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, Dagstuhl, Germany, 2007. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.
7. John Håkansson and Paul Pettersson. Partial order reduction for verification of real-time components. In J.-F. Raskin and P.S. Thiagarajan, editors, *FORMATS 2007*, Lecture Notes in Computer Science 4763. Springer-Verlag, 2007.
8. Kevin Hammond and Greg J. Michaelson. Hume: a domain-specific language for real-time embedded systems. In *Proc. Intl. Conf. on Generative Programming and Component Engineering (GPCE '03)*, Lecture Notes in Computer Science 2830, pages 37–56. Springer-Verlag, 2003.
9. Christoph A. Herrmann and Kevin Hammond. Towards compositional worst-case execution time analysis for Hume programs. In *Proc. ERCIM/DECOS Workshop*, 2008.
10. Meng-Luo Ji, Ji Wang, Shuhao Li, and Zhi-Chang Qi. Automated WCET analysis based on program modes. In *AST'06*, pages 36–42. ACM Press, May 2006.
11. Martin Korsgaard and Sverre Hendseth. Combining EDF scheduling with OCCAM using the Toc programming language. In Peter H. Welch, Susan Stepney, Fiona A.C. Polack, Frederick R.M. Barnes, Alistair A. McEwan, Gardiner S. Stiles, Jan F. Broenink, and Adam T. Sampson, editors, *Communicating Process Architectures 2008*, volume 66 of *Concurrent Systems Engineering Series*, pages 55–66. IOS Press, 2008.
12. Björn Lisper. Fully automatic, parametric worst-case execution time analysis. In *Proc. WCET '03: Int'l Workshop on Worst-Case Execution Time Analysis*, pages 99–102, 2003.
13. Daniel Le Métayer. ACE: an automatic complexity evaluator. *ACM Transactions on Programming Languages and Systems*, 10(2):248–266, 1988.
14. Simon Peyton Jones et al. Haskell 98 language and libraries — the revised report. Available from `http://www.haskell.org/`, December 2002.
15. Brian Reistad and David K. Gifford. Static dependent costs for estimating execution time. In *LFP '94: Proceedings of the 1994 ACM conference on LISP and functional programming*, pages 65–78, New York, NY, USA, 1994. ACM.
16. Mads Rosendahl. Automatic complexity analysis. In *FPCA '89: Proceedings of the fourth international conference on Functio nal programming languages and computer architecture*, pages 144–156, New York, NY, USA, 1989. ACM.
17. Olha Shkaravska, Marko van Eekelen, and Alejandro Tamalet. Collected Size Semantics for Functional Programs. In S.-B. Scholz, editor, *Implementation and Application of Functional Languages:*20^{th} *International Workshop, IFL 2008, Hertfordshire, UK, 2008. RevisedPapers*, LNCS. Springer-Verlag, 2008. to appear.
18. Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

19. Ron van Kesteren, Olha Shkaravska, and Marko van Eekelen. Inferring static non-monotone size-aware types through testing. *Electronic Notes in Theoretical Computer Science*, 216:45–63, 2008.
20. Ben Wegbreit. Mechanical program analysis. *Communication of the ACM*, 18(9):528–539, 1975.
21. Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *Transactions on Embedded Computing Systems*, 7(3):1–53, 2008.