Quasi-friendly sup-interpretations

Jean-Yves Marion, Romain Péchoux

Quasi-friendly sup-interpretations

Jean-Yves Marion and Romain Péchoux

Loria, Calligramme project, B.P. 239, 54506 Vandœuvre-lès-Nancy Cedex, France, and École Nationale Supérieure des Mines de Nancy, INPL, France.
Jean-Yves.Marion@loria.fr Romain.Pechoux@loria.fr

Abstract. In a previous paper [16], the sup-interpretation method was proposed as a new tool to control memory resources of first order functional programs with pattern matching by static analysis. Basically, a sup-interpretation provides an upper bound on the size of function outputs. In this former work, a criterion, which can be applied to terminating as well as non-terminating programs, was developed in order to bound polynomially the stack frame size. In this paper, we suggest a new criterion which captures more algorithms computing values polynomially bounded in the size of the inputs. Since this work is related to quasi-interpretations, we compare the two notions obtaining two main features. The first one is that, given a program, we have heuristics for finding a sup-interpretation when we consider polynomials of bounded degree. The other one consists in the characterizations of the set of function computable in polynomial time and in polynomial space.

1 Introduction

This paper is part of general investigation on program complexity analysis and, particularly, on first order functional programming static analysis. It studies the notion of sup-interpretation introduced in [16], a method that provides an upper bound on the size of every stack frame if the program is non-terminating, and establishes an upper bound on the size of function outputs if the program is terminating. Basically, a sup-interpretation is a *partial* assignment of symbols, which ranges over positive real numbers and which gives a bound on the size of the computed values. We use this notion to develop a criterion which ensures that the size of the values computed by a program verifying this criterion is polynomially bounded in the size of the inputs and which allows to bound polynomially the size of the stack frames whenever the program is not terminating.

The practical issue is to provide the amount of space resources that a program needs during its execution. This is crucial for at least many critical applications, and is of real interest in computer security. There are several approaches which are trying to solve the same problem. The first one is by monitoring computations. However, the monitor may crash unpredictably by memory leak if it is compiled with the program. The second one, complementary to static analysis, is a testing-based approach. Indeed, such an approach provides lower bounds on the memory needed. The last approach is type checking which can be done by a bytecode verifier. Our approach is rather distinct 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 studied by Hofmann [10, 11] and Aspinall and Compagnoni [5].

The sup-interpretation can be considered as some program annotation provided by the programmer. Sup-interpretations strongly inherit from:

– The notion of quasi-interpretation developed by Bonfante, Marion and Moyen in [7, 8, 15, 14]. Quasi-interpretation, like sup-interpretation, provides a bound on function outputs by static analysis for first order functional programs and allows the programmer to find a bound on the size of every stack frame. The paper [8] 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 or yet the set of polynomial space functions. Like quasi-interpretations, sup-interpretations, were developed with the aim to pay more attention to the algorithmic aspects of complexity than to the functional (or extensional) one and then it is part of study of the implicit complexity of programs. But the main interest of sup-interpretation is to capture a larger class of algorithms. In fact, programs computing logarithm or division admits 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 bytecodeverifier by following the lines of [2] or language with synchronous cooperative threads as in [3].
– The dependency pair methods introduced by Arts and Giesl in [4] which was initially introduced for proving termination of term rewriting systems automatically. In order to obtain a polynomial space bound, a criterion is developed on sup-interpretations using the underlying notion of dependency pairs by Arts and Giesl [4].
– The size-change principle by Jones et al. [13] which is another method developed for proving program termination. Indeed, there is a very strong relation between termination and computational complexity and, in order to prove both complexity bounds and termination, 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 a program outputs. In section 5, we introduce a criterion, called quasi-friendly criterion, which enlarges, in practice, the class of programs captured by a former criterion, called friendly criterion, of [16] (for example, it captures algorithms over trees whereas the friendly criterion fails). This criterion provides a polynomial bound on the size of the values and the stack frame size computed by a quasi-friendly programs (depending on whether the programs terminate or not). Finally, in a last section, we also compare the notion of sup-interpretation to the one of quasi-interpretation. First, we show that quasi-interpretation is a particular sup-interpretation. As a consequence, we obtain heuristics for the synthesis of sup-interpretations, which consists in finding a sup-interpretation for a given program, as far as far, we consider the set of **Max-Poly** functions defined to be constant functions, projections, max, $+$, $\times$ and closed by composition. Finally, using former results about quasi-interpretations, we give two characterizations of the sets of functions computable in polynomial time and respectively polynomial space.

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 basically function symbols definitions and which are characterized by the following grammar:

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

where $x, x_1, \ldots, x_n$ are variables, $\mathbf{c} \in Cns$ 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 extend this notation $\overline{e}$ in a clarity concern 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 that allows pattern matching. It is convenient, because it avoids tedious details, to restrict case definitions in such a way that an expression involved in a **Case** expression does not contain nested **Case** (In other words, an expression e^j does not contain a **Case** expression). This is not a severe restriction since a program involving nested **Case** can be transformed in linear time in its size into an equivalent program without the nested **Case** construction.

In a definition, a variable of $e^{\mathtt{f}}$ is either a variable in the parameter list $x_1, \cdots, x_n$ of the definition of $\mathtt{f}$ or a variable which occurs in a pattern of a **Case** definition. In a **Case** expression, patterns are not overlapping. Such a restriction ensures that considered programs are confluent.

2.2 Semantics

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

A substitution σ is a finite mapping from Var to $\mathcal{V}$. The application of a substitution σ to an expression e is noted $e\sigma$.

The language has a closure-based call-by-value semantics which is displayed in Appendix A. Given a substitution σ, the meaning of $e\sigma \downarrow w$ is that is that e evaluates to the value w of $\mathcal{V}^*$. If no rule is applicable, then an error occurs, and $e\sigma \downarrow \mathbf{Err}$. A program $\mathbf{p}$ computes a partial function $[\![\mathbf{p}]\!] : \mathcal{V}^n \to \mathcal{V}^*$ defined by: For all $v_i \in \mathcal{V}$, $[\![\mathbf{p}]\!](v_1, \cdots, v_n) = w$ iff $\mathbf{p}(v_1, \cdots, v_n) \downarrow w$.

Example 1 (Division). Consider the following definitions that encode the division:

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

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

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

3 Fraternities

In this section, we define the notion of fraternity based on dependency pairs, that Arts and Giesl [4] introduced to prove termination automatically. Fraternities will be used to tame the size of arguments of recursive calls.

A *context* is an expression $\mathsf{C}[\diamond_1, \cdots, \diamond_r]$ containing one 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 1. *Assume that* $\mathtt{f}(x_1, \cdots, x_n) = e^{\mathtt{f}}$ *is a definition of a program.* An expression d is activated *by* $\mathtt{f}(p_1, \cdots, p_n)$ *where the* p_i*'s are patterns if there is a context with one hole* $\mathsf{C}[\diamond]$ *such that:*

- *If* $e^{\mathtt{f}}$ *is a compositional expression (that is with no case definition inside it), then* $e^{\mathtt{f}} = \mathsf{C}[d]$. *In this case,* $p_1 = x_1 \ldots p_n = x_n$.

– *Otherwise,* $e^{\mathtt{f}} = \textbf{Case } e_1, \cdots, e_n \textbf{ of } \overline{q_1} \to e^1 \ldots \overline{q_\ell} \to e^\ell$*, then there is a position* j *such that* $e^j = \mathsf{C}[d]$*. In this case,* $p_1 = q_{j,1} \ldots p_n = q_{j,n}$ *where* $\overline{q_j} = q_{j,1} \ldots q_{j,n}$*.*

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, such that $\mathsf{C}[d]$ is activated by $\mathtt{f}(p_1, \cdots, p_n)$.

Definition 2 (Precedence). *The notion of activated expression 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 themselves recursively.*

Definition 3 (Fraternity). *In a program* $\boldsymbol{p}$*, 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 maximal*
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 2. The program of example 1 admits two fraternities $\mathtt{minus}(u, v)$ and $\mathbf{S}[\mathtt{q}(\mathtt{minus}(z, u), \mathbf{S}(u))]$ which are respectively activated by $\mathtt{minus}(\mathbf{S}(u), \mathbf{S}(v))$ and $\mathtt{q}(\mathbf{S}(z), \mathbf{S}(u))$.

4 Sup-interpretations

Definition 4 (Partial assignment). *A partial assignment* I *is a partial mapping from the vocabulary* Σ *which assigns a partial function* $I(b) : (\mathbb{R}^+)^n \longmapsto \mathbb{R}^+$ *to each symbol* b *in the domain of* I*. The domain of a partial assignment* I *is noted* $dom(I)$*. Because it is convenient, we shall always assume that partial assignments that we consider, are defined on constructor and operator symbols (i.e.* $Cns \cup Op \subseteq dom(I)$*).*

An assignment I *is defined over an expression* e *if each symbol of* $Cns \cup Op \cup Fct$ *in* e *belongs to* $dom(I)$*. 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 $\overline{e}$ is a sequence of n expressions, then $I^*(\overline{e}) = \max(I^*(e_1), \ldots, I^*(e_n))$*
3. *If e is a* **Case** *expression of the shape* **Case** $\overline{e}$ **of** $\overline{p_1} \to e^1 \ldots \overline{p_\ell} \to e^\ell$*, then $I^*(e) = \max(I^*(\overline{e}), I^*(e^1), \ldots, I^*(e^\ell))$*
4. *If b is a 0-ary symbol or $b =$* **Err***, then $I^*(b) = I(b)$.*
5. *If b is a symbol of arity $n > 0$ and $e_1, \cdots, e_n$ are expressions, then we have $I^*(b(e_1, \cdots, e_n)) = I(b)(I^*(e_1), \ldots, I^*(e_n))$*

Definition 5 (Additive assignments). *A partial assignment I is polynomial if for each symbol b of arity n of dom(I), $I(b)$ is* ***bounded*** *by a polynomial in $\mathbb{R}^+[X_1, \cdots, X_n]$. An assignment of a constructor symbol $\mathbf{c}$ is* additive *if*

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

If the polynomial assignment of each constructor symbol is additive then the assignment is additive. ***Throughout the following paper we only consider additive assignments****.*

Definition 6. *The size of an expression e is noted $|e|$ and defined by $|e| = 0$ if e is a 0-ary symbol or if $e =$* **Err** *and $|b(e_1, \ldots, e_n)| = 1 + \sum_i |e_i|$ if $e = b(e_1, \ldots, e_n)$ with $n > 0$.*

Lemma 1. *Given an assignment I, there is a constant α such that for each value v of $\mathcal{V}^*$, the following inequality is satisfied :*

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

Definition 7 (Sup-interpretation). *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 $b \in$ dom(θ), 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 $v \in \mathcal{V}^*$,*

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

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

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

We say that expression e admits a sup-interpretation θ if θ is defined over e. The sup-interpretation of e wrt θ is $\theta^*(e)$.

Intuitively, the 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 sup-interpretation is a complexity measure in the sense of Blum [6].

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

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

Lemma 2. *Let e be an expression with no variable and which admits a sup-interpretation θ. If $[\![e]\!] \in \mathcal{V}^*$ then have:*

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

Proof. The proof is in [16]. □

Example 3. Consider the program for exponential:

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

By taking $\theta(\mathbf{S})(X) = X+1$, $\theta(\mathtt{double})(X) = 2X$, we define a sup-interpretation of the function symbol `double`.

Now we are going to define the notion of weight which allows us to control the size of the arguments in recursive calls. A weight is an assignment having the subterm property but no longer giving a bound on the size of a value computed by a function.

Definition 8 (Weight). *A weight ω is a partial assignment which ranges over Fct. To a given function symbol `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*

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

Definition 9 (Call-tree). *A* state *is a tuple $\langle \mathtt{f}, u_1, \cdots, u_n \rangle$ where `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 $\mathtt{f}(p_1, \cdots, p_n)$. A* transition *is noted $\eta_1 \rightsquigarrow \eta_2$ and defined by:*

1. *There is a substitution* σ *such that* $p_i\sigma = u_i$ *for* $i = 1, \ldots, n$
2. *and* $[\![e_j\sigma]\!] = v_j$ *for* $j = 1, \ldots, k$.

We call such a graph a call-tree of $\mathtt{f}$ *over values* $u_1, \ldots, u_n$ *if* $\langle \mathtt{f}, u_1, \cdots, u_n\rangle$ *is its root. A state may be seen as a stack frame. 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* $\mathtt{f}(u_1, \ldots, u_n)$.

5 Criterion to control space resources

Definition 10 (Quasi-friendly). *A program* $\boldsymbol{p}$ *is* quasi-friendly *iff there are a sup-interpretation* θ *and a 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:*

1. $\omega_{\mathtt{f}}(\theta^*(p_1), \ldots, \theta^*(p_n)) \geq \max_{i=1..r}(\omega_{\mathtt{g}_i}(\theta^*(\overline{e_i})))$
2. $\omega_{\mathtt{f}}(\theta^*(p_1), \ldots, \theta^*(p_n)) \geq \theta^*(\mathsf{C})[\omega_{\mathtt{g}_1}(\theta^*(\overline{e_1})), \ldots, \omega_{\mathtt{g}_r}(\theta^*(\overline{e_r}))]$

Notice that nested fraternities (i.e. a fraternity d containing another fraternity inside it) are not of real interest for this criterion. In fact, consider for example the following nested fraternity $\mathtt{f}(x) = \mathtt{f}(\mathtt{f}(x))$. In the quasi-friendly criterion, one need to guess a weight and a sup-interpretation for the function symbol $\mathtt{f}$, so that, 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 the Ackermann one.

Since θ^* has no subterm property, conditions 1 and 2 are independent and useful in order to control the size of the values added by recursive calls. An example showing this independence is given in appendix B.

Theorem 1. *Assume that* $\boldsymbol{p}$ *is a quasi-friendly program, then for each function symbol* $\mathtt{f}$ *of* $\boldsymbol{p}$ *there is a polynomial* P *such that for every value* $v_1, \ldots, v_n$,

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

Proof. The proof can be found in appendix C. □

Example 4. The program of example 1 is quasi-friendly. Taking:

$$\begin{array}{rl|c} \theta(\mathbf{S})(X) = X+1 & & \theta(\mathbf{0}) = 0 \\ \theta(\mathtt{minus})(X,Y) = X & & \omega_{\mathtt{minus}}(X,Y) = \max(X,Y) \\ \omega_{\mathtt{q}}(X,Y) = X+Y & & \end{array}$$

We check the conditions for the fraternity defined by q:

$$\begin{aligned}
\omega_{\mathtt{q}}(\theta^*(\mathbf{S}(z)), \theta^*(\mathbf{S}(u))) &= U + Z + 2 \\
&\geq Z + U + 1 \\
&= \omega_{\mathtt{q}}(\theta^*(\mathtt{minus}(z,u)), \theta^*(\mathbf{S}(u))) && \text{(Condition 1)} \\
\omega_{\mathtt{q}}(\theta^*(\mathbf{S}(z)), \theta^*(\mathbf{S}(u))) &= U + Z + 2 \\
&\geq Z + U + 2 \\
&= \theta^*(\mathbf{S})(\omega_{\mathtt{q}}(\theta^*(\mathtt{minus}(z,u)), \theta^*(\mathbf{S}(u)))) && \text{(Condition 2)}
\end{aligned}$$

Example 5. The program of example 3 is not quasi-friendly. Indeed since the sup-interpretation of `double` is greater than $2X$. One has to find a polynomial weight ω_{exp} such that:

$$\omega_{\mathrm{exp}}(X+1) \geq \theta(\mathtt{double})(\omega_{\mathrm{exp}}(X)) \geq 2\omega_{\mathrm{exp}}(X)$$

which is impossible.

Theorem 2. *Assume that* $\boldsymbol{p}$ *is a quasi-friendly program. For each function symbol* f *of* $\boldsymbol{p}$ *there is a polynomial* R *such that for every node* $\langle \mathtt{g}, u_1, \cdots, u_m\rangle$ *of the call-tree of root* $\langle \mathtt{f}, v_1, \cdots, v_n\rangle$*,*

$$\max_{j=1..m}(|u_j|) \leq R(\max(|v_1|, ..., |v_n|))$$

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

Proof. The proof relies on theorem 1 and is essentially the same than the one in [16]. □

In the paper [16], a first criterion, called friendly criterion, was developed in order to bound the stack frame size during the execution of a program. However, as mentioned in the conclusion of [16], this criterion was suffering from a lack because of a too restrictive condition on the contexts. Indeed, the sup-interpretations of the contexts were forced to be max functions forbidding, for example, recursion over tree data structure as in the example of Appendix D. Thus, from practical experience, the quasi-friendly criterion captures more algorithms than the friendly criterion.

6 Comparison with quasi-interpretations

Definition 11. *A quasi-interpretation is a total (i.e. defined for every symbol of the program) additive assignment* $(\!|-|\!)$ *monotonic and having the subterm property (i.e. For all symbol* f *of arity* $n, \forall i \in \{1, n\}, (\!|\mathtt{f}|\!)(\ldots, X_i, \ldots) \geq X_i$*) such that for every maximal expression* e *activated by* $\mathtt{f}(p_1, \cdots, p_n)$ *we have:*

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

where the assignment $(\!|-|\!)$ *is extended canonically to terms by*

$$(\!|\mathtt{g}(e_1, \cdots, e_n)|\!) = (\!|\mathtt{g}|\!)((\!|e_1|\!), \ldots, (\!|e_n|\!))$$

As demonstrated in [7, 8, 15], quasi-interpretations have the following property:

Proposition 1. *Given a program* $\boldsymbol{p}$ *which admits a 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 3. *Every quasi-interpretation is a sup-interpretation.*

Proof. By previous proposition, conditions 2 and 3 of Definition 7 hold. By Definition 11, a quasi-interpretation is monotonic, so that condition 1 of Definition 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 [1] for quasi-interpretations. This problem is very relevant in a perspective of automating the complexity analysis of programs. However the synthesis of quasi-interpretation is a very tricky problem which is undecidable in general. However Amadio showed [1] that some rich classes of quasi-interpretation are in NP and in [9], 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-interpretation in **Max-Poly**, the set of functions defined to be constant functions, projections, max, $+$, $\times$ and closed by composition: Given a program $\mathbf{p}$, we try to find a quasi-interpretation for this program, and, by previous Theorem, we know that it is a sup-interpretation.

Theorem 4. *Every program that admits a quasi-interpretation is quasi-friendly.*

Proof. By previous theorem every quasi-interpretation defines a sup-interpretation. Moreover every quasi-interpretation is a weight.

Proposition 2. *There exist quasi-friendly programs that do not have any 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 q. For the last definition, we have:

$$\begin{aligned}
(\!|\mathsf{q}(\mathbf{S}(v), \mathbf{S}(u))|\!) &= (\!|\mathsf{q}|\!)(U + k, V + k) && \text{For some constant } k \\
&\geq (\!|\mathbf{S}(\mathtt{q}(\mathtt{minus}(v, u), \mathbf{S}(u)))|\!) && \text{By Dfn of } (\!|-|\!) \\
&\geq k + (\!|\mathsf{q}|\!)(\max(U, V), U + k) && \\
&> (\!|\mathsf{q}|\!)(U + k, V + k) && \text{for } V \geq U + 1
\end{aligned}$$

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

In [7, 8, 15], some characterizations of the functions computable in polynomial time and polynomial space were given. Theorems 1 and 3 allow to adapt these results to the sup-interpretations.

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

Definition 12. *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 13. *Given a precedence $\geq_{Fct \cup Cns}$ and a status st, we define the recursive path ordering $\prec_{rpo}$ as follows:*

$$\frac{u \preceq_{rpo} t_i}{u \prec_{rpo} f(\ldots, t_i, \ldots)} \qquad \frac{\forall i\ u_i \prec_{rpo} f(t_1, \cdots, t_n) \qquad g \geq_{Fct \cup Cns} f}{g(u_1, \cdots, u_m) \prec_{rpo} f(t_1, \cdots, t_n)}$$

$$\frac{(u_1, \cdots, u_n) \prec_{rpo}^{st(f)} (t_1, \cdots, t_n) \qquad f \approx_{Fct \cup Cns} g \qquad \forall i\ u_i \prec_{rpo} f(t_1, \cdots, t_n)}{g(u_1, \cdots, u_n) \prec_{rpo} f(t_1, \cdots, t_n)}$$

The **Case**...**of**... $\rightarrow$ *(and the symbol = in a definition without* **Case***) expressions induce a rewrite relation noted* $\rightarrow$. *A program is ordered by* $\prec_{rpo}$ *if there are a precedence* $\preceq_{Fct}$ *and a status st such that for each rule* $l \rightarrow r$ *of the rewrite relation, the inequality* $r \prec_{rpo} l$ *holds.*

Theorem 5.

- *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. The main idea of the proof is fully written in [8]. Due to the $\prec_{rpo}$ ordering with product status, any recursive subcall 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 1 is that any other subcalls will be done on arguments of polynomial size. So one may use a memoization technique a la Jones [12], which leads us to define a call-by-value interpreter with cache in Appendix E. $\square$

References

1. R. Amadio. Max-plus quasi-interpretations. In Martin Hofmann, editor, *Typed Lambda Calculi and Applications, 6th International Conference, TLCA 2003, Valencia, Spain, June 10-12, 2003, Proceedings*, volume 2701 of *Lecture Notes in Computer Science*, pages 31–45. Springer, 2003.
2. R. Amadio, S. Coupet-Grimal, S. Dal-Zilio, and L. Jakubiec. A functional scenario for bytecode verification of resource bounds. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *Computer Science Logic, 18th International Workshop, CSL 13th Annual Conference of the EACSL, Karpacz, Poland*, volume 3210 of *Lecture Notes in Computer Science*, pages 265–279. Springer, 2004.
3. R. Amadio and S. Dal Zilio. Resource control for synchronous cooperative threads. Research Report LIF.
4. T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.
5. D. Aspinall and A. Compagnoni. Heap bounded assembly language. *Journal of Automated Reasoning (Special Issue on Proof-Carrying Code)*, 31:261–302, 2003.
6. M. Blum. A machine-independent theory of the complexity of recursive functions. *Journal of the Association for Computing Machinery*, 14:322–336, 1967.
7. G. Bonfante, J.-Y. Marion, and J.-Y. Moyen. On lexicographic termination ordering with space bound certifications. In *PSI 2001, Akademgorodok, Novosibirsk, Russia, Ershov Memorial Conference*, volume 2244 of *Lecture Notes in Computer Science*. Springer, Jul 2001.
8. G. Bonfante, J.-Y. Marion, and J.-Y. Moyen. Quasi-interpretation a way to control resources. *Submitted to Theoretical Computer Science*, 2005. `http://www.loria.fr/~marionjy`.
9. G. Bonfante, J.-Y. Marion, J.-Y. Moyen, and R. Péchoux. Synthesis of quasi-interpretations. *Workshop on Logic and Complexity in Computer Science, LCC2005, Chicago*, 2005. `http://www.loria/~pechoux`.
10. M. Hofmann. Linear types and non-size-increasing polynomial time computation. In *Proceedings of the Fourteenth IEEE Symposium on Logic in Computer Science (LICS'99)*, pages 464–473, 1999.
11. M. Hofmann. A type system for bounded space and functional in-place update. In *European Symposium on Programming, ESOP'00*, volume 1782 of *Lecture Notes in Computer Science*, pages 165–179, 2000.
12. N. D. Jones. *Computability and complexity, from a programming perspective*. MIT press, 1997.
13. Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *Symposium on Principles of Programming Languages*, volume 28, pages 81–92. ACM press, january 2001.
14. J.-Y. Marion. Analysing the implicit complexity of programs. *Information and Computation*, 183:2–18, 2003.
15. J.-Y. Marion and J.-Y. Moyen. Efficient first order functional program interpreter with time bound certifications. In Michel Parigot and Andrei Voronkov, editors, *Logic for Programming and Automated Reasoning, 7th International Conference, LPAR 2000, Reunion Island, France*, volume 1955 of *Lecture Notes in Computer Science*, pages 25–42. Springer, Nov 2000.
16. J.-Y. Marion and R. Péchoux. Resource analysis by sup-interpretation. In M. Hagiya and P. Wadler, editors, *Functional and Logic Programming: 8th International Symposium, FLOPS 2006*, volume 3945 of *Lecture Notes in Computer Science*, pages 163–176, 2006.

A Call-by-value semantics

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

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

$$\frac{e \downarrow u \quad \exists \sigma,\ i\ :\ p_i\sigma = u \quad e_i\sigma \downarrow w}{\mathbf{Case}\ e\ \mathbf{of}\ p_1 \to e_1 \ldots p_\ell \to e_\ell \downarrow w} \ \mathbf{Case} \text{ and } u \neq \mathbf{Err}$$

$$\frac{e_1 \downarrow w_1 \ldots e_n \downarrow w_n \quad \mathtt{f}(x_1, \cdots, x_n) = e^{\mathtt{f}} \quad e^{\mathtt{f}}\sigma \downarrow w}{\mathtt{f}(e_1, \cdots, e_n) \downarrow w} \text{ where } \sigma(x_i) = w_i \neq \mathbf{Err} \text{ and } w \neq \mathbf{Err}$$

Fig. 1. Call by value semantics of ground expressions wrt a program **p**

B Example

The following non-terminating program illustrates that conditions 1 and 2 of the quasi-friendly criterion are independent.

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

where `double` is the function of example 3. The arguments of `f` computed by the recursive calls are unbounded. However by taking $\theta(\mathtt{half})(X) = X/2$, $\theta(\mathtt{double})(X) = 2X$ and $\omega_{\mathtt{f}}(X) = X$, we can check that the Condition 2 of the quasi-friendly criterion is satisfied, even if Condition 1 is not.

C Proof of Theorem 1

We start by showing the following lemma:

Lemma 3. *If a locally friendly program has a call-tree containing a branch of the shape* $\langle \mathtt{f}, u_1, \cdots, u_n\rangle \overset{*}{\rightsquigarrow} \langle \mathtt{g}, v_1, \cdots, v_k\rangle$ *with* $\mathtt{f} \approx_{Fct} \mathtt{g}$ *then:*

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

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

– If $n = 1$, $\langle \mathtt{f}, u_1, \cdots, u_n\rangle \rightsquigarrow \langle \mathtt{g}, v_1, \cdots, v_k\rangle$ then there is a definition with a fraternity of the shape $\mathtt{f}(x_1, \cdots, x_n) =$ **Case** $x_1, \cdots, x_n$ **of** $p_1, \cdots, p_n \to \mathtt{C}[\mathtt{g}(e_1, \cdots, e_k)]$ with $\mathtt{f} \approx_{Fct} \mathtt{g}$ and a substitution σ such that $p_i\sigma = u_i$ and $[\![e_j\sigma]\!] = v_j$. Applying the Condition 1 of the quasi-friendly criterion, we obtain:

$$\omega_{\mathtt{f}}(\theta^*(u_1), \cdots, \theta^*(u_n)) \geq \omega_{\mathtt{g}}(\theta^*(e_1\sigma), \cdots, \theta^*(e_k\sigma)) \geq \omega_{\mathtt{g}}(\theta^*(v_1), \cdots, \theta^*(v_k))$$

By monotonicity of weights and by definition of sup-interpretations.

– Now suppose by induction hypothesis that if $\langle \mathtt{f}, u_1, \cdots, u_n\rangle \overset{k}{\rightsquigarrow} \langle \mathtt{g}, v_1, \cdots, v_k\rangle$ with $\mathtt{f} \approx_{Fct} \mathtt{g}$ and $k \leq n$, we have

$$\omega_{\mathtt{f}}(\theta^*(u_1), \cdots, \theta^*(u_n)) \geq \omega_{\mathtt{g}}(\theta^*(v_1), \cdots, \theta^*(v_k)) \quad (I.H.)$$

And consider the following branch of length $n+1$:

$$\langle \mathtt{f}, u_1, \cdots, u_n\rangle \overset{n}{\rightsquigarrow} \langle \mathtt{g}, v_1, \cdots, v_k\rangle \rightsquigarrow \langle \mathtt{h}, v'_1, \cdots, v'_l\rangle$$

with $\mathtt{h} \approx_{Fct} \mathtt{f}$. Then as in the base case, we can derive

$$\omega_{\mathtt{g}}(\theta^*(v_1), \cdots, \theta^*(v_k)) \geq \omega_{\mathtt{h}}(\theta^*(v'_1), \cdots, \theta^*(v'_l))$$

and combine it with the Induction Hypothesis to obtain:

$$\omega_{\mathtt{f}}(\theta^*(u_1), \cdots, \theta^*(u_n)) \geq \omega_{\mathtt{h}}(\theta^*(v'_1), \cdots, \theta^*(v'_l))$$

□

Theorem 1. *Assume that* $\mathbf{p}$ *is a quasi-friendly program. For each function symbol* $\mathtt{f}$ *of* $\mathbf{p}$ *there is a polynomial* P *such that for every value* $v_1, \ldots, v_n$*,*

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

Proof. Suppose that we have a program $\mathbf{p}$ and 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 are going to show the previous result by an induction on the precedence $\geq_{Fct}$.

– If $\mathtt{f}$ is defined without function symbols (i.e. $\mathtt{f}$ is strictly smaller than any other function symbol for $\geq_{Fct}$), then a definition of the shape $\mathtt{f}(x_1, \cdots, x_n) = e$ with $e \in \mathcal{T}(Cns \cup \mathcal{X})$ is applied. We define $P_{\mathtt{f}}(X) = |e|$ with the size of a variable y being defined by $|y| = X$. Taking a substitution σ such that $p_i\sigma = v_i$, we can check easily that

$$P_{\mathtt{f}}(\max_{i=1..n} |v_i|) = |e[X := \max_{i=1..n} |v_i|]| \geq |e\sigma| = \|\mathtt{f}(v_1, \cdots, v_n)\|$$

where $|e[X := |v|]|$ denotes the substitution of the variable X by the value $|v|$ in the function $|e|$.

– Now, if the function symbol $\mathtt{f}$ is defined without fraternities, then we have definitions of this shape $\mathtt{f}(x_1, \cdots, x_n) = \textbf{Case } x_1, \cdots, x_n \textbf{ of } p_1, \cdots, p_n \to e$ with for all function symbol $\mathtt{g} \in e, \mathtt{f} >_{Fct} \mathtt{g}$. We suppose by induction hypothesis that we have already defined a polynomial upper bound on the function symbols $\mathtt{g}$. Moreover, for every constructor symbol $\mathbf{c} \in e$ of arity n, we define $P_{\mathbf{c}}(X) = nX + 1$, which represents a polynomial upper bound on its computation (i.e. the constructor symbol keeps its arguments and adds 1 to the global size). Finally, if $e = \mathtt{h}(e_1, \cdots, e_m)$, we define inductively a polynomial upper bound on the size of the computation of e by $P_e(X) = P_{\mathtt{h}}(\max_{i=1..m} P_{e_i}(X))$. By definition of such a polynomial, we know that $P_e(\max_{i=1..n} |v_i|) \geq \|f(v_1, \cdots, v_n)\|$.
– Now, suppose that the function symbol is defined with some definitions leading to fraternities and some definitions similar to the one of the previous case (i.e. definitions which are not recursive). First, we build a polynomial $P_{\mathtt{f}>_{Fct}}$, as in the previous case, for these latter definitions. Notice also that since we know, by hypothesis, that the computation is terminating, every recursive call will be ended by such definitions. However it can be ended by such a definition for some other equivalent function symbol. Thus for each $\mathtt{g} \approx_{Fct} \mathtt{f}$, we also define $P_{\mathtt{g}>_{Fct}}$ and 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 dependency pair cycle in Arts and Giesl's work. Now, combining condition 2 of Definition 10 and lemma 3, we know that if for some values $v_1, \cdots, v_n$, $\mathtt{f}(v_1, \cdots, v_n) \xrightarrow{*} \mathsf{C}[\mathtt{g}_1(\overline{u_1}), \ldots, \mathtt{g}_r(\overline{u_l})]$ with $\mathtt{g}_1 \approx_{Fct} \ldots \approx_{Fct} \mathtt{g}_l \approx_{Fct} \mathtt{f}$ and $\to$ the rewrite relation induced by the definitions of the program, then:

$$\omega_{\mathtt{f}}(\theta^*(v_1), \cdots, \theta^*(v_n)) \geq \theta^*_{\overline{v}}(\mathsf{C})[\omega_{\mathtt{g}_1}(\theta^*_{\overline{v}}(\overline{u_1})), \ldots, \omega_{\mathtt{g}_l}(\theta^*_{\overline{v}}(\overline{u_l}))] \tag{1}$$

where the notation $\theta^*_{\overline{v}}(e)$ means that the sup-interpretation of e may depend on $\overline{v} = v_1, \cdots, v_n$.
This result holds particularly in the case where the $\mathtt{g}_i(\overline{u_i})$ 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. By condition 2 of Definition 7, we know that $\theta^*(\overline{u_i}) \geq |\overline{u_i}|$. By subterm property of weights, we obtain $\omega_{\mathtt{g}_i}(\theta^*(\overline{u_i})) \geq \max|\overline{u_i}|$ and since $Q_{\mathtt{f}}$ is monotone (by construction) $Q_{\mathtt{f}}(\omega_{\mathtt{g}_i}(\theta^*(\overline{u_i}))) \geq Q_{\mathtt{f}}(\max|\overline{u_i}|)$. Now, since sup-interpretations represent an upper bound on the values computed by the functions, if we have $\mathsf{C}[\mathtt{g}_1(\overline{u_1}), \ldots, \mathtt{g}_l(\overline{u_r})] \downarrow [\![\mathtt{f}]\!](v_1, \cdots, v_n)$ then by monotonicity of sup-

interpretations, weights and $Q_{\mathtt{f}}$:

$$\theta^*_{\overline{v}}(\mathsf{C})[Q_{\mathtt{f}}(\omega_{\mathtt{g}_1}(\theta^*_{\overline{v}}(\overline{u_1}))), \ldots, Q_{\mathtt{f}}(\omega_{\mathtt{g}_l}(\theta^*_{\overline{v}}(\overline{u_l})))] \geq$$
$$\theta^*_{\overline{v}}(\mathsf{C})[Q_{\mathtt{f}}(\max|\overline{u_1}|), \ldots, Q_{\mathtt{f}}(\max|\overline{u_l}|))] \geq \|\mathtt{f}(v_1, \cdots, v_n)\|$$

It remains to show that the left-hand side of this inequality is bounded polynomially in the size of the inputs. Inequality (1), implies that $\theta^*_{\overline{v}}(\mathsf{C})[\diamond_1, \cdots, \diamond_l]$ is polynomial in $\diamond_j$ whenever $\omega_{\mathtt{g}_j}(\theta^*_{\overline{v}}(\overline{u_j}))$ depends on $\overline{v}$ (Else we obtain a contradiction since $\omega_{\mathtt{f}}(\theta^*(v_1), \cdots, \theta^*(v_n))$ is polynomial in the $\theta^*(v_1), \cdots, \theta^*(v_n)$. Moreover, if $\omega_{\mathtt{g}_j}(\theta^*_{\overline{v}}(\overline{u_j}))$ does not depend on $\overline{v}$ then it is constant. By lemma 3 and by monotonicity of $Q_{\mathtt{f}}$, $Q_{\mathtt{f}}(\omega_{\mathtt{g}_j}(\theta^*_{\overline{v}}(\overline{u_j})))$ is bounded by $Q_{\mathtt{f}}(\omega_{\mathtt{f}}(\theta^*(\overline{v})))$. Finally, the ring of polynomials being closed by composition, we know that $\|\mathtt{f}(v_1, \cdots, v_n)\|$ is polynomially bounded in the $\theta^*(v_1), \cdots, \theta^*(v_n)$. Since the considered sup-interpretations are additive, we have by lemma 1 that $\theta^*(v) \leq \alpha|v|$ for some constant α. Consequently, $\|\mathtt{f}(v_1, \cdots, v_n)\|$ is also bounded by a polynomial in $|v_1|, \cdots, |v_n|$ which is independent from the inputs.

□

D Example

The following example illustrates that the quasi-friendly criterion captures, in practice, more algorithms than the friendly criterion of [16]. In fact, contrary to this latter criterion, the quasi-friendly criterion captures algorithms over trees (where the tree algebra is generated by the binary constructor symbol **c** for nodes and the unary constructor symbol **tip** for leaves).

$$\begin{aligned} \mathtt{f}(s,t) = \textbf{Case}\ s,t\ \textbf{of}\ &\mathbf{c}(x,y), \mathbf{c}(x',y') \to \mathbf{c}(\mathtt{f}(x,y), \mathtt{f}(x',y')) \\ &\mathbf{c}(x,y), \mathbf{tip}(u) \to \mathbf{tip}(u) \\ &\mathbf{tip}(u), \mathbf{c}(x,y) \to \mathbf{tip}(u) \\ &\mathbf{tip}(u), \mathbf{tip}(v) \to \mathtt{q}(u,v) \end{aligned}$$

If the leaves of s and t are the words $u_1, \cdots, u_n$ and $v_1, \cdots, v_n$, then $\mathtt{f}$ computes the tree whose leaves form the word $\mathtt{q}(u_1, v_2), \ldots, \mathtt{q}(u_n, v_n)$ with $\mathtt{q}$ the division function described in example 1. Taking $\omega_{\mathtt{f}}(X,Y) = X+Y$, $\theta(\mathbf{tip})(X) = X+1$, $\theta(\mathtt{q})(X,Y) = X$ and $\theta(\mathbf{c})(X,Y) = X+Y+1$ we can show easily that it is quasi-friendly.

$$\begin{aligned} \omega_{\mathtt{f}}(\theta^*(\mathbf{c}(x,y)), \theta^*(\mathbf{c}(x',y')) &= X+Y+X'+Y'+2 \\ &> \max(X+Y, X'+Y') \\ &= \max(\omega_{\mathtt{f}}(\theta^*(x), \theta^*(y), \omega_{\mathtt{f}}(\theta^*(x'), \theta^*(y')) && \text{(Cnd 1)} \\ \omega_{\mathtt{f}}(\theta^*(\mathbf{c}(x,y)), \theta^*(\mathbf{c}(x',y')) &= X+Y+X'+Y'+2 \\ &> X+Y+X'+Y'+1 \\ &= \theta(\mathbf{c})(\omega_{\mathtt{f}}(\theta^*(x), \theta^*(y)), \omega_{\mathtt{f}}(\theta^*(x'), \theta^*(y'))) && \text{(Cnd 2)} \end{aligned}$$

E Interpreter with cache

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

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

$$\frac{\mathcal{R}, \sigma \vdash \langle C_{i-1}, t_i \rangle \to \langle C_i, w_i \rangle \quad \mathtt{f}(p_1, \cdots, p_n) \to r \in \mathcal{R} \quad p_i\sigma' = w_i \quad \mathcal{R}, \sigma' \vdash \langle C_n, r \rangle \to \langle C, w \rangle}{\mathcal{R}, \sigma \vdash \langle C_0, \mathtt{f}(t_1, \cdots, t_n) \rangle \to \langle C\mathtt{union}(\mathtt{f}(w_1, \cdots, w_n), w), w \rangle}(Push)$$

Fig. 2. Evaluation of a rewriting system with memoization of intermediate evaluations