Sup-interpretations, a semantic method for static analysis of program resources

Jean-Yves Marion, Romain Péchoux

Sup-interpretations, a semantic method for static analysis of program resources

JEAN-YVES MARION
and
ROMAIN PÉCHOUX
Loria, Carte project, Vandœuvre-lès-Nancy, France, and École Nationale Supérieure des Mines de Nancy, INPL, Nancy, France.

The sup-interpretation method is proposed as a new tool to control memory resources of first order functional programs with pattern matching by static analysis. It has beeen introduced in order to increase the intensionality, that is the number of captured algorithms, of a previous method, the quasi-interpretations. Basically, a sup-interpretation provides an upper bound on the size of function outputs. A criterion, which can be applied to terminating as well as non-terminating programs, is developed in order to bound the stack frame size polynomially. Since this work is related to quasi-interpretation, dependency pairs and size-change principle methods, we compare these notions obtaining several results. The first result is that, given any program, we have heuristics for finding a sup-interpretation when we consider polynomials of bounded degree. Another result consists in the characterizations of the sets of functions computable in polynomial time and in polynomial space. A last result consists in applications of sup-interpretations to the dependency pair and the size-change principle methods.

Categories and Subject Descriptors: F.2.m [**Analysis of Algorithms and Problem Complexity**]: Miscellaneous ; F.3.1 [**Computation by Abstract Devices**]: Complexity Measures and Classes

General Terms: Resources control, static analysis of first order languages

1. INTRODUCTION

This work is part of a general investigation on program complexity analysis and, particularly, on first order functional programming static analysis. It deals with the notion of sup-interpretation previously introduced in [Marion and Péchoux 2006]. A sup-interpretation is an interpretation which gives an upper bound on values computed by the functions and expressions of a program. It provides methods to control some resource aspects by static analysis. The notion of sup-interpretation is used to define a criterion, called quasi-friendly criterion, which ensures that the size of the values computed by a program verifying this criterion is polynomially bounded in the size of its inputs.

Author's address: Loria, Carte project, B.P. 239, 54506 Vandœuvre-lès-Nancy Cedex, France. Jean-Yves.Marion@loria.fr and Romain.Pechoux@loria.fr

The practical issue of such a criterion is to provide the amount of space resources that a program needs during its execution. This is crucial for many critical applications and is of real interest in computer security. There are several approaches which aim at solving the same problem. The first approach is by monitoring computations. However, the monitor may crash unpredictably by memory leak if it is compiled with the program. Moreover, we cannot predict the memory size of each application by monitoring. The second approach, complementary to static analysis, is a testing-based approach. Indeed, such an approach provides lower bounds on the required memory. The last approach is type checking which can be done by a bytecode verifier. Our own approach is rather different and consists in an attempt to control resources by providing resource certificates in such a way that the compiled code is safe w.r.t. memory overflow. Similar works have been studied by Hofmann [Hofmann 1999; 2000] and Aspinall and Compagnoni [Aspinall and Compagnoni 2003].

Sup-interpretations are closely related to the works of Niggl, Wunderlich [Niggl and Wunderlich 2006] and Jones, Kristiansen [Jones and Kristiansen 2005] and are strongly inspired by:

—The notion of quasi-interpretation which was introduced by Marion in [Marion 2003] and studied by Bonfante, Marion and Moyen in [Marion and Moyen 2000; Bonfante et al. 2001]. A quasi-interpretation, like a sup-interpretation, provides an upper bound on function outputs by static analysis of first order functional programs and allows the programmer to find a bound on the size of every stack frame. The paper [Bonfante et al. 2007] is a comprehensive introduction to quasi-interpretations which, combined with recursive path orderings, allow to characterize complexity classes such as the set of polynomial time functions as well as the set of polynomial space functions. Like quasi-interpretations, sup-interpretations were developed with a view of paying more attention to the algorithmic aspects of complexity than to the functional (or extensional) one. But the main interest of sup-interpretations is to capture a larger class of algorithms. In fact, programs computing logarithm or division admit a sup-interpretation but have no quasi-interpretation. Consequently, we firmly believe that sup-interpretations, like quasi-interpretations, could be applied to other languages such as resource bytecode verifier by following the lines of [Amadio et al. 2004] or language with synchronous cooperative threads as in [Amadio and Dal-Zilio 2004].

—The dependency pair method by Arts and Giesl in [Arts and Giesl 2000] which was initially introduced for proving termination of term rewriting systems automatically.

—The size-change principle by Jones et al. [Lee et al. 2001] which is another method developed for proving program termination. Indeed, there is a very strong relation between termination and computational complexity since, in order to prove termination and to find complexity bounds, we need to control the arguments occurring in the recursive calls of a program.

Section 2 introduces the first order functional language and its semantics. Section 3 introduces the syntactical notion of fraternity which is of real interest to control the size of values added by the recursive calls. Section 4 defines the main notions

of sup-interpretation and weight used to bound the size of program outputs. In section 5, we introduce three polynomial criteria:

(1) The first criterion is called the *quasi-friendly criterion*. It is an improvement of a previous criterion suggested in [Marion and Péchoux 2006]. The quasi-friendly criterion allows to capture a broad class of programs as we shall illustrate. Roughly speaking, a program which admits a polynomial sup-interpretation computes only values of polynomial size.

(2) The second criterion is called *quasi-friendly criterion with bounded recursive calls*. It allows to consider non-terminating programs. This criterion provides a polynomial bound on the size of the values computed during the execution of a program. Particularly, we can consider programs over infinite stream data and check that every step of the computation is polynomially bounded.

(3) Finally, the last criterion is called *quasi-friendly modulo projection criterion* and allows to deal with programs using particular destructive operations or functions. In practice, such a criterion captures a lot of divide-and-conquer programs, like the quicksort algorithm, that were not captured by the quasi-friendly criterion.

In the last three sections, we compare the notion of sup-interpretation with:

—the notion of quasi-interpretation. First, we show that any quasi-interpretation is a particular sup-interpretation. Since the synthesis of quasi-interpretations was shown to be decidable in [Bonfante et al. 2005], if we consider the set of **Max-Poly** functions defined to be constant functions, projections, max, $+$, $\times$ and closed by composition, we obtain heuristics for the synthesis of sup-interpretations, which consists in finding a quasi-interpretation of a given program. Finally, using former results about quasi-interpretations, we give two characterizations of the sets of functions computable in polynomial time and respectively polynomial space.

—the dependency pair method. We derive a termination criterion from the dependency pair method. This termination criterion only uses assignments over natural numbers in order to preserve the well-foundedness. Combined with the quasi-friendly criterion of the previous section, it allows to characterize the set of functions computable in polynomial space, in a distinct manner.

—the size-change principle to obtain a new termination criterion. The programs whose termination is captured by the size change principle are captured by this criterion but the converse is not true.

2. FIRST ORDER FUNCTIONAL PROGRAMMING

2.1 Syntax of programs

In this paper, we consider a generic first order functional programming language. The vocabulary $\Sigma = \langle Var, Cns, Op, Fct \rangle$ is composed of four disjoint domains of symbols which represent respectively the set of variables, the set of constructor symbols, the set of basic operator symbols and the set of function symbols. The arity of a symbol is the number n of its arguments. A program $\mathbf{p}$ of our language

is composed by a sequence of definitions $def_1, \cdots, def_m$ which are function symbol definitions and which are characterized by the following grammar:

$$\begin{array}{lll} \texttt{Definitions} \ni def & ::= & \mathtt{f}(x_1, \cdots, x_n) = \mathbf{Case}\ x_1, \cdots, x_n\ \mathbf{of}\ \overline{p_1} \to e^1 \ldots \overline{p_\ell} \to e^\ell \\ \texttt{Expression} \ni e & ::= & x \mid \mathbf{c}(e_1, \cdots, e_n) \mid \mathbf{op}(e_1, \cdots, e_n) \mid \mathtt{f}(e_1, \cdots, e_n) \\ \texttt{Patterns} \ni p & ::= & x \mid \mathbf{c}(p_1, \cdots, p_n) \\ \texttt{Values} \ni v & ::= & \mathbf{c} \mid \mathbf{c}(v_1, \cdots, v_n) \end{array}$$

where $x, x_1, \ldots, x_n$ are variables, $p_1, \cdots, p_n$ are patterns, $v_1, \cdots, v_n$ are values, $e_1, \cdots, e_n, e^1, \ldots, e^l$ are expressions, $\mathbf{c} \in Cns$ is a constructor symbol, $\mathbf{op} \in Op$ is an operator symbol, $\mathtt{f} \in Fct$ is a function symbol and $\overline{p_i}$ is a sequence of n patterns. Throughout the paper, we use the notation $\overline{e}$ for any sequence of expressions $e_1, \ldots, e_n$, for some n clearly determined by the context.

The **Case** operator is a special symbol which allows pattern matching. In a definition of the shape $\mathtt{f}(x_1, \cdots, x_n) = \mathbf{Case}\ x_1, \cdots, x_n\ \mathbf{of}\ \overline{p_1} \to e^1 \ldots \overline{p_\ell} \to e^\ell$, a variable of e^j is a variable appearing in the sequence of patterns $\overline{p_j}$. In a **Case** expression, patterns are not overlapping and patterns variable are used linearly. Such restrictions ensure that programs are confluent [Huet 1980].

2.2 Semantics

The computational domain of a program $\mathbf{p}$ is $\mathcal{V}^{\mathbf{Err}} = \mathcal{V} \cup \{\mathbf{Err}\}$, where $\mathcal{V}$ represents the set of values `Values` defined above and **Err** is a special constructor symbol of arity 0 returned by the program when an error occurs. Each operator symbol **op** of arity n is interpreted by a function $[\![\mathbf{op}]\!]$ from $\mathcal{V}^n$ to $\mathcal{V}^{\mathbf{Err}}$. Operators are essentially basic partial functions like destructors or characteristic functions of predicates like $=$. The destructor **hd** illustrates the purpose of **Err** when it satisfies $[\![\mathbf{hd}]\!](\mathbf{nil}) = \mathbf{Err}$.

The language has a call-by-value semantics which is displayed in Figure 1.

A substitution σ is a partial function from Var to $\mathcal{V}$. The application of a substitution σ to an expression e is noted $e\sigma$. Given a substitution σ and an expression e, the meaning of the judgement $e\sigma \downarrow w$ is that the expression $e\sigma$ evaluates to the value w of $\mathcal{V}^{\mathbf{Err}}$.

$$\frac{t_1 \downarrow w_1 \ldots t_n \downarrow w_n}{\mathbf{c}(t_1, \cdots, t_n) \downarrow \mathbf{c}(w_1, \cdots, w_n)}\ \mathbf{c} \in Cns \text{ and } \forall i, w_i \neq \mathbf{Err}$$

$$\frac{t_1 \downarrow w_1 \ldots t_n \downarrow w_n}{\mathbf{op}(t_1, \cdots, t_n) \downarrow [\![\mathbf{op}]\!](w_1, \cdots, w_n)}\ \mathbf{op} \in Op \text{ and } \forall i, w_i \neq \mathbf{Err}$$

$$\frac{\overline{e} \downarrow \overline{w} \quad \mathtt{f}(\overline{x}) = \mathbf{Case}\ \overline{x}\ \mathbf{of}\ \overline{p_1} \to e^1 \ldots \overline{p_\ell} \to e^\ell \quad \exists \sigma,\ i\ :\ \overline{p_i}\sigma = \overline{w} \quad e^i\sigma \downarrow u}{\mathtt{f}(\overline{e}) \downarrow u}$$

Fig. 1. Call-by-value semantics of a program $\mathbf{p}$

Definition 2.1. A function symbol $\mathtt{f}$ of arity n of a given program $\mathbf{p}$ computes a partial function $[\![\mathtt{f}]\!] : \mathcal{V}^n \to \mathcal{V}^{\mathbf{Err}}$ defined by: For all $v_i \in \mathcal{V}$, $[\![\mathtt{f}]\!](v_1, \cdots, v_n) = w$ iff $\mathtt{f}(v_1, \cdots, v_n) \downarrow w$.

We extend this notation to expressions by $[\![e]\!] = w$ iff $e \downarrow w$.

EXAMPLE 1 (DIVISION). *Consider the following definitions that encode the division:*

$$\begin{aligned}
\mathtt{minus}(x,y) &= \mathbf{Case}\ x,y\ \mathbf{of}\\
&\quad \mathbf{0}, z \to \mathbf{0}\\
&\quad \mathbf{S}(z), \mathbf{0} \to \mathbf{S}(z)\\
&\quad \mathbf{S}(u), \mathbf{S}(v) \to \mathtt{minus}(u,v)\\
\mathtt{q}(x,y) &= \mathbf{Case}\ x,y\ \mathbf{of}\\
&\quad \mathbf{0}, \mathbf{S}(z) \to \mathbf{0}\\
&\quad \mathbf{S}(z), \mathbf{S}(u) \to \mathbf{S}(\mathtt{q}(\mathtt{minus}(z,u), \mathbf{S}(u)))
\end{aligned}$$

Using the notation $\underline{n}$*, for* $\underbrace{\mathbf{S}(\ldots\mathbf{S}(\mathbf{0})\ldots)}_{n\ times\ \mathbf{S}}$*, we have:*

$$[\![\mathtt{q}]\!](\underline{n},\underline{m}) = \underline{\lceil n/m \rceil},\ for\ m > 0$$

2.3 Call-trees

A *context* is an expression $\mathsf{C}[\diamond_1, \cdots, \diamond_r]$ containing a single occurrence of each $\diamond_i$. We suppose that the $\diamond_i$'s are fresh variables which are not in Σ. The substitution of each $\diamond_i$ by an expression d_i is noted $\mathsf{C}[d_1, \cdots, d_r]$.

Definition 2.2. Assume that $\mathtt{f}(\overline{x}) = \mathbf{Case}\ \overline{x}\ \mathbf{of}\ \overline{p_1} \to e^1 \ldots \overline{p_\ell} \to e^\ell$ is a definition of a program. *An expression* d *is activated* by $\mathtt{f}(\overline{p_j})$ if there is a context with one hole $\mathsf{C}[\diamond]$ such that $e^j = \mathsf{C}[d]$.

This definition is convenient in order to predict the computational data flow involved. Indeed, an expression is activated by $\mathtt{f}(p_1, \cdots, p_n)$ when $\mathtt{f}(v_1, \cdots, v_n)$ is called and each v_i matches the corresponding pattern p_i. An expression d activated by $\mathtt{f}(p_1, \cdots, p_n)$ is **maximal** if there is no context $\mathsf{C}[\diamond]$, distinct from the empty context (i.e. $\mathsf{C}[\diamond] \neq \diamond$), such that $\mathsf{C}[d]$ is activated by $\mathtt{f}(p_1, \cdots, p_n)$.

EXAMPLE 2. *In the program of example 1, the expressions* $\mathbf{S}(\mathtt{q}(\mathtt{minus}(z,u), \mathbf{S}(u)))$ *and* $\mathtt{minus}(z,u)$ *are activated by* $\mathtt{q}(\mathbf{S}(z), \mathbf{S}(u))$*. However,* $\mathbf{S}(\mathtt{q}(\mathtt{minus}(z,u), \mathbf{S}(u)))$ *is the only maximal expression activated by* $\mathtt{q}(\mathbf{S}(z), \mathbf{S}(u))$*.*

Now we define the notion of call-tree which corresponds to the tree of function calls generated by one execution of a program.

A *state* is a tuple $\langle \mathtt{f}, u_1, \cdots, u_n \rangle$ where $\mathtt{f}$ is a function symbol of arity n and $u_1, \ldots, u_n$ are values. Assume that $\eta_1 = \langle \mathtt{f}, u_1, \cdots, u_n \rangle$ and $\eta_2 = \langle \mathtt{g}, v_1, \cdots, v_k \rangle$ are two states. Assume also that $\mathsf{C}[\mathtt{g}(e_1, \cdots, e_k)]$ is activated by $\mathtt{f}(p_1, \cdots, p_n)$. There is a *transition* between two states η_1 and η_2, noted $\eta_1 \rightsquigarrow \eta_2$, if there is a substitution σ such that:

(1) $p_i\sigma = u_i$, for $i = 1, \ldots, n$.
(2) and $[\![e_j\sigma]\!] = v_j$, for $j = 1, \ldots, k$.

We write $\overset{*}{\rightsquigarrow}$ to denote the reflexive and transitive closure of $\rightsquigarrow$. The call-tree of $\mathbf{p}$ of root $\langle \mathtt{f}, u_1, \cdots, u_n \rangle$ is the tree defined by:

—the root is the node labeled by the state $\langle \mathtt{f}, u_1, \cdots, u_n\rangle$.

—the nodes are labeled by states of $\{\eta \mid \langle \mathtt{f}, u_1, \cdots, u_n\rangle \stackrel{*}{\rightsquigarrow} \eta\}$,

—there is an edge between two nodes η_1 and η_2 if there is a transition between both states which label the nodes (i.e. $\eta_1 \rightsquigarrow \eta_2$).

Notice that a call-tree may be infinite if it corresponds to a non-terminating call. A state may be seen as a stack frame since it contains a function call and its respective arguments. A call-tree of root $\langle \mathtt{f}, u_1, \cdots, u_n\rangle$ represents all the stack frames which will be pushed on the stack when we compute $\mathtt{f}(u_1, \ldots, u_n)$.

3. FRATERNITIES

In this section, we define the notion of fraternity inspired by two termination techniques, the dependency pairs by Arts and Giesl [Arts and Giesl 2000] and the size-change principle by Jones et al [Lee et al. 2001]. Fraternity is a key notion used to control the size of the arguments in a recursive call.

Definition 3.1. (Precedence) The notion of activated expressions provides a precedence $\geq_{Fct}$ on function symbols. Indeed, set $\mathtt{f} \geq_{Fct} \mathtt{g}$ if there are $\overline{e}$ and $\overline{p}$ such that $\mathtt{g}(\overline{e})$ is activated by $\mathtt{f}(\overline{p})$. Then, take the reflexive and transitive closure of $\geq_{Fct}$, that we also note $\geq_{Fct}$. It is not difficult to establish that $\geq_{Fct}$ is a preorder. Next, say that $\mathtt{f} \approx_{Fct} \mathtt{g}$ if $\mathtt{f} \geq_{Fct} \mathtt{g}$ and, inversely, $\mathtt{g} \geq_{Fct} \mathtt{f}$. Lastly, $\mathtt{f} >_{Fct} \mathtt{g}$ if $\mathtt{f} \geq_{Fct} \mathtt{g}$ and $\mathtt{g} \geq_{Fct} \mathtt{f}$ does not hold. Intuitively, $\mathtt{f} \geq_{Fct} \mathtt{g}$ means that $\mathtt{f}$ calls $\mathtt{g}$ in some executions. And $\mathtt{f} \approx_{Fct} \mathtt{g}$ means that $\mathtt{f}$ and $\mathtt{g}$ call each other recursively.

Definition 3.2. (Fraternity) In a program $\mathbf{p}$, an expression $\mathsf{C}[\mathtt{g}_1(\overline{e_1}), \ldots, \mathtt{g}_r(\overline{e_r})]$ activated by $\mathtt{f}(p_1, \cdots, p_n)$ is a fraternity if:

(1) $\mathsf{C}[\mathtt{g}_1(\overline{e_1}), \ldots, \mathtt{g}_r(\overline{e_r})]$ is a maximal expression.
(2) For each $i \in \{1, r\}$, $\mathtt{g}_i \approx_{Fct} \mathtt{f}$.
(3) For every function symbol $\mathtt{h}$ that appears in the context $\mathsf{C}[\diamond_1, \cdots, \diamond_r]$, we have $\mathtt{f} >_{Fct} \mathtt{h}$.

A fraternity may correspond to a recursive call since it involves function symbols that are equivalent for the precedence $\geq_{Fct}$.

Example 3. *The program of example 1 admits two fraternities. The first fraternity is* $\mathtt{minus}(u, v)$ *which is activated by* $\mathtt{minus}(\mathbf{S}(u), \mathbf{S}(v))$ *and the second one is* $\mathbf{S}(\mathtt{q}(\mathtt{minus}(z, u), \mathbf{S}(u)))$ *which is activated by* $\mathtt{q}(\mathbf{S}(z), \mathbf{S}(u))$.

4. SUP-INTERPRETATIONS AND WEIGHTS

4.1 Partial assignments

Definition 4.1. A partial assignment I is a partial mapping from the vocabulary Σ which assigns a function $I(b) : (\mathbb{R}^+)^m \longmapsto \mathbb{R}^+$ to each symbol b of arity m in the domain of I. The domain of a partial assignment I is noted $\mathrm{dom}(I)$. Because it is convenient, we shall always assume that partial assignments are defined on constructor symbols and operators (i.e. $\{\mathbf{Err}\} \cup \mathit{Cns} \cup \mathit{Op} \subseteq \mathrm{dom}(I)$).

An assignment I is defined over an expression e if each symbol of $\mathit{Cns} \cup \mathit{Op} \cup \mathit{Fct}$ in e belongs to $\mathrm{dom}(I)$. Suppose that the assignment I is defined over an expression

e with n variables. The partial assignment of e w.r.t. I, that we note $I^*(e)$, is the canonical extension of the assignment I and denotes a function from $(\mathbb{R}^+)^n$ to $\mathbb{R}^+$ defined as follows:

(1) If x_i is in Var, let $I^*(x_i) = X_i$, with $X_1, \ldots, X_n$ a sequence of new variables ranging over $\mathbb{R}^+$.
(2) If b is a 0-ary symbol, then $I^*(b) = I(b)$.
(3) If b is a symbol of arity $m > 0$ and $e_1, \cdots, e_m$ are expressions, then we have $I^*(b(e_1, \cdots, e_m)) = I(b)(I^*(e_1), \ldots, I^*(e_m))$

The notion of assignment is extended in a natural way to contexts. The assignment of a context $\mathsf{C}[\diamond_1, \cdots, \diamond_l]$ is a function $I^*(\mathsf{C})$ from $(\mathbb{R}^+)^l$ to $\mathbb{R}^+$ such that for any expressions $e_1, \cdots, e_l$, we have $I^*(\mathsf{C})(I^*(e_1), \ldots, I^*(e_l)) = I^*(\mathsf{C}[e_1, \cdots, e_l])$. If $\overline{e}$ is a sequence of expressions $e_1, \cdots, e_m$ then we will use the notation $I^*(\overline{e})$ in order to represent the sequence $I^*(e_1), \ldots, I^*(e_m)$.

Definition 4.2. Given a semiring $\mathbb{K}$, let **Max-Poly** $\{\mathbb{K}\}$ be the set of functions defined to be constant functions in $\mathbb{K}$, projections, max, $+$, $\times$ and closed by composition. An assignment I is said to be max-polynomial in $\mathbb{K}$ if for every symbol b such that $I(b)$ is defined, $I(b)$ is a function of **Max-Poly** $\{\mathbb{K}\}$.

Definition 4.3. (Polynomial assignments) A partial assignment I is *polynomial* if for each symbol b of $\mathrm{dom}(I)$, $I(b)$ is in **Max-poly** $\{\mathbb{R}^+\}$.

Definition 4.4. (Additive assignments) An assignment of a symbol b is *additive* if:

$$I(b)(X_1, \cdots, X_n) = \sum_{i=1}^{n} X_i + \alpha_b \text{ where } \alpha_b \geq 1 \qquad \text{if } b \text{ is of arity } n > 0$$
$$I(b) = 0 \qquad \text{otherwise}$$

An assignment is additive if the assignment of each constructor symbol is additive.

Definition 4.5. The size of an expression e is noted $|e|$ and defined by $|e| = 0$, if e is a 0-ary symbol, and $|b(e_1, \ldots, e_n)| = 1 + \sum_i |e_i|$, if $e = b(e_1, \ldots, e_n)$ with $n > 0$.

LEMMA 4.6. *Given an additive assignment I, there is a constant α such that for each value v of $\mathcal{V}^{\mathbf{Err}}$, the following inequality is satisfied:*

$$|v| \leq I^*(v) \leq \alpha \times |v|$$

PROOF. Define $\alpha = \max_{\mathbf{c} \in Cns}(\alpha_{\mathbf{c}})$ where $\alpha_{\mathbf{c}}$ is taken to be the constant of definition 4.4, if $\mathbf{c}$ is of strictly positive arity, and $\alpha_{\mathbf{c}}$ is equal to the constant $I^*(\mathbf{c})$ otherwise. The inequalities follow directly by induction on the size of a value. □

4.2 Sup-interpretations

Definition 4.7. A sup-interpretation is a partial assignment θ which verifies the three conditions below:

(1) The assignment θ is weakly monotonic. That is, for each symbol $b \in \mathrm{dom}(\theta)$, the function $\theta(b)$ satisfies:

$$\forall i = 1, \ldots, n \; X_i \geq Y_i \Rightarrow \theta(b)(X_1, \cdots, X_n) \geq \theta(b)(Y_1, \cdots, Y_n)$$

(2) For each value v of the computational domain $\mathcal{V}^{\mathbf{Err}}$, the sup-interpretation of v is greater than the size of v:

$$\theta^*(v) \geq |v|$$

(3) For each symbol $b \in \mathrm{dom}(\theta)$ of arity n and for each value $v_1, \ldots, v_n$ of $\mathcal{V}$, if $[\![b]\!](v_1, \ldots, v_n) \in \mathcal{V}^{\mathbf{Err}}$, then

$$\theta^*(b(v_1, \ldots, v_n)) \geq \theta^*([\![b]\!](v_1, \ldots, v_n))$$

An expression e admits a sup-interpretation θ if the sup-interpretation θ is an assignment defined over e. The sup-interpretation of e with respect to θ is $\theta^*(e)$.

Intuitively, a sup-interpretation is a special program interpretation. Instead of yielding the program denotation, a sup-interpretation provides an upper bound on the output size of the function denoted by the program. It is worth noticing that a sup-interpretation is a complexity measure in the sense of Blum [Blum 1967].

Remark 4.8. If a sup-interpretation θ is an additive assignment then Condition 2 of Definition 4.7 always holds by Lemma 4.6.

Example 4. *The program of example 1 admits the following sup-interpretation in* ***Max-poly***$\{\mathbb{R}^+\}$*:*

$$\begin{aligned} \theta(\mathbf{0}) &= 0 \\ \theta(\mathbf{S})(X) &= X + 1 \\ \theta(\mathtt{minus})(X, Y) &= X \\ \theta(\mathtt{q})(X, Y) &= X \end{aligned}$$

Example 5. *Consider the program for exponential:*

$$\begin{aligned} \mathtt{exp}(x) \ &= \mathbf{Case}\ x\ \mathbf{of} \\ &\quad \mathbf{0} \to \mathbf{S}(\mathbf{0}) \\ &\quad \mathbf{S}(y) \to \mathtt{double}(\mathtt{exp}(y)) \\ \mathtt{double}(x) \ &= \mathbf{Case}\ x\ \mathbf{of} \\ &\quad \mathbf{0} \to \mathbf{0} \\ &\quad \mathbf{S}(y) \to \mathbf{S}(\mathbf{S}(\mathtt{double}(y))) \end{aligned}$$

By taking $\theta(\mathbf{0}) = 0$, $\theta(\mathbf{S})(X) = X+1$, $\theta(\mathtt{double})(X) = 2 \times X$ *and* $\theta(\mathtt{exp})(X) = 2^X$, *we define a sup-interpretation of the function symbols* `double` *and* `exp` *which is not in* ***Max-poly***$\{\mathbb{R}^+\}$. *Indeed, it is routine to check the 3 conditions of definition 4.7. For example,* $\forall \underline{n} \in \mathcal{V}$ *we have* $\theta^*(\mathtt{double}(\underline{n})) \geq \theta^*([\![\mathtt{double}]\!](\underline{n}))$ *since:*

$$\begin{aligned} \theta^*(\mathtt{double}(\underline{n})) &= \theta(\mathtt{double})(\theta^*(\underline{n})) = 2 \times \theta^*(\underline{n}) = 2 \times n \\ \theta^*([\![\mathtt{double}]\!](\underline{n})) &= \theta^*(\underline{2 \times n}) = 2 \times n \end{aligned}$$

Lemma 4.9. *Let* e *be an expression with no variable and which admits a sup-interpretation* θ. *Assume that* $[\![e]\!] \in \mathcal{V}^{\mathbf{Err}}$ *then we have:*

$$\theta^*([\![e]\!]) \leq \theta^*(e)$$

PROOF. The proof is done by structural induction on expressions. The base case is when e is a constant constructor symbol. We have $[\![e]\!] = e$ and, consequently, $\theta^*([\![e]\!]) = \theta^*(e)$.

Take an expression $e = \mathtt{f}(e_1, \cdots, e_n)$ that has a sup-interpretation θ. By induction hypothesis (IH), we have $\theta^*(e_i) \geq \theta^*([\![e_i]\!])$. Now,

$$\begin{aligned}
\theta^*(e) &= \theta(\mathtt{f})(\theta^*(e_1), ..., \theta^*(e_n)) && \text{by definition of } \theta^* \\
&\geq \theta(\mathtt{f})(\theta^*([\![e_1]\!]), ..., \theta^*([\![e_n]\!])) && \text{by 1 of Dfn 4.7 and (IH)} \\
&= \theta^*(\mathtt{f}([\![e_1]\!], ..., [\![e_n]\!])) && \text{by definition of } \theta^* \\
&\geq \theta^*([\![\mathtt{f}]\!]([\![e_1]\!], ..., [\![e_n]\!])) && \text{by 3 of Dfn 4.7} \\
&= \theta^*([\![e]\!])
\end{aligned}$$

□

Given an expression e, we define $\|e\|$ by:

$$\|e\| = \begin{cases} |[\![e]\!]| & \text{if } [\![e]\!] \in \mathcal{V}^{\mathbf{Err}} \\ 0 & \text{otherwise} \end{cases}$$

Hence we can consider non-terminating programs smoothly:

COROLLARY 4.10. *If e is an expression with no variable and which admits a sup-interpretation θ then we have:*

$$\|e\| \leq \theta^*(e)$$

PROOF. The case where $[\![e]\!] \notin \mathcal{V}^{\mathbf{Err}}$ is trivial. Now assume that $[\![e]\!] \in \mathcal{V}^{\mathbf{Err}}$.

$$\begin{aligned}
\theta^*(e) &\geq \theta^*([\![e]\!]) && \text{by Lemma 4.9} \\
&\geq \|e\| && \text{by Condition 2 of Dfn 4.7}
\end{aligned}$$

□

4.3 Weights

Now we are going to define the notion of weight which allows to control the size of the arguments in recursive calls.

Definition 4.11. A weight ω is a partial assignment which ranges over *Fct*. To a given function symbol $\mathtt{f}$ of arity n, it assigns a total function $\omega_{\mathtt{f}}$ from $(\mathbb{R}^+)^n$ to $\mathbb{R}^+$ which satisfies:

(1) $\omega_{\mathtt{f}}$ is weakly monotonic.

$$\forall i = 1, \ldots, n,\ X_i \geq Y_i \Rightarrow \omega_{\mathtt{f}}(\ldots, X_i, \ldots) \geq \omega_{\mathtt{f}}(\ldots, Y_i, \ldots)$$

(2) $\omega_{\mathtt{f}}$ has the subterm property

$$\forall i = 1, \ldots, n,\ \forall X_i \in \mathbb{R}^+\ \omega_{\mathtt{f}}(\ldots, X_i, \ldots) \geq X_i$$

5. CRITERIA TO CONTROL SPACE RESOURCES

In this section, we introduce distinct criteria combining polynomial sup-interpretations and weights. These criteria allow to bound the size of the values computed

by a program polynomially in the size of the inputs. The main criterion is called quasi-friendly criterion. It is inspired by the friendly criterion developed in a former paper [Marion and Péchoux 2006]. However the quasi-friendly criterion captures more algorithms than this former one. For example, recursion on tree data structure of the shape $\mathtt{f}(x) = \mathbf{Case}\ x\ \mathbf{of}\ t * t' \to \mathtt{f}(t) * \mathtt{f}(t')$ is captured by quasi-friendly programs whereas it is not captured by friendly programs.

5.1 Quasi-friendly criterion

Definition 5.1. A program $\mathbf{p}$ is *quasi-friendly* iff there are a polynomial and additive sup-interpretation θ and a polynomial weight ω such that for each fraternity $\mathsf{C}[\mathtt{g}_1(\overline{e_1}), \ldots, \mathtt{g}_r(\overline{e_r})]$ of $\mathbf{p}$, activated by $\mathtt{f}(p_1, \cdots, p_n)$, we have:

$$\omega_{\mathtt{f}}(\theta^*(p_1), \ldots, \theta^*(p_n)) \geq \theta^*(\mathsf{C})(\omega_{\mathtt{g}_1}(\theta^*(\overline{e_1})), \ldots, \omega_{\mathtt{g}_r}(\theta^*(\overline{e_r})))$$

Remark 5.2. Notice that nested recursive calls are not of real interest for this criterion. In fact, consider for example the following definition $\mathtt{f}(x) = \mathbf{Case}\ x\ \mathbf{of}\ x \to \mathtt{f}(\mathtt{f}(x))$. In order to check the quasi-friendly criterion, one needs to find a weight and a sup-interpretation for the function symbol $\mathtt{f}$ satisfying:

$$\omega_{\mathtt{f}}(X) \geq \omega_{\mathtt{f}}(\theta(\mathtt{f})(X))$$

It means that we already know a bound on the computation of the function symbol $\mathtt{f}$. Consequently, the criterion becomes useless. However, this is not a severe drawback since such programs are not that natural in a programming perspective and either they have to be really restricted or they rapidly generate complex functions like Ackermann's one.

THEOREM 5.3. *Assume that $\mathbf{p}$ is a quasi-friendly program, then for each function symbol $\mathtt{f}$ of $\mathbf{p}$, there is a polynomial $P_{\mathtt{f}}$ such that for every values $v_1, \ldots, v_n$,*

$$\|\mathtt{f}(v_1, \ldots, v_n)\| \leq P_{\mathtt{f}}(\max(|v_1|, ..., |v_n|))$$

PROOF. Suppose that we have a program $\mathbf{p}$, a function symbol $\mathtt{f} \in Fct$ and $v_1, \cdots, v_n \in \mathcal{V}$ such that $[\![\mathtt{f}]\!](v_1, \cdots, v_n)$ is defined (i.e. the function computation terminates on inputs $v_1, \cdots, v_n$).

We assign to each pattern p a max-polynomial $P'_p(X)$ in one variable X as follows:

—if p is a variable then $P'_p(X) = X$
—if $p = \mathbf{c}(p_1, \cdots, p_n)$ then $P'_p(X) = n \times \max_{i=1..n}(P'_{p_i}(X)) + 1$

By construction, if p is a pattern with n variables $x_1, \cdots, x_n$ then for each substitution σ such that $x_i\sigma = v_i$ we have:

$$P'_p(\max(|v_1|, \cdots, |v_n|)) \geq |p\sigma| = \|p\sigma\| \tag{1}$$

We are going to show the result by an induction on the precedence $\geq_{Fct}$.

—If the function symbol $\mathtt{f}$ is defined without fraternities, then we have a definition of this shape $\mathtt{f}(x_1, \cdots, x_n) = \mathbf{Case}\ x_1, \cdots, x_n\ \mathbf{of}\ \overline{p^1} \to e_1 \ldots \overline{p^l} \to e_l$ with $\mathtt{f} >_{Fct} \mathtt{g}$ for all function symbols $\mathtt{g} \in e_j$, $j \in \{1, l\}$. Suppose, by induction hypothesis, that we have already defined a polynomial upper bound $P_{\mathtt{g}}$ on every function symbol $\mathtt{g}$ s.t. $\mathtt{f} >_{Fct} \mathtt{g}$. If $e_j = \mathtt{h}(d_1, \cdots, d_m)$, we define inductively a polynomial upper bound on the size of the computation of e_j by

$P_{e_j}(X) = P_{\mathtt{h}}(\max_{i=1..m} P_{d_i}(X))$ and we take $P_{\mathtt{f}}(X) = \max_{j=1..l}(P_{e_j}(X))$. By construction, we obtain that $P_{\mathtt{f}}(\max_{i=1..n} |v_i|) \geq \|\mathtt{f}(v_1, \cdots, v_n)\|$ because of the induction hypothesis combined with (1).

—Now, suppose that the function symbol $\mathtt{f}$ is defined by some fraternities. Let E be the set of the maximal expressions activated by $\mathtt{f}(p_1, \cdots, p_n)$, for some patterns $p_1, \cdots, p_n$, and which are not a fraternity. For every expression $e \in E$, we first define the polynomial P_e, as in the previous case. Then, we define the polynomial $P_{\mathtt{f}_{>_{Fct}}}$ by $P_{\mathtt{f}_{>_{Fct}}}(X) = \max_{e \in E}(P_e(X))$. For each $\mathtt{g} \approx_{Fct} \mathtt{f}$, we also define $P_{\mathtt{g}_{>_{Fct}}}$ in the same fashion. Finally, we define a new polynomial $Q_{\mathtt{f}}(X) = \max_{\mathtt{g} \approx_{Fct} \mathtt{f}}(P_{\mathtt{g}_{>_{Fct}}}(X))$. Intuitively, this polynomial is an upper bound on the size of every value computed by a definition which will leave a recursive call, that is a definition of a function symbol that calls function symbols strictly smaller for the precedence.

Now, combining the inequalities of the quasi-friendly criterion, we establish that if, for some values $v_1, \cdots, v_n$, $\mathtt{f}(v_1, \cdots, v_n) \xrightarrow{*} \mathsf{C}^*[\mathtt{g}_1(\overline{u_1}), \ldots, \mathtt{g}_r(\overline{u_r})]$, with $\mathtt{g}_1 \approx_{Fct} \ldots \approx_{Fct} \mathtt{g}_r \approx_{Fct} \mathtt{f}$ and where $\rightarrow$ is the rewrite relation induced by the definitions of the program, then:

$$\omega_{\mathtt{f}}(\theta^*(v_1), \cdots, \theta^*(v_n)) \geq \theta^*(\mathsf{C}^*)(\omega_{\mathtt{g}_1}(\theta^*(\overline{u_1})), \ldots, \omega_{\mathtt{g}_r}(\theta^*(\overline{u_r}))) \tag{2}$$

This result can be shown by induction on the number k of reduction steps corresponding to the evaluation of function symbols equivalent to $\mathtt{f}$. For $k = 1$, it corresponds to the quasi-friendly criterion. Now suppose that it holds for $k > 1$, that is for a reduction of the shape $\mathtt{f}(\overline{v}) \xrightarrow{k} \mathsf{E}[\mathtt{g}_1(\overline{e_1}), \ldots, \mathtt{g}_r(\overline{e_r})]$. Moreover, suppose, without restriction, that $\overline{[\![e_j]\!]} = \overline{u_j}$, for all $j \in \{1, r\}$. We obtain that $\mathtt{f}(\overline{v}) \xrightarrow{k} \mathsf{E}[\mathtt{g}_1(\overline{u_1}), \ldots, \mathtt{g}_r(\overline{u_r})]$ since the evaluation of e_j involves function symbols strictly smaller than $\mathtt{f}$ for the precedence. Suppose, with respect to our evaluation strategy, that the next rule applied is of the shape $\mathtt{g}_j(\overline{u_j}) \xrightarrow{1} \mathsf{D}[\mathtt{h}_1(d_1), \ldots, \mathtt{h}_m(d_m)]$, with $\mathtt{h}_i \approx_{Fct} \mathtt{g}_j$ for all $i \in \{1, m\}$, hence we can apply the quasi-friendly criterion. Finally, by monotonicity of sup-interpretations, we obtain:

$$\begin{aligned}
\omega_{\mathtt{f}}(\theta^*(v_1), \cdots, \theta^*(v_n)) &\geq \theta^*(\mathsf{E})(\omega_{\mathtt{g}_1}(\theta^*(\overline{e_1})), \ldots, \omega_{\mathtt{g}_r}(\theta^*(\overline{e_r}))) && \text{By I.H.} \\
&\geq \theta^*(\mathsf{E})(\omega_{\mathtt{g}_1}(\theta^*(\overline{u_1})), \ldots, \omega_{\mathtt{g}_r}(\theta^*(\overline{u_r}))) && \text{By Lemma 4.9} \\
&\geq \theta^*(\mathsf{C}^*)(\omega_{\mathtt{f}_1}(\theta^*(\overline{b_1})), \ldots, \omega_{\mathtt{f}_s}(\theta^*(\overline{b_s}))) && \text{By Dfn 5.1}
\end{aligned}$$

where $\mathtt{f}(v_1, \cdots, v_n) \xrightarrow{k+1} \mathsf{C}^*[\mathtt{f}_1(b_1), \ldots, \mathtt{f}_s(b_s)]$ with $s = r + m - 1$, $\mathsf{C}^*[\diamond_1, \ldots \diamond_s]$ $= \mathsf{E}[\diamond_1, \ldots, \diamond_{j-1}, \mathsf{D}[\diamond_j, \ldots, \diamond_{j+m-1}], \diamond_{j+m}, \ldots, \diamond_s]$ and such that $\mathtt{f}_i(b_i)$ is equal to $\mathtt{g}_i(u_i)$, $\mathtt{h}_{i-j+1}(d_{i+j-1})$ or $\mathtt{g}_{i+1-m}(u_{i+1-m})$ depending on whether i is in $\{1, j-1\}$, $\{j, j+m-1\}$ or $\{j+m, s\}$.

This result holds particularly in the case where the $\mathtt{f}_i(\overline{b_i})$ calls correspond to function calls that will leave the recursive call (i.e. function symbols that call function symbols strictly smaller for the precedence). Since we are considering defined values (i.e. evaluations that terminate), such calls exist.

Define $P(\overline{X}) = \alpha \times Q_{\mathtt{f}}(\max(\overline{X}))$, with α the constant of Lemma 4.6. For all

$i \in \{1, s\}$, $\mathtt{f}_i(\overline{b_i})$ is terminating and, if the b_i are values, we have:

$$\begin{array}{lll} P(\theta^*(\overline{u_i})) \geq P(|\overline{u_i}|) & & \text{By Condition 2 of Definition 4.7} \\ \geq \alpha \times |[\![\mathtt{f}_i]\!](\overline{b_i})| & & \text{By construction of } Q_{\mathtt{f}} \\ \geq \theta^*([\![\mathtt{f}_i]\!](\overline{b_i})) & & \text{By Lemma 4.6} \end{array}$$

Consequently, if $\mathtt{f}(v_1, \cdots, v_n) \xrightarrow{*} \mathsf{C}^*[\mathtt{f}_1(\overline{b_1}), \ldots, \mathtt{f}_s(\overline{b_s})]$ then we have:

$$\begin{array}{lll} \theta^*(\mathsf{C}^*)(P(\theta^*(\overline{b_1})), \ldots, P(\theta^*(\overline{b_s}))) & & \\ \geq \theta^*(\mathsf{C}^*)(\theta^*([\![\mathtt{f}_1]\!](\overline{b_1})), \ldots, \theta^*([\![\mathtt{f}_s]\!](\overline{b_s}))) & & \text{By monotonicity of } \theta^*(\mathsf{C}^*) \\ \geq \|\mathtt{f}(v_1, \cdots, v_n)\| & & \text{By Corollary 4.10} \end{array}$$

Now it remains to show that there is a function $R_{\mathtt{f}} \in \mathbf{Max\text{-}poly}\{\mathbb{R}^+\}$ such that $R_{\mathtt{f}}(\theta^*(\overline{v})) \geq \theta^*(\mathsf{C}^*)(P(\theta^*(\overline{b_1})), \ldots, P(\theta^*(\overline{b_s})))$. This is the case since inequality (2) implies that $\theta^*(\mathsf{C}^*)(\diamond_1, \cdots, \diamond_s)$ is polynomial in $\diamond_j$ because $\theta^*(\mathsf{C}^*)$ is bounded by a polynomial depending on $\omega_{\mathtt{f}}(\theta^*(v_1), \cdots, \theta^*(v_n))$ *independently of the derivation length*. We apply Lemma 4.6 again, obtaining that $\|\mathtt{f}(v_1, \cdots, v_n)\|$ is polynomially bounded by $P'_{\mathtt{f}}(\max |v_i|) = R_{\mathtt{f}}(\alpha \times \max_{i=1..n}(|v_i|))$.

Finally, $\forall \mathtt{f} \in \mathit{Fct}$, $P'_{\mathtt{f}} \in \mathbf{Max\text{-}poly}\{\mathbb{R}^+\}$ and we can find a polynomial $P_{\mathtt{f}}$ such that $\forall X\ P_{\mathtt{f}}(X) \geq P'_{\mathtt{f}}(X)$. □

Corollary 5.4. *Suppose that we have a quasi-friendly program which terminates on all inputs. Then for each function* $\mathtt{f}$ *there is a polynomial* $P_{\mathtt{f}}$ *such that for every values* $v_1, \cdots, v_n$*:*

$$|[\![\mathtt{f}]\!](v_1, \cdots, v_n)| \leq P_{\mathtt{f}}(\max(|v_1|, \ldots, |v_n|))$$

Example 6. *The program of example 1 is quasi-friendly. Taking:*

$$\begin{array}{rl|rl} \theta(\mathbf{S})(X) & = X + 1 & \omega_{\mathtt{q}}(X, Y) & = X + Y \\ \theta(\mathtt{minus})(X, Y) & = X & \omega_{\mathtt{minus}}(X, Y) & = X + Y \end{array}$$

We check that:

$$\begin{array}{rl} \omega_{\mathtt{minus}}(\theta^*(\mathbf{S}(v)), \theta^*(\mathbf{S}(u))) & = V + U + 2 \\ & \geq V + U \\ & = \omega_{\mathtt{minus}}(\theta^*(v), \theta^*(u)) \\ \omega_{\mathtt{q}}(\theta^*(\mathbf{S}(z)), \theta^*(\mathbf{S}(u))) & = U + Z + 2 \\ & = \theta^*(\mathbf{S})(\omega_{\mathtt{q}}(\theta^*(\mathtt{minus}(z, u)), \theta^*(\mathbf{S}(u)))) \end{array}$$

Example 7 (Gcd). *The following program computes the greatest common divi-*

sor:

$$
\begin{array}{l}
\mathtt{minus}(x,y) = \textbf{Case } x,y \textbf{ of} \\
\qquad \mathbf{0}, z \to \mathbf{0} \\
\qquad \mathbf{S}(z), \mathbf{0} \to \mathbf{S}(z) \\
\qquad \mathbf{S}(u), \mathbf{S}(v) \to \mathtt{minus}(u,v) \\
\mathtt{if}(x,y,z) = \textbf{Case } x,y,z \textbf{ of} \\
\qquad \mathbf{True}, u, v \to u \\
\qquad \mathbf{False}, u, v \to v \\
\mathtt{gcd}(x,y) = \textbf{Case } x,y \textbf{ of} \\
\qquad \mathbf{0}, z \to z \\
\qquad \mathbf{S}(z), \mathbf{0} \to \mathbf{S}(z) \\
\qquad \mathbf{S}(u), \mathbf{S}(v) \to \mathtt{if}(\mathbf{le}(u,v), \mathtt{gcd}(\mathtt{minus}(v,u), \mathbf{S}(u)), \mathtt{gcd}(\mathtt{minus}(u,v), \mathbf{S}(v)))
\end{array}
$$

le *is an operator which, given two inputs* n *and* m*, returns* **True** *(respectively* **False***) if the unary representation of* n *is smaller (strictly greater) than the one of* m*. Consequently,* $\theta(\mathbf{le})(X,Y) = 0$ *defines a sup-interpretation for* **le***.*

This program admits two fraternities $\mathtt{minus}(u,v)$ *and* $\mathtt{if}(\mathbf{le}(u,v), \mathtt{gcd}(\mathtt{minus}(v,u), \mathbf{S}(u)), \mathtt{gcd}(\mathtt{minus}(u,v), \mathbf{S}(v)))$*. The first one depends on* minus *and verifies the quasi-friendly criterion. The last one depends on* gcd *and is activated by* $\mathtt{gcd}(\mathbf{S}(u), \mathbf{S}(v))$*. By taking* $\theta(\mathbf{S})(X) = X + 1$*,* $\theta(\mathtt{if})(X,Y,Z) = \max(Y,Z)$ *and* $\theta(\mathtt{minus})(X,Y) = X$*, we only have to check that there is a polynomial weight* ω *such that:*

$$
\begin{array}{l}
\omega_{\mathtt{gcd}}(U+1, V+1) \\
= \omega_{\mathtt{gcd}}(\theta(\mathbf{S})(U), \theta(\mathbf{S})(V)) \\
\geq \theta(\mathtt{if})(\theta(\mathbf{le})(U,V), \theta(\mathtt{minus})((V,U), \theta(\mathbf{S})(U)), \omega_{\mathtt{gcd}}(\theta(\mathtt{minus})(U,V), \mathbf{S}(V))) \\
= \max(\omega_{\mathtt{gcd}}(V, U+1), \omega_{\mathtt{gcd}}(U, V+1))
\end{array}
$$

Taking $\omega_{\mathtt{gcd}}(X,Y) = X + Y$*, we can check that previous inequality becomes:*

$$U + V + 2 \geq V + U + 1$$

Consequently the program is quasi-friendly and Theorem 5.3 applies.

Example 8 (Huffman Coding Trees). *The following program computes the Huffman coding trees algorithm which can be found in [Bird and Wadler 1988]. The domain of computation is built from two constructor symbols,* **c** *for nodes and* **Tip** *for leaves, and three constructor symbols* **0**, **1** *and* **nil** *of arity 0. We first begin by the decoding function which, given a tree* t *and a path* p*, returns the word in* t

corresponding to the path p:

$$
\begin{array}{l}
\mathtt{decode}(t,p) = \textbf{Case } t,p \textbf{ of } t,p \to \mathtt{trace}(t,t,p)\\
\mathtt{trace}(x,y,z) = \textbf{Case } x,y,z \textbf{ of}\\
\qquad t,t',\mathbf{nil} \to \mathbf{nil}\\
\qquad t,\mathbf{c}(\mathbf{Tip}(x),t_2),\mathbf{c}(\mathbf{0},p) \to \mathbf{c}(\mathbf{Tip}(x),\mathtt{trace}(t,t,p))\\
\qquad t,\mathbf{c}(t_1,\mathbf{Tip}(x)),\mathbf{c}(\mathbf{1},p) \to \mathbf{c}(\mathbf{Tip}(x),\mathtt{trace}(t,t,p))\\
\qquad t,\mathbf{c}(\mathbf{c}(t_1,t_2),\mathbf{c}(t_3,t_4)),\mathbf{c}(\mathbf{0},p) \to \mathtt{trace}(t,\mathbf{c}(t_1,t_2),p)\\
\qquad t,\mathbf{c}(\mathbf{c}(t_1,t_2),\mathbf{c}(t_3,t_4)),\mathbf{c}(\mathbf{1},p) \to \mathtt{trace}(t,\mathbf{c}(t_3,t_4),p)
\end{array}
$$

Taking $\theta(\mathbf{0}) = \theta(\mathbf{1}) = 0$, $\theta(\mathbf{Tip})(X) = X+1$, $\theta(\mathbf{c})(X,Y) = X+Y+1$ *and* $\omega_{\mathtt{trace}}(X,Y,Z) = \max(X,Y)\times Z + \max(X,Y) + Z$, *we let the reader check that the condition of the quasi-friendly criterion is satisfied.*

Next we study the coding function returning the path in the tree t corresponding to a list of characters p given as input:

$$
\begin{array}{l}
\mathtt{codes}(t,p) = \textbf{Case } t,p \textbf{ of}\\
\qquad t,\mathbf{nil} \to \mathbf{nil}\\
\qquad t,\mathbf{c}(x,y) \to \mathbf{c}(\mathtt{code}(t,x),\mathtt{codes}(t,y))\\
\mathtt{code}(u,v) = \textbf{Case } u,v \textbf{ of}\\
\qquad \mathbf{Tip}(x),y \to \mathtt{if}(x{=}y,\mathbf{nil},\mathbf{Err})\\
\qquad \mathbf{c}(t_1,t_2),y \to \mathtt{if}(\mathtt{member}(y,t_1),\mathbf{c}(\mathbf{0},\mathtt{code}(t_1,y)),\\
\qquad\qquad\qquad \mathtt{if}(\mathtt{member}(y,t_2),\mathbf{c}(\mathbf{1},\mathtt{code}(t_2,y)),\mathbf{Err}))\\
\mathtt{member}(u,v) = \textbf{Case } u,v \textbf{ of}\\
\qquad x,\mathbf{Tip}(y) \to \mathtt{if}(x{=}y,\mathbf{True},\mathbf{False})\\
\qquad x,\mathbf{c}(t_1,t_2) \to \mathtt{or}(\mathtt{member}(x,t_1),\mathtt{member}(x,t_2))\\
\mathtt{if}(u,v,w) = \textbf{Case } u,v,w \textbf{ of}\\
\qquad \mathbf{True},x,y \to x\\
\qquad \mathbf{False},x,y \to y
\end{array}
$$

Notice that we have used two special operators, $=$ that tests whether two characters are equal and or *which computes the classical disjunction. Since this latter symbol returns only boolean values of size* 0, *we set its sup-interpretation to* $\theta(\mathtt{or})(X,Y) = 0$. *The function symbol* if *is quasi-friendly since it does not involve any recursive call and we take its sup-interpretation to be* $\theta(\mathtt{if})(X,Y,Z) = \max(X,Y,Z)$. *Consequently we can show that* member *is quasi-friendly. Since* member *returns a boolean value, we know that* $\theta(\mathtt{member})(X,Y) = 0$ *is a suitable sup-interpretation. Combining* $\theta(\mathbf{c})(X,Y) = X+Y+1$ *with* $\omega_{\mathtt{code}}(X,Y) = X+Y$, *we obtain that* code *is quasi-friendly. Now, we take* $\omega_{\mathtt{codes}}(X,Y) = (X+1)\times(Y+1)$. *Since* $\mathtt{code}(t,x)$ *is computing the path of x in the tree t, we know that* $\theta(\mathtt{code})(T,X) = T$ *is a suitable sup-interpretation. Now, we check the quasi-friendly criterion for the*

function codes*:*

$$\begin{aligned}\omega_{\mathtt{codes}}(\theta^*(t),\theta^*(\mathbf{c}(x,y))) &= (T+1)\times(X+Y+2)\\ &\geq (T+1)\times(Y+1)+T+1\\ &= \theta(\mathbf{c})(\theta^*(\mathtt{code}(t,x)),\omega_{\mathtt{codes}}(\theta^*(t),\theta^*(y)))\end{aligned}$$

Thus the program is quasi-friendly.

Now we describe the program that builds the Huffman tree. Given a list of pairs representing a character and a weight, the program first builds a list of tips, where the tips represent the pairs (here, the constructor symbol **Tip** *has arity 2 in order to combine characters and weights).Then, it combines the trees having the smallest weights into a new tree whose weight is the sum of its descendant weights. Finally, the program sorts the distinct trees by increasing weights, and goes into a recursive call until only one tree remains, the Huffman Tree. Notice that this program requires the input list to be already ordered by increasing weight.*

$\mathtt{single}(u) = \mathbf{Case}\ u\ \mathbf{of}$
$\quad \mathbf{nil} \to \mathbf{True}$
$\quad \mathbf{c}(p,\mathbf{nil}) \to \mathbf{True}$
$\quad \mathbf{c}(p,\mathbf{c}(q,l)) \to \mathbf{False}$
$\mathtt{head}(u) = \mathbf{Case}\ u\ \mathbf{of}$
$\quad \mathbf{c}(p,q) \to p$
$\mathtt{weight}(u) = \mathbf{Case}\ u\ \mathbf{of}$
$\quad \mathbf{Tip}(x,w) \to w$
$\quad \mathbf{c}(t_1,t_2) \to \mathtt{add}(\mathtt{weight}(t_1),\mathtt{weight}(t_2))$
$\mathtt{tiping}(u) = \mathbf{Case}\ u\ \mathbf{of}$
$\quad \mathbf{nil} \to \mathbf{nil}$
$\quad \mathbf{c}((x,w),p) \to \mathbf{c}(\mathbf{Tip}((x,w)),\mathtt{tiping}(p))$
$\mathtt{insert}(u,v) = \mathbf{Case}\ u,v\ \mathbf{of}$
$\quad p,\mathbf{nil} \to \mathbf{c}(p,\mathbf{nil})$
$\quad p,\mathbf{c}(q,r) \to \mathtt{if}(\mathbf{le}(\mathtt{weight}(p),\mathtt{weight}(q)),\mathbf{c}(p,\mathbf{c}(q,r)),\mathbf{c}(q,\mathtt{insert}(p,r)))$
$\mathtt{combine}(u) = \mathbf{Case}\ u\ \mathbf{of}$
$\quad p \to \mathtt{if}(\mathtt{single}(p),\mathtt{head}(p),\mathtt{combine}(p))$
$\quad \mathbf{c}(p,\mathbf{c}(q,l)) \to \mathtt{insert}(\mathbf{c}(p,q),l)$
$\mathtt{build}(p) = \mathbf{Case}\ p\ \mathbf{of}\ p \to \mathtt{combine}(\mathtt{tiping}(p))$

We can check that this program is quasi-friendly by taking

$$\omega_{\mathtt{combine}}(X) = \omega_{\mathtt{tiping}}(X) = \omega_{\mathtt{weight}}(X) = X \text{ and } \omega_{\mathtt{insert}}(X,Y) = X+Y$$
$$\theta(\mathtt{weight})(X) = \theta(\mathtt{head})(X) = X,\ \theta(\mathtt{add})(X,Y) = X+Y \text{ and } \theta(\mathtt{single})(X) = 0$$

EXAMPLE 9. *The program of example 5 is not quasi-friendly. Indeed, since the sup-interpretation of* double *is greater than* $2\times X$*, one has to find a polynomial*

weight $\omega_{\exp}$ *such that:*

$$\omega_{\exp}(X+1) \geq \theta(\texttt{double})(\omega_{\exp}(X)) \geq 2 \times \omega_{\exp}(X)$$

which is impossible.

5.2 Quasi-friendly with bounded calls criterion

The next result strengthens Theorem 5.3. Indeed it claims that even if a program is not terminating then the size of the intermediate values and, consequently, the stack frame sizes are polynomially bounded. Our goal is to control the size of the intermediate values computed during the execution of non-terminating programs. Consequently, it allows to consider programs over streams, and possible extensions to reactive programming as in [Amadio and Dal-Zilio 2004].

Definition 5.5. (Bounded recursive calls) A program $\mathbf{p}$ has *bounded recursive calls* iff it admits an additive and polynomial sup-interpretation θ and a polynomial weight ω such that for each fraternity of the shape $\mathsf{C}[\mathtt{g}_1(\overline{e_1}), \ldots, \mathtt{g}_r(\overline{e_r})]$, activated by $\mathtt{f}(p_1, \cdots, p_n)$, we have:

$$\omega_{\mathtt{f}}(\theta^*(p_1), \ldots, \theta^*(p_n)) \geq \max_{i=1..r}(\omega_{\mathtt{g}_i}(\theta^*(\overline{e_i})))$$

The condition of the quasi-friendly criterion and the condition on bounded recursive calls are independent and useful in order to control the size of the values added by recursive calls, as illustrated by the following example.

Example 10. *Consider the following non-terminating program:*

$$\begin{aligned}
&\mathtt{half}(t) = \textbf{Case } t \textbf{ of}\\
&\qquad \mathbf{S}(\mathbf{S}(x)) \to \mathbf{S}(\mathtt{half}(x))\\
&\qquad \mathbf{S}(\mathbf{0}) \to \mathbf{0}\\
&\qquad \mathbf{0} \to \mathbf{0}\\
&\mathtt{f}(x) = \textbf{Case } x \textbf{ of } x \to \mathtt{half}(\mathtt{f}(\mathtt{double}(x)))
\end{aligned}$$

where `double` *is the function of example 5. The size of the argument of* `f` *is duplicated at each recursive call. However, by taking* $\theta(\mathtt{half})(X) = X/2$, $\theta(\mathtt{double})(X) = 2 \times X$ *and* $\omega_{\mathtt{f}}(X) = X$, *we can check that the quasi-friendly criterion is satisfied, whereas the program has not bounded recursive calls, since we cannot find a polynomial weight* $\omega_{\mathtt{f}}$ *such that* $\omega_{\mathtt{f}}(X) \geq \omega_{\mathtt{f}}(2 \times X)$.

Lemma 5.6. *If a program with bounded recursive calls has a call-tree containing a branch of the shape* $\langle \mathtt{f}, v_1, \cdots, v_n\rangle \overset{*}{\rightsquigarrow} \langle \mathtt{g}, u_1, \cdots, u_m\rangle$ *with* $\mathtt{f} \approx_{Fct} \mathtt{g}$ *then:*

$$\omega_{\mathtt{f}}(\theta^*(v_1), \cdots, \theta^*(v_n)) \geq \omega_{\mathtt{g}}(\theta^*(u_1), \cdots, \theta^*(u_m))$$

Proof. We show it by induction on the number k of states in the branch:

—If $k = 1$ then $\langle \mathtt{f}, v_1, \cdots, v_n\rangle \rightsquigarrow \langle \mathtt{g}, u_1, \cdots, u_m\rangle$ and there are a definition, with a fraternity $\mathsf{C}[\mathtt{g}(e_1, \cdots, e_m)]$ activated by $\mathtt{f}(p_1, \cdots, p_n)$ such that $\mathtt{f} \approx_{Fct} \mathtt{g}$, and a substitution σ such that $p_i\sigma = v_i$ and $[\![e_j\sigma]\!] = u_j$. Combining the condition on bounded recursive calls, the monotonicity of weights and Lemma 4.9, we obtain:

$$\omega_{\mathtt{f}}(\theta^*(v_1), \cdots, \theta^*(v_n)) \geq \omega_{\mathtt{g}}(\theta^*(e_1\sigma), \cdots, \theta^*(e_m\sigma)) \geq \omega_{\mathtt{g}}(\theta^*(u_1), \cdots, \theta^*(u_m))$$

—Now suppose by induction hypothesis that if $\langle \mathtt{f}, v_1, \cdots, v_n\rangle \overset{l}{\rightsquigarrow} \langle \mathtt{h}, v'_1, \cdots, v'_r\rangle$ with $\mathtt{f} \approx_{Fct} \mathtt{h}$ and $l \leq k$, we have

$$\omega_{\mathtt{f}}(\theta^*(v_1), \cdots, \theta^*(v_n)) \geq \omega_{\mathtt{h}}(\theta^*(v'_1), \cdots, \theta^*(v'_r)) \quad (I.H.)$$

And consider the following branch of length $k+1$, with $\mathtt{g} \approx_{Fct} \mathtt{f}$:

$$\langle \mathtt{f}, v_1, \cdots, v_n\rangle \overset{k}{\rightsquigarrow} \langle \mathtt{h}, v'_1, \cdots, v'_r\rangle \rightsquigarrow \langle \mathtt{g}, u_1, \cdots, u_m\rangle$$

$$\begin{aligned} \omega_{\mathtt{f}}(\theta^*(v_1), \cdots, \theta^*(v_n)) &\geq \omega_{\mathtt{h}}(\theta^*(v'_1), \cdots, \theta^*(v'_r)) && \text{By I.H.} \\ &\geq \omega_{\mathtt{g}}(\theta^*(u_1), \cdots, \theta^*(u_m)) && \text{By I.H. again} \end{aligned}$$

□

THEOREM 5.7. *Assume that* $\boldsymbol{p}$ *is a quasi-friendly program with bounded recursive calls. For each function symbol* $\mathtt{f}$ *of* $\boldsymbol{p}$ *there is a polynomial* $R_{\mathtt{f}}$ *such that for every node* $\langle \mathtt{g}, u_1, \cdots, u_m\rangle$ *of the call-tree of root* $\langle \mathtt{f}, v_1, \cdots, v_n\rangle$,

$$\max_{i=1..m}(|u_i|) \leq R_{\mathtt{f}}(\max(|v_1|, ..., |v_n|))$$

even if $\mathtt{f}(v_1, \ldots, v_n)$ *is not terminating.*

PROOF. Given a call-tree of root $\langle \mathtt{f}, v_1, \cdots, v_n\rangle$ corresponding to a program $\mathbf{p}$. Define the level of $\mathtt{f}$ by $\mathtt{lv}(\mathtt{f}) =_{\text{def}} 0$. If $\mathtt{h} \approx_{Fct} \mathtt{g}$ then $\mathtt{lv}(\mathtt{h}) =_{\text{def}} \mathtt{lv}(\mathtt{g})$. For all $\mathtt{g}$ s.t. $\mathtt{g} >_{Fct} \mathtt{h}$, we define $\mathtt{lv}(\mathtt{h}) =_{\text{def}} \max_{\mathtt{g} >_{Fct} \mathtt{h}}(\mathtt{lv}(\mathtt{g})) + 1$. Notice that the level is fixed by the size of the program since it is bounded by the number of function symbols. We extend the notion of level to the states of the call-tree, saying that the level of a state is the level of the corresponding function symbol. The level of a call-tree is the highest level of a function occurring in the call-tree. We are going to build the required polynomial R by induction on the level of a node $\langle \mathtt{g}, u_1, \cdots, u_m\rangle$:

—If $\mathtt{lv}(\langle \mathtt{g}, u_1, \cdots, u_m\rangle) = 0$ and $\langle \mathtt{f}, v_1, \cdots, v_n\rangle \overset{*}{\rightsquigarrow} \langle \mathtt{g}, u_1, \cdots, u_m\rangle$ then $\mathtt{f} \approx_{Fct} \mathtt{g}$. Defining $R_0(X) = \omega_{\mathtt{f}}(\alpha \times X, \ldots, \alpha \times X)$, with α the constant of Lemma 4.6, we check that:

$$\begin{aligned} R_0(\max_{j=1..n}(|v_j|)) &\geq \omega_{\mathtt{f}}(\theta^*(v_1), \cdots, \theta^*(v_n)) && \text{By Lemma 4.6} \\ &\geq \omega_{\mathtt{g}}(\theta^*(u_1), \cdots, \theta^*(u_m)) && \text{By Lemma 5.6} \\ &\geq \max_{i=1..m}(\theta^*(u_i)) && \text{By Cdn 2 of Dfn 4.11} \\ &\geq \max_{i=1..m}(|u_i|) && \text{By Cdn 2 of Dfn 4.7} \end{aligned}$$

—Now, suppose that we have built a polynomial R_k at level k and take the state $\langle \mathtt{g}, u_1, \cdots, u_m\rangle$ to be of level $k+1$. If we consider the branch of the call-tree from $\langle \mathtt{f}, v_1, \cdots, v_n\rangle$ to $\langle \mathtt{g}, u_1, \cdots, u_m\rangle$, we know that there are two states $\langle \mathtt{h}, v'_1, \cdots, v'_l\rangle$ and $\langle \mathtt{h'}, u'_1, \cdots, u'_j\rangle$ of respective levels k' and $k+1$ such that $k' < k+1$ and:

$$\begin{array}{llll} \langle \mathtt{f}, v_1, \cdots, v_n\rangle \overset{*}{\rightsquigarrow} & \langle \mathtt{h}, v'_1, \cdots, v'_l\rangle \rightsquigarrow & \langle \mathtt{h'}, u'_1, \cdots, u'_j\rangle \overset{*}{\rightsquigarrow} & \langle \mathtt{g}, u_1, \cdots, u_m\rangle \\ 0 & k' & k+1 & k+1 \end{array}$$

Moreover, we know that there are a substitution σ and a definition *def* of the shape $\mathtt{h}(x_1, \cdots, x_l) = \textbf{Case } x_1, \cdots, x_l \textbf{ of } p_1, \cdots, p_l \rightarrow \mathtt{C}[\mathtt{h'}(e_1, \cdots, e_j)]$ such

that $p_i\sigma = v'_i$ and $[\![e_i\sigma]\!] = u'_i$. Since the program is quasi-friendly, we apply Theorem 5.3 over the symbols of e_i, hence we have a polynomial upper bound P_{e_i} satisfying $P_{e_i}(\max_{i=1..l}(|v'_i|)) \geq \max_{i=1..j}(|u'_i|)$. Notice that this bound remains polynomial since the number of polynomial compositions is bounded by the depth of the expressions. We define

$$Q_{def}(X) = \max_{i=1..j} P_{e_i}(X)$$
$$\text{and } S^k_{def}(X) = S(\alpha \times Q_{def}(R_k(X)))$$

with $S(X) = \omega_{\mathtt{h'}}(X, \ldots, X)$ and α the constant of Lemma 4.6. Intuitively, S^k_{def} represents a polynomial upper bound on the size of the values occurring in the states of level smaller than $k+1$ in the considered branch of the call-tree:

$$\begin{aligned}
S^k_{def}(\max_{i=1..n}|v_i|) &= S(\alpha \times Q_{def}(R_k(\max_{i=1..n}|v_i|))) \\
&\geq S(\alpha \times Q_{def}(\max_{i=1..l}|v'_i|)) && \text{By I.H.} \\
&\geq S(\max_{i=1..j}(\alpha \times |u'_i|)) && \text{By definition of } Q_{def} \\
&\geq S(\max_{i=1..j}(\theta^*(u'_i))) && \text{By Lemma 4.6} \\
&\geq \omega_{\mathtt{h'}}(\theta^*(u'_1), \cdots, \theta^*(u'_j)) && \text{By monotonicity of weight} \\
&\geq \omega_{\mathtt{g}}(\theta^*(u_1), \cdots, \theta^*(u_m)) && \text{By Lemma 5.6} \\
&\geq \max_{i=1..m}(|u_i|) && \text{By Dfn 4.11}
\end{aligned}$$

Now we have to build a polynomial which bounds the values of level smaller than $k+1$ in each branch of the call-tree. Let E_k be the set of definitions of the shape $\mathtt{h}(x_1, \cdots, x_l) = \textbf{Case } x_1, \cdots, x_l \textbf{ of } p_1, \cdots, p_l \to \mathtt{C[h'}(e_1, \cdots, e_j)]$ with levels of $\mathtt{h}$ and $\mathtt{h'}$ being respectively k' and $k+1$ with $k' < k+1$. As in the previous case, we define the polynomials Q_{def} and S^k_{def} for every definition $def \in E_k$. Finally, just define $R_{k+1}(X) = \max_{def \in E_k}(S^k_{def}(X))$

By construction, it defines a polynomial bound on the size of the values occurring in the states of level smaller than $k+1$. Now define $R_{\mathtt{f}}$ to be a polynomial such that $R_{\mathtt{f}}(X) \geq R_\gamma(X)$. γ being the highest level, R_γ corresponds to a bound on the values occurring in each state of the call-tree. Notice that the polynomial $R_{\mathtt{f}}$ exists since R_γ remains in $\textbf{Max-poly}\{\mathbb{R}^+\}$. Indeed the number of compositions at each step remains bounded by γ, which is bounded by the size of the program. □

Example 11 (Streams). *As mentioned above, Theorem 5.7 also holds for non-terminating programs. Thus it particularly holds for a class of programs including streams. For that purpose we introduce streams in the programming language using a binary constructor* :: . *In a stream* $h :: t$, h *is called the head of the stream and* t *is the tail of the stream. We suppose that we have already defined a semantics over*

streams in a classical way, i.e. left-to-right and call-by-value semantics.

$$\begin{aligned}
&\mathtt{add}(x,y) = \mathbf{Case}\ x,y\ \mathbf{of}\\
&\qquad \mathbf{S}(u), v \to \mathbf{S}(\mathtt{add}(u,v))\\
&\qquad \mathbf{0}, v \to v\\
&\mathtt{addstream}(x,y) = \mathbf{Case}\ x,y\ \mathbf{of}\ u :: l, v :: l' \to \mathtt{add}(u,v) :: \mathtt{addstream}(l,l')
\end{aligned}$$

This (merging) program is quasi-friendly with bounded recursive calls by taking $\theta^*(l) = L$, $\theta(\mathtt{add})(X,Y) = X+Y$, $\theta^*(x :: l) = X+L+1$, $\omega_{\mathtt{add}}(X,Y) = X+Y$ *and* $\omega_{\mathtt{addstream}}(X,Y) = X+Y$*:*

—Condition of the quasi-friendly criterion:

$$\begin{aligned}
\omega_{\mathtt{add}}(\theta^*(\mathbf{S}(u)), \theta^*(v)) &= U+V+1\\
&\geq U+V+1 = \theta(\mathbf{S})(\omega_{\mathtt{add}}(\theta^*(u), \theta^*(v)))\\
\omega_{\mathtt{addstream}}(\theta^*(u :: l), \theta^*(v :: l')) &= U+V+L+L'+2\\
&\geq L+L'+U+V+1\\
&= \theta^*(\mathtt{add}(u,v) :: \omega_{\mathtt{addstream}}(\theta^*(l), \theta^*(l')))
\end{aligned}$$

—Condition on the bounded recursive calls:

$$\begin{aligned}
\omega_{\mathtt{add}}(\theta^*(\mathbf{S}(u)), \theta^*(v)) &= U+V+1\\
&\geq U+V = \omega_{\mathtt{add}}(\theta^*(u), \theta^*(v))\\
\omega_{\mathtt{addstream}}(\theta^*(u :: l), \theta^*(v :: l')) &= U+V+L+L'+2\\
&\geq L+L' = \omega_{\mathtt{addstream}}(\theta^*(l), \theta^*(l'))
\end{aligned}$$

Thus Theorem 5.7 holds. It would be non-sense to consider streams as inputs, since the size of a stream is unbounded. Consequently, the inputs in the application of this Theorem are chosen to be a restricted number of stream heads. In the same way, every mapping program over streams of the shape:

$$\mathtt{f}(x) = \mathbf{Case}\ x\ \mathbf{of}\ z :: l \to \mathtt{g}(z) :: \mathtt{f}(l)$$

is quasi-friendly with bounded recursive calls in so far as g *represents a quasi-friendly program. Thus Theorem 5.7 also applies. Moreover, for all these programs we know that the values computed in the output streams (i.e. in the heads of right-hand side definition) are polynomially bounded in the size of some of the inputs (heads) since the computations involve only quasi-friendly function symbols over non-stream data (otherwise some parts of the program would never be evaluated). Finally, an example of non-quasi-friendly with bounded recursive calls program is:*

$$\mathtt{f}(x) = \mathbf{Case}\ x\ \mathbf{of}\ z :: l \to \mathtt{f}(z :: z :: l)$$

In fact, this program does not fit our requirements since it adds infinitely the head of the stream to its argument, computing thus an unbounded value.

5.3 Quasi-friendly modulo projection criterion

In the case of a particular destructive operation over a recursive argument, one has to know a precise upper bound on the size of the recurrence arguments in order

to control the recursion. However such a task is very tricky when we consider destructors. Consider the following example:

Example 12 (Identity).

$$\begin{aligned}\mathtt{f}(x) = \mathbf{Case}\ x\ \mathbf{of}\\ l \to \mathbf{c}(\mathbf{hd}(l), \mathtt{f}(\mathbf{tl}(l)))\\ \mathbf{nil} \to \mathbf{nil}\end{aligned}$$

This program computes the identity of a list l using the destructors **tl** *and* **hd** *which compute respectively the tail and the head of a list l. $\theta^*(\mathbf{tl}(l))$ and $\theta^*(\mathbf{hd}(l))$ are at least taken to be equal to $\theta(l) = L$ in order to bound the size of the value they compute. If we want to satisfy the quasi-friendly criterion, taking $\theta(\mathbf{c})(X, Y) = X + Y + k$, for some $k \geq 1$, we obtain:*

$$\omega_{\mathtt{f}}(L) \geq L + k + \omega_{\mathtt{f}}(L)$$

Consequently, this program is not quasi-friendly.

The problem comes directly from the fact that $\theta^*(\mathbf{hd}(l))$ and $\theta^*(\mathbf{tl}(l))$ are considered as functions of $\theta(l)$, thus generating a too large upper bound. In what follows, we try to overcome this problem by replacing the sup-interpretation of destructor symbols by new variables satisfying a system of inequalities.

Definition 5.8. (Projector) A function symbol $\mathtt{d}_i^{\mathbf{c}}$ is called the i-th projector relative to the constructor **c** if it is defined by:

$$\mathtt{d}_i^{\mathbf{c}}(x) = \mathbf{Case}\ x\ \mathbf{of}\ \mathbf{c}(e_1, \cdots, e_n) \to e_i$$

The sup-interpretation of a projection $\mathtt{d}_j^{\mathbf{c}}(e)$ is not a function and is considered as a new variable.

Definition 5.9. (Projection Sup-interpretation) Given a projector $\mathtt{d}_j^{\mathbf{c}}$, an expression e and a sup-interpretation θ, the canonical extension θ^* of θ over the projection $\mathtt{d}_j^{\mathbf{c}}(e)$ is modified by the following definition:

$$\theta^*(\mathtt{d}_j^{\mathbf{c}}(e)) =_{\text{def}} X^e_{\mathtt{d}_j^{\mathbf{c}}}$$

with $X^e_{\mathtt{d}_j^{\mathbf{c}}}$ a fresh variable.

For every program, in presence of projections, we generate a set of constraints where the sup-interpretations of projections are taken to be new variables:

Definition 5.10. (Projector Constraints) Given a program **p**, let $\mathtt{Expression}(\mathbf{p})$ be the set of expressions $e \in \mathtt{Expression}$ which occur in the definitions of **p**. We define the set of projector constraints S by:

$$S = \bigcup_{\mathtt{d}_j^{\mathbf{c}}(e) \in \mathtt{Expression}(\mathbf{p})} \left\{ \sum_{j=1}^{n} X^e_{\mathtt{d}_j^{\mathbf{c}}} + 1 \leq \theta^*(e) \right\}$$

These inequalities correspond to constraints on the sup-interpretations of projections. In practice, they are always satisfied if the sup-interpretation is additive.

Indeed, suppose that $e = \mathbf{c}(e_1, \cdots, e_n)$, taking $X^e_{\mathtt{d}^{\mathbf{c}}_i} = \theta^*(e_i)$, we have:

$$\begin{aligned}\theta^*(e) &= \theta^*(\mathbf{c})(\theta^*(e_1), \cdots, \theta^*(e_n)) \\ &= \theta^*(\mathbf{c})(X^e_{\mathtt{d}^{\mathbf{c}}_1}, \ldots, X^e_{\mathtt{d}^{\mathbf{c}}_n}) \\ &\geq \sum_{j=1}^{n} X^e_{\mathtt{d}^{\mathbf{c}}_j} + 1\end{aligned}$$

Definition 5.11. (Quasi-friendly modulo projection) Given a program $\mathbf{p}$ and the corresponding set of projector constraints S, $\mathbf{p}$ is quasi-friendly modulo projection if there are a polynomial and additive sup-interpretation θ and a polynomial weight ω such that S implies that $\mathbf{p}$ is quasi-friendly.

EXAMPLE 13. *Consider the following program which reverses a list given as input and can be found in [Lee et al. 2001]:*

$$\begin{aligned}\mathtt{reverse}(l) &= \mathbf{Case}\ l\ \mathbf{of}\ l \to \mathtt{rev}(l, \mathbf{nil}) \\ \mathtt{rev}(l, a) &= \mathbf{Case}\ l, a\ \mathbf{of}\ l, a \to \mathtt{if}(l{=}\mathbf{nil}, a, \mathtt{rev}(\mathbf{tl}(l), \mathbf{c}(\mathbf{hd}(l), a)))\end{aligned}$$

The generated system of projector constraints is defined by:

$$S = \left\{ X^{\mathbf{tl}(l)} + X^{\mathbf{hd}(l)} + 1 \leq L) \right\}$$

This program has only one fraternity $\mathtt{if}(l{=}\mathbf{nil}, a, \mathtt{rev}(\mathbf{tl}(l), \mathbf{c}(\mathbf{hd}(l), a)))$. *Hence the quasi-friendly criterion corresponds to:*

$$\omega_{\mathtt{rev}}(L, A) \geq \max(A, \omega_{\mathtt{rev}}(X^{\mathbf{tl}(l)}, A + X^{\mathbf{hd}(l)} + k))$$

with $\theta(\mathbf{c})(X, Y) = X + Y + k$ *and* $\theta(\mathtt{if})(X, Y, Z) = \max(Y, Z)$. *Taking* $k = 1$ *and* $\omega_{\mathtt{rev}}(X, Y) = X + Y$, *we obtain:*

$$L + A \geq X^{\mathbf{tl}(l)} + X^{\mathbf{hd}(l)} + A + 1$$

Consequently, S implies that $\boldsymbol{p}$ *is quasi-friendly and the program is quasi-friendly modulo projection.*

EXAMPLE 14. *The program of example 12 is quasi-friendly modulo projection by taking* $\omega_{\mathtt{f}}(X) = X$ *and* $\theta(\mathbf{c})(X, Y) = X + Y + 1$.

THEOREM 5.12. *Assume that a program* $\boldsymbol{p}$ *is quasi-friendly modulo projection, then for each function symbol* $\mathtt{f}$ *of* $\boldsymbol{p}$ *there is a polynomial* $P_{\mathtt{f}}$ *such that for every values* $v_1, \ldots, v_n$,

$$\|\mathtt{f}(v_1, \ldots, v_n)\| \leq P_{\mathtt{f}}(\max(|v_1|, ..., |v_n|))$$

PROOF. Just notice that the system S is always satisfied, so the satisfaction of the sentence "S implies that $\mathbf{p}$ is quasi-friendly" is equivalent to "$\mathbf{p}$ is quasi-friendly". □

EXAMPLE 15 (QUICKSORT). *The following program computes the quicksort algorithm using the function* `order` *which, given a unary number* n *and a list* l *as inputs, returns a pair* $\mathbf{pair}(u, v)$ *of two lists* u *and* v *which represent the elements*

of the input list l smaller than n and, respectively, strictly greater than n.

$$\begin{aligned}
&\mathtt{append}(x,y) = \mathbf{Case}\ x,y\ \mathbf{of}\\
&\qquad \mathbf{nil}, u \to u\\
&\qquad \mathbf{c}(n,v), u \to \mathbf{c}(n, \mathtt{append}(v,u))\\
&\mathtt{p}_1(x) = \mathbf{Case}\ x\ \mathbf{of}\ \mathbf{pair}(p_1,p_2) \to p_1\\
&\mathtt{p}_2(x) = \mathbf{Case}\ x\ \mathbf{of}\ \mathbf{pair}(p_1,p_2) \to p_2\\
&\mathtt{order}(w,x,y,z) = \mathbf{Case}\ w,x,y,z\ \mathbf{of}\\
&\qquad n, \mathbf{c}(m,l), u, v \to \mathtt{if}(\mathbf{le}(m,n), \mathtt{order}(n,l,\mathbf{c}(m,u),v), \mathtt{order}(n,l,u,\mathbf{c}(m,v))\\
&\qquad n, \mathbf{nil}, u, v \to \mathbf{pair}(u,v)\\
&\mathtt{qs}(x) = \mathbf{Case}\ x\ \mathbf{of}\\
&\mathbf{nil} \to \mathbf{nil}\\
&\mathbf{c}(n,u) \to \mathtt{append}(\mathtt{qs}(\mathtt{p}_1(\mathtt{order}(n,u,\mathbf{nil},\mathbf{nil}))), \mathbf{c}(n, \mathtt{qs}(\mathtt{p}_2(\mathtt{order}(n,u,\mathbf{nil},\mathbf{nil})))))
\end{aligned}$$

append *is a quasi-friendly program. We can show it by taking* $\theta(\mathbf{c})(X,Y) = X+Y+1$, $\theta(\mathbf{S})(X) = X+1$ *and* $\omega_{\mathtt{append}}(X,Y) = X+Y$. *Since* $\mathtt{p}_1$ *and* $\mathtt{p}_2$ *are projectors, the set S of projector constraints corresponding to this system of inequalities is equal to*

$$\left\{ X_{\mathtt{p}_1}^{\mathtt{order}(n,u,\mathbf{nil},\mathbf{nil})} + X_{\mathtt{p}_2}^{\mathtt{order}(n,u,\mathbf{nil},\mathbf{nil})} + 1 \leq \theta^*(\mathtt{order}(n,u,\mathbf{nil},\mathbf{nil})) \right\}$$

Moreover, taking $\theta(\mathtt{order})(W,X,Y,Z) = X+Y+Z+1$, *we obtain:*

$$S = \left\{ X_{\mathtt{p}_1}^{\mathtt{order}(n,u,\mathbf{nil},\mathbf{nil})} + X_{\mathtt{p}_2}^{\mathtt{order}(n,u,\mathbf{nil},\mathbf{nil})} \leq U \right\}$$

Taking $\theta(\mathtt{if})(X,Y,Z) = \max(Y,Z)$ *and* $\theta(\mathtt{append})(X,Y) = X+Y$, *we have to check that the following inequalities hold in order to show that* $\boldsymbol{p}$ *is quasi-friendly:*

$$\begin{aligned}
\omega_{\mathtt{order}}(N, M+L+1, U, V) &\geq \omega_{\mathtt{order}}(N, L, M+U+1, V)\\
&\geq \omega_{\mathtt{order}}(N, L, U, V+M+1)\\
\omega_{\mathtt{qs}}(N+U+1) &\geq \sum_{i=1}^{2} \omega_{\mathtt{qs}}(X_{\mathtt{p}_i}^{\mathtt{order}(n,u,\mathbf{nil},\mathbf{nil})}) + N + 1
\end{aligned}$$

Finally, taking $\omega_{\mathtt{qs}}(X) = X$ *and* $\omega_{\mathtt{order}}(W,X,Y,Z) = W+X+Y+Z$, *these inequalities are transformed into the following system:*

$$\begin{aligned}
&\{ N+M+L+U+V+1 \geq N+M+L+U+V+1,\\
&N+U+1 \geq \max(X_{\mathtt{p}_1}^{\mathtt{order}(n,u,\mathbf{nil},\mathbf{nil})}, X_{\mathtt{p}_2}^{\mathtt{order}(n,u,\mathbf{nil},\mathbf{nil})}),\\
&N+U+1 \geq X_{\mathtt{p}_1}^{\mathtt{order}(n,u,\mathbf{nil},\mathbf{nil})} + X_{\mathtt{p}_2}^{\mathtt{order}(n,u,\mathbf{nil},\mathbf{nil})} + N + 1\ \}
\end{aligned}$$

which is implied by S. Consequently, we conclude that the program is quasi-friendly modulo projection.

6. COMPARISON WITH QUASI-INTERPRETATIONS

The quasi-interpretations were introduced by Bonfante, Marion and Moyen in [Marion and Moyen 2000; Bonfante et al. 2001; 2007]. Like a sup-interpretation, a quasi-interpretation is an assignment which provides an upper bound on function outputs

by static analysis of first order functional programs. However it differs for two main reasons. The first one is that a quasi-interpretation is defined for each symbol of a program. The second one is that the the quasi-interpretation of each symbol has the subterm property. Combined with recursive path orderings, quasi-interpretations allow to characterize complexity classes such as the set of polynomial time functions as well as the set of polynomial space functions.

Definition 6.1. A quasi-interpretation is a total (i.e. defined for every symbol of the program) additive assignment $(\!|-|\!)$ which is monotonic and has the subterm property (i.e. For every symbol b of arity $n, \forall i \in \{1,n\}, (\!|b|\!)(\ldots,X_i,\ldots) \geq X_i$) such that for every maximal expression e activated by $\mathtt{f}(p_1,\cdots,p_n)$ we have:

$$(\!|\mathtt{f}(p_1,\cdots,p_n)|\!)^* \geq (\!|e|\!)^*$$

As demonstrated in [Bonfante et al. 2001; 2007; Marion and Moyen 2000], quasi-interpretations have the following property:

PROPOSITION 6.2. *Given a program $\boldsymbol{p}$ which admits an additive quasi-interpretation $(\!|-|\!)$, for each function symbol* $\mathtt{f}$ *of $\boldsymbol{p}$ and any $v, v_1, \cdots, v_n \in \mathcal{V}$,*

$$(\!|\mathtt{f}|\!)((\!|v_1|\!)^*,\ldots,(\!|v_n|\!)^*) \geq (\!|[\![\mathtt{f}]\!](v_1,\cdots,v_n)|\!)^*$$
$$(\!|v|\!)^* \geq |v|$$

THEOREM 6.3. *Every additive quasi-interpretation is a sup-interpretation.*

PROOF. By Proposition 6.2, conditions 2 and 3 of Definition 4.7 hold. By Definition 6.1, a quasi-interpretation is monotonic, so condition 1 of Definition 4.7 holds. □

A very interesting consequence of this Theorem concerns the sup-interpretation synthesis problem. The synthesis problem consists in finding a sup-interpretation for a given program. It was introduced by Amadio in [Amadio 2003] for quasi-interpretations. This problem is relevant in a perspective of automating the complexity analysis of programs. Amadio showed [Amadio 2003] that some rich classes of quasi-interpretations are in NP and in [Bonfante et al. 2005], it was demonstrated that the quasi-interpretation synthesis with bounded polynomials over reals is decidable. Consequently, we get some heuristics for the synthesis of sup-interpretations in **Max-Poly** $\{\mathbb{R}^+\}$.

THEOREM 6.4. *Every program that admits a polynomial and additive quasi-interpretation is quasi-friendly.*

PROOF. By Theorem 6.3, every quasi-interpretation defines a sup-interpretation. Moreover, every quasi-interpretation is a weight. □

PROPOSITION 6.5. *There exist quasi-friendly programs that do not have any polynomial quasi-interpretation.*

PROOF. Program of example 1 is quasi-friendly but does not admit any quasi-interpretation. In fact, suppose that it admits an additive quasi-interpretation $(\!|-|\!)$

satisfying $(\!|\mathtt{S}|\!)(X) = X + k$, for some constant k. For the last definition, we have:

$$\begin{aligned}
(\!|\mathtt{q}(\mathbf{S}(v), \mathbf{S}(u))|\!)^* &= (\!|\mathtt{q}|\!)(V + k, U + k) && \text{By Dfn of assignments}\\
&\geq (\!|\mathbf{S}(\mathtt{q}(\mathtt{minus}(v, u), \mathbf{S}(u)))|\!)^* && \text{By Dfn of quasi-interpretations}\\
&\geq k + (\!|\mathtt{q}|\!)(\max(U, V), U + k) && \text{By subterm property}\\
&> (\!|\mathtt{q}|\!)(V + k, U + k) && \text{for } U \geq V + k
\end{aligned}$$

Consequently, we obtain a contradiction and $\mathtt{q}$ does not admit any quasi-interpretation. □

In [Bonfante et al. 2001; 2007; Marion and Moyen 2000], some characterizations of the functions computable in polynomial time and polynomial space were given. Theorems 5.3 and 6.3 allow to adapt these results to the sup-interpretations.

Given a precedence (quasi-order) $\geq'_{Fct}$ on Fct. Define the equivalence relation $\approx'_{Fct}$ as $\mathtt{f} \approx'_{Fct} \mathtt{g}$ iff $\mathtt{f} \geq'_{Fct} \mathtt{g}$ and $\mathtt{g} \geq'_{Fct} \mathtt{f}$. We associate to each function symbol $\mathtt{f}$ a status $st(\mathtt{f})$ in $\{p, l\}$, satisfying if $\mathtt{f} \approx'_{Fct} \mathtt{g}$ then $st(\mathtt{f}) = st(\mathtt{g})$. The status indicates how to compare the arguments of recursive calls.

Definition 6.6. The product extension $\prec^p$ and the lexicographic extension $\prec^l$ of $\prec$ over sequences are defined by:

—$(m_1, \cdots, m_k) \prec^p (n_1, \cdots, n_k)$ if and only if (i) $\forall i \leq k, m_i \preceq n_i$ and (ii) $\exists j \leq k$ such that $m_j \prec n_j$.

—$(m_1, \cdots, m_k) \prec^l (n_1, \cdots, n_l)$ if and only if $\exists j$ such that $\forall i < j,\ m_i \preceq n_i$ and $m_j \prec n_j$

Definition 6.7. Given a precedence $\geq'_{Fct}$ and a status st, we define the recursive path ordering $\prec_{rpo}$ as follows:

$$\frac{u \preceq_{rpo} t_i}{u \prec_{rpo} \mathtt{f}(\ldots, t_i, \ldots)} \qquad \frac{\forall i\ u_i \prec_{rpo} \mathtt{f}(t_1, \cdots, t_n) \qquad \mathtt{g} \geq'_{Fct} \mathtt{f}}{\mathtt{g}(u_1, \cdots, u_m) \prec_{rpo} \mathtt{f}(t_1, \cdots, t_n)}$$

$$\frac{(u_1, \cdots, u_n) \prec_{rpo}^{st(\mathtt{f})} (t_1, \cdots, t_n) \qquad \mathtt{f} \approx'_{Fct} \mathtt{g} \qquad \forall i\ u_i \prec_{rpo} \mathtt{f}(t_1, \cdots, t_n)}{\mathtt{g}(u_1, \cdots, u_n) \prec_{rpo} \mathtt{f}(t_1, \cdots, t_n)}$$

A program is ordered by $\prec_{rpo}$ if there are a precedence $\geq'_{Fct}$ and a status st such that for each maximal expression r activated by l, the inequality $r \prec_{rpo} l$ holds.

THEOREM 6.8.

—*The set of functions computed by quasi-friendly programs admitting an additive sup-interpretation and ordered by $\prec_{rpo}$ where each function symbol has a product status is exactly the set of functions computable in polynomial time.*

—*The set of functions computed by quasi-friendly programs admitting an additive sup-interpretation and ordered by $\prec_{rpo}$ is exactly the set of functions computable in polynomial space.*

PROOF. We give here the main ingredients of the proof which can be found in [Bonfante et al. 2007] for quasi-interpretations.

$$\frac{x\sigma = w}{\mathcal{R}, \sigma \vdash \langle C, x\rangle \to \langle C, w\rangle}\,(Variable) \qquad \frac{\mathbf{c} \in Cns \quad \mathcal{R}, \sigma \vdash \langle C_{i-1}, t_i\rangle \to \langle C_i, w_i\rangle}{\mathcal{R}, \sigma \vdash \langle C_0, \mathbf{c}(t_1, \cdots, t_n)\rangle \to \langle C_n, \mathbf{c}(w_1, \cdots, w_n)\rangle}\,(Cons)$$

$$\frac{\mathtt{f} \in Fct \quad \mathcal{R}, \sigma \vdash \langle C_{i-1}, t_i\rangle \to \langle C_i, w_i\rangle \quad (\mathtt{f}(w_1, \cdots, w_n), w) \in C_n}{\mathcal{R}, \sigma \vdash \langle C_0, \mathtt{f}(t_1, \cdots, t_n)\rangle \to \langle C_n, w\rangle}\,(Cache\ reading)$$

$$\frac{\mathcal{R}, \sigma \vdash \langle C_{i-1}, t_i\rangle \to \langle C_i, w_i\rangle \quad \mathtt{f}(\overline{x}) = \mathbf{Case}\ \overline{x}\ \mathbf{of}\ \overline{p} \to e \quad p_i\sigma' = w_i \quad \mathcal{R}, \sigma' \vdash \langle C_n, e\rangle \to \langle C, w\rangle}{\mathcal{R}, \sigma \vdash \langle C_0, \mathtt{f}(t_1, \cdots, t_n)\rangle \to \langle C \cup (\mathtt{f}(w_1, \cdots, w_n), w), w\rangle}\,(Push)$$

Fig. 2. Evaluation of a program with memoization of intermediate evaluations

—Due to the $\prec_{rpo}$ ordering with product status, any recursive sub-call of some $\mathtt{f}(v_1, \cdots, v_n)$, with $\mathtt{f}$ function symbol and v_i constructor terms, will be done on subterms of the v_i. A consequence of Theorem 5.3 is that any other subcalls will be done on arguments of polynomial size. So one may use a memoization technique à la Jones [Jones 1997] which leads us to define a call-by-value interpreter with cache displayed in Figure 2. The completeness is obtained combining the proof of [Bonfante et al. 2007] and Theorem 6.4 and we obtain the set of functions computable in polynomial time.

—Theorem 5.3 and the $\prec_{rpo}$ ordering imply that both the size of a state and the length of a branch in the call-tree are polynomially bounded by the size of the inputs. The completeness is obtained combining the proof of [Bonfante et al. 2007] and Theorem 6.4 and we obtain the set of functions computable in polynomial space.

□

7. APPLICATION TO DEPENDENCY PAIRS

Definition 7.1. Assume that $\mathbf{p}$ is a program. A dependency pair

$$\langle \mathtt{f}(p_1, \cdots, p_n), \mathtt{g}(e_1, \cdots, e_m)\rangle$$

is a couple such that $\mathtt{g}(e_1, \cdots, e_m)$ is activated by $\mathtt{f}(p_1, \cdots, p_n)$ and $\mathtt{g} \in Fct$. We define the dependency pair graph by:

—The nodes are the dependency pairs

—Given $u = \langle \mathtt{f}_1(p_1, \cdots, p_n), \mathtt{f}_2(e_1, \cdots, e_m)\rangle$, $v = \langle \mathtt{f}_3(q_1, \cdots, q_k), \mathtt{f}_4(d_1, \cdots, d_l)\rangle$, two dependency pairs, there is an edge from u to v if there is a substitution σ such that $\mathtt{f}_2(e_1, \cdots, e_m)\sigma \xrightarrow{*} \mathtt{f}_3(q_1, \cdots, q_k)\sigma$, where $\xrightarrow{*}$ is the rewrite relation induced by the definitions of the program.

A cycle of dependency pairs is defined to be a cycle in the dependency pair graph. We say that the dependency pair u is involved in a cycle if u belongs to a cycle in the dependency graph.

Remark 7.2. A fraternity $\mathsf{C}[\mathtt{f}_1(\overline{e_1}), \ldots, \mathtt{f}_n(\overline{e_n})]$ activated by $\mathtt{f}(p_1, \cdots, p_n)$ corresponds to n dependency pairs $\langle \mathtt{f}(p_1, \cdots, p_n), \mathtt{f}_i(\overline{e_i})\rangle$ involved in some cycles of the

dependency pair graph.

The following Theorem is due to Arts and Giesl [Arts and Giesl 2000]:

THEOREM 7.3. *A program* $\boldsymbol{p}$ *is terminating if there is a well-founded weakly monotonic quasi-ordering* $\geq_{q.o.}$*, closed under substitution, such that:*

(1) *For each definition* $\mathtt{f}(\overline{x}) = \textbf{Case } \overline{x} \textbf{ of } \overline{p_1} \to e_1 \dots \overline{p_m} \to e_m$*, we have:*

$$\forall i \in \{1, m\}\,,\ \mathtt{f}(\overline{p_i}) \geq_{q.o.} e_i$$

(2) *For each dependency pair* $\langle s, t\rangle$*,* $s \geq_{q.o.} t$

(3) *For each cycle in the dependency pair graph, there is a dependency pair* $\langle s, t\rangle$ *such that* $s >_{q.o.} t$

Now we derive a termination criterion which is an application of the quasi-friendly criterion to the dependency pairs method.

Definition 7.4. (Strictly bounded recursive calls) A program $\mathbf{p}$ has strictly bounded recursive calls if it admits an additive sup-interpretation θ and a weight ω, both in $\textbf{Max-Poly}\,\{\mathbb{N}\}$, such that:

—$\mathbf{p}$ has bounded recursive calls.

—For each cycle in the dependency pair graph, there is a dependency pair of the shape $\langle \mathtt{f}(p_1, \cdots, p_n), \mathtt{g}(e_1, \cdots, e_m)\rangle$ such that

$$\omega_{\mathtt{f}}(\theta^*(p_1), \cdots, \theta^*(p_n)) > \omega_{\mathtt{g}}(\theta^*(e_1), \cdots, \theta^*(e_m))$$

THEOREM 7.5. *A program which has strictly bounded recursive calls is terminating.*

PROOF. Define the quasi-ordering $\geq_{q.o.}$ by $s \geq_{q.o.} t$ if $s = \mathtt{f}(\overline{e})$ and $t = \mathtt{g}(\overline{d})$ and either $\mathtt{f} >_{Fct} \mathtt{g}$ or $\mathtt{f} \approx_{Fct} \mathtt{g}$ and $\omega_{\mathtt{f}}(\theta^*(\overline{e})) > \omega_{\mathtt{g}}(\theta^*(\overline{d}))$ (Notice that $\geq_{Fct}$ is extended to constructor symbols by $\forall \mathtt{f} \in Fct,\ \forall \mathbf{c} \in Cns,\ \mathtt{f} >_{Fct} \mathbf{c}$). Applying Lemma 5.6 (just notice that this Lemma still holds for programs with strictly bounded recursive calls, the only distinction is in the strict inequality), for two successive states of the call-tree $\langle \mathtt{f}, u_1, \cdots, u_n\rangle$ and $\langle \mathtt{f}, v_1, \cdots, v_n\rangle$ involving the same function symbol, we obtain:

$$\omega_{\mathtt{f}}(\theta^*(u_1), \cdots, \theta^*(u_n)) > \omega_{\mathtt{f}}(\theta^*(v_1), \cdots, \theta^*(v_n))$$

Since the considered assignments are in $\textbf{Max-poly}\,\{\mathbb{N}\}$, the condition on strictly bounded recursive calls implies that every cycle of dependency pairs decreases the weight by at least 1. The above remark combined with the fact that the number of function symbols is bounded by the size of the program implies that the quasi-ordering is well-founded. Moreover, this quasi-ordering is weakly monotonic and closed by substitution. Consequently, we can apply Theorem 7.3 and the program terminates. □

Remark 7.6. Notice that Theorem 7.5 can also be applied to non-polynomial sup-interpretations, the only requirement is to consider functions over natural numbers for preserving the well-foundedness properties.

LEMMA 7.7. *Suppose that a program is quasi-friendly with strictly bounded recursive calls, then the size of each branch of the call-tree is polynomially bounded by the input size, where the size of a branch is taken to be the sum of the size of all its states.*

PROOF. By Theorem 5.7, we know that every value of a state has a size polynomially bounded by the input size. That is, there is a polynomial R such that for every state $\langle \mathtt{g}, v_1, \cdots, v_k \rangle$ of a call-tree of root $\langle \mathtt{f}, u_1, \cdots, u_n \rangle$, we have:

$$\forall i \in \{1, k\}\,,\ |v_i| \leq R(\max_{j=1..n}(|u_j|))$$

So the size of each state is bounded by $Q(\max_{j=1..n} |u_j|)$ with $Q(X) = m \times R(X)$ and m the maximal arity of the program. In the proof of Theorem 7.5, we have shown that each cycle starting from $\langle \mathtt{g}, v_1, \cdots, v_k \rangle$ has a number of occurrences bounded by $\omega_{\mathtt{g}}(\theta^*(v_1), \cdots, \theta^*(v_k))$ (which is bounded by $\omega_{\mathtt{g}}(\alpha \times |v_1|, \ldots, \alpha \times |v_k|)$ by Lemma 4.6). Consequently, each cycle starting from $\langle \mathtt{g}, v_1, \cdots, v_k \rangle$ has at most $\omega_{\mathtt{g}}(\alpha \times Q(\max_{j=1..n} |u_j|), \ldots, \alpha \times Q(\max_{j=1..n} |u_j|))$ occurrences. Now define $\omega(X) = \max_{\mathtt{g} \in Fct}(\omega_{\mathtt{g}}(\alpha \times Q(X), \ldots, \alpha \times Q(X)))$ whenever $\omega_{\mathtt{g}}$ is defined. Let A be the maximal number of cycles in the program (Notice that A is considered as a constant since it only depends on the size of the program). We know that the depth of each branch starting from $\langle \mathtt{f}, u_1, \cdots, u_n \rangle$ is bounded by $A \times \omega(\max_{j=1..n}(|u_j|))$. Finally, $A \times \omega(\max_{j=1..n}(|u_j|)) \times Q(\max_{j=1..n}(|u_j|))$ is the required polynomial bound on the size of each branch. □

THEOREM 7.8. *The set of functions computed by quasi-friendly programs with strictly bounded recursive calls is exactly the set of functions computable in polynomial space.*

PROOF. By Lemma 7.7, we know that the size of each branch and each state of the call-tree is polynomially bounded by the size of the inputs. Evaluating the program in the depth of the call-tree, we obtain that the set of functions computed by quasi-friendly programs which have strictly bounded recursive calls is included in FPSPACE. The proof of completeness is inspired by a characterization of [Bonfante et al. 2007] using Parallel Register Machines (PRM). Savitch [Savitch 1970] and Chandra, Kozen and Stockmeyer [Chandra et al. 1981] have shown that the set of functions computed by PRM in polynomial time is exactly the set of functions computable in polynomial space. We let the reader check that the program given in [Bonfante et al. 2007] which simulates PRM by a TRS is clearly quasi-friendly with strictly bounded recursive calls. □

8. APPLICATION TO SIZE-CHANGE PRINCIPLE

Since the condition on strictly bounded recursive calls tries to control the arguments of a recursive call together, it is closer from the dependency pairs method than from the size-change principle method of Jones et al. [Lee et al. 2001] which considers the arguments of a recursive call separately (See more recently [Anderson and Khoo 2003; Avery 2006]). For a more detailed comparison between both termination criteria, see [Thiemann and Giesl 2005]. Consequently, an interesting application of sup-interpretations would consist in an adaptation to the size-change principle method in order to prove the termination of more algorithms.

Definition 8.1. (Size-change graphs and multipaths) Given a well-founded ordering $>_{w.f.o.}$ on $\mathcal{V}$, a program $\mathbf{p}$ and two function symbols $\mathtt{f}$ and $\mathtt{g}$ of $\mathbf{p}$, of respective arity n and m, such that the expression $\mathtt{g}(d_1, \cdots, d_m)$ is activated by $\mathtt{f}(p_1, \cdots, p_n)$ for some expressions $d_1, \cdots, d_m$ and some patterns $p_1, \cdots, p_n$, a size-change graph from $\mathtt{f}$ to $\mathtt{g}$ is a bipartite graph noted $G : \mathtt{f} \to \mathtt{g}$ from the arguments $x_1, \cdots, x_n$ of $\mathtt{f}$ to the arguments $y_1, \cdots, y_m$ of $\mathtt{g}$ where:

—The nodes are the arguments $x_1, \cdots, x_n$,$y_1, \cdots, y_m$.
—There is an arc from x_i to y_j if and only if, for each substitution σ, $p_i\sigma \geq_{w.f.o.} d_j\sigma$[1].
—Moreover, if, for each substitution σ, $p_i\sigma >_{w.f.o.} d_j\sigma$, then the arc is labeled by $\downarrow$.

A size-change multipath is a possibly infinite sequence $G_1, G_2, \ldots$ of size-change graphs such that G_i is from $\mathtt{f}_i$ to $\mathtt{f}_{i+1}$ and G_{i+1} is from $\mathtt{f}_{i+1}$ to $\mathtt{f}_{i+2}$.
A thread of a mutipath is defined to be a connected path of arcs.

Notice that they are only finitely many size-change graphs for a given program.

Example 16. *If $>_{w.f.o.}$ is taken to be a well-founded order on the size of the values (i.e. $u >_{w.f.o.} v$ if and only if $|u| > |v|$), then the function* `minus` *of example 1 has only one size-change graph defined by:*

$$G : \mathtt{minus} \to \mathtt{minus}$$

$$x \overset{\downarrow}{\to} x$$

$$y \overset{\downarrow}{\to} y$$

G^ω is a size-change multipath and $(x \overset{\downarrow}{\to} x)^\omega$ is a thread of this multipath, where A^ω defines a possibly infinite number of occurrences of A.

Theorem 8.2 [Lee et al. 2001]. *A program $\mathbf{p}$ is terminating if each infinite size-change multipath has a thread with infinitely many arcs labeled by $\downarrow$.*

Now we try to combine this result with the sup-interpretations.

Definition 8.3. (θ-Size-change graphs) Given a program $\mathbf{p}$ and a sup-interpretation θ, a θ-size-change graph, noted $G_\theta : \mathtt{f} \to \mathtt{g}$, is a size-change graph $G : \mathtt{f} \to \mathtt{g}$ corresponding to the activation of an expression $\mathtt{g}(d_1, \cdots, d_m)$ by $\mathtt{f}(p_1, \cdots, p_n)$ and which is modified by:

—The nodes $\theta^*(p_1), \cdots, \theta^*(p_n)$, $\theta^*(d_1), \cdots, \theta^*(d_m)$ are the sup-interpretations of the function arguments.
—There is an arc from $\theta^*(p_i)$ to $\theta^*(d_j)$ iff $\theta^*(p_i) \geq \theta^*(d_j)$.
—Moreover, if $\theta^*(p_i) > \theta^*(d_j)$, then the arc is labeled by $\downarrow$.

A θ-size-change multipath is a possibly infinite sequence $G^1_\theta, G^2_\theta, \ldots$ of size-change graphs with sup-interpretation θ.

[1]Notice that if $d_j\sigma$ is not a value then $p_i\sigma \geq_{w.f.o.} d_j\sigma$ cannot be checked and, consequently, no arc is added in the corresponding size-change graph.

THEOREM 8.4. *Given a sup-interpretation θ whose codomain is included in the set of functions from $\mathbb{N}$ to $\mathbb{N}$, a program $\boldsymbol{p}$ is terminating if each infinite θ-size-change multipath has a thread with infinitely many arcs labeled by $\downarrow$.*

PROOF. The well-foundedness considered in Theorem 8.2 is replaced by the fact that the sup-interpretation of a closed expression is a natural number. Thus, an arc $\theta^*(p_i)\xrightarrow{\downarrow}\theta^*(e_j)$ of the θ-size-change graph G^k_θ from $\mathtt{f}_k$ to $\mathtt{f}_{k+1}$ corresponds to the activation of an expression $\mathtt{f}_{k+1}(e_1,\cdots,e_m)$ by $\mathtt{f}_k(p_1,\cdots,p_n)$. By definition of $\downarrow$, $\theta^*(p_i)\xrightarrow{\downarrow}\theta^*(e_j)$ iff $\theta^*(p_i) > \theta^*(e_j)$. The strict inequality corresponds to a decrease, by some fixed constant. By hypothesis, every infinite multigraph has at least one thread with infinitely many arcs of this shape. As a consequence, the program is terminating. □

This Theorem is an application of the size-change principle method. However, it is not just an instance of Theorem 8.2. In fact, Jones et al. were considering only well-founded orders on values, whereas Theorem 8.4, allows to deal with any expression, if its sup-interpretation is defined. Consequently, it allows to show the termination of more algorithms, as illustrated by the following example:

EXAMPLE 17. *Taking $\theta(\mathtt{minus})(X,Y) = X$ and $\theta(\mathbf{S})(X) = X+1$, the program* `q` *of example 1 has three size-change graphs defined by:*

$G^1_\theta : \mathtt{minus} \to \mathtt{minus}$	$G^2_\theta : \mathtt{q} \to \mathtt{minus}$	$G^3_\theta : \mathtt{q} \to \mathtt{q}$
$U+1\xrightarrow{\downarrow}U$	$Z+1\xrightarrow{\downarrow}Z$	$Z+1\xrightarrow{\downarrow}Z$
$V+1\xrightarrow{\downarrow}V$	$U+1\xrightarrow{\downarrow}U$	$U+1\to U+1$

The infinite θ-size-change multipaths starting from `q` *are all of the shape $G^{3^\omega}_\theta, G^{2^\omega}_\theta$, $G^{1^\omega}_\theta$, where G^ω defines a possibly infinite number of occurrences of G. However they all contain a thread of the shape $(Z+1\xrightarrow{\downarrow}Z)^\omega$,$Z+1\xrightarrow{\downarrow}Z$,$(U+1\xrightarrow{\downarrow}U)^\omega$ with infinitely many arcs labeled by $\downarrow$. Notice that this example is not captured by Theorem 8.2 since the symbol* `minus` *is a function symbol and cannot be compared with other values.*

REFERENCES

AMADIO, R. 2003. Max-plus quasi-interpretations. In *TLCA*. Lecture Notes in Computer Science, vol. 2701. Springer, 31–45.

AMADIO, R., COUPET-GRIMAL, S., DAL-ZILIO, S., AND JAKUBIEC, L. 2004. A functional scenario for bytecode verification of resource bounds. In *CSL*. Lecture Notes in Computer Science, vol. 3210. Springer, 265–279.

AMADIO, R. AND DAL-ZILIO, S. 2004. Resource control for synchronous cooperative threads. In *CONCUR*. Lecture Notes in Computer Science, vol. 3170. Springer, 68–82.

ANDERSON, H. AND KHOO, S. 2003. Affine-based size-change termination. *APLAS*. Lecture Notes in Computer Science, vol. 2895. Springer, 122–140.

ARTS, T. AND GIESL, J. 2000. Termination of term rewriting using dependency pairs. *Theoretical Computer Science 236*, 133–178.

ASPINALL, D. AND COMPAGNONI, A. 2003. Heap bounded assembly language. *Journal of Automated Reasoning (Special Issue on Proof-Carrying Code) 31*, 261–302.

AVERY, J. 2006. Size-change termination and bound analysis. In *FLOPS*. Lecture Notes in Computer Science, vol. 3945. Springer, 192–207.

Bird, R. and Wadler, P. 1988. *Introduction to Functional Programming*. Prentice-Hall, New York, NY.

Blum, M. 1967. A machine-independent theory of the complexity of recursive functions. *Journal of the Association for Computing Machinery 14*, 322–336.

Bonfante, G., Marion, J.-Y., and Moyen, J.-Y. 2001. On lexicographic termination ordering with space bound certifications. In *PSI*. Lecture Notes in Computer Science, vol. 2244. Springer.

Bonfante, G., Marion, J.-Y., and Moyen, J.-Y. 2007. Quasi-interpretations, a way to control resources. *Theoretical Computer Science, Accepted*.

Bonfante, G., Marion, J.-Y., Moyen, J.-Y., and Péchoux, R. 2005. Synthesis of quasi-interpretations. *LCC, LICS Satellite Workshop*. `http://hal.inria.fr`.

Chandra, A., Kozen, D., and Stockmeyer, L. 1981. Alternation. *Journal of the ACM 28*, 114–133.

Hofmann, M. 1999. Linear types and non-size-increasing polynomial time computation. In *LICS*. 464–473.

Hofmann, M. 2000. A type system for bounded space and functional in-place update. In *ESOP*. Lecture Notes in Computer Science, vol. 1782. 165–179.

Huet, G. 1980. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM 27,* 4, 797–821.

Jones, N. and Kristiansen, L. 2005. The flow of data and the complexity of algorithms. *Lecture notes in computer science 3526*, 263–274.

Jones, N. D. 1997. *Computability and complexity, from a programming perspective*. MIT press.

Lee, C. S., Jones, N., and Ben-Amram, A. 2001. The Size-Change Principle for Program Termination. In *POPL*. Vol. 28. ACM press, 81–92.

Marion, J.-Y. 2003. Analysing the implicit complexity of programs. *Information and Computation 183*, 2–18.

Marion, J.-Y. and Moyen, J.-Y. 2000. Efficient first order functional program interpreter with time bound certifications. In *LPAR*. Lecture Notes in Computer Science, vol. 1955. Springer, 25–42.

Marion, J.-Y. and Péchoux, R. 2006. Resource analysis by sup-interpretation. In *FLOPS*. Lecture Notes in Computer Science, vol. 3945. Springer, 163–176.

Niggl, K. and Wunderlich, H. 2006. Certifying Polynomial Time and Linear/Polynomial Space for Imperative Programs. *SIAM Journal on Computing 35*, 1122.

Savitch, W. J. 1970. Relationship between nondeterministic and deterministic tape classes. *Journal of Computer System Science 4*, 177–192.

Thiemann, R. and Giesl, J. 2005. The size-change principle and dependency pairs for termination of term rewriting. *Applicable Algebra in Engineering, Communication and Computing 16,* 4, 229–270.

Received October 2006; revised April 2008; accepted June 2008