Resource analysis by sup-interpretation

Jean-Yves Marion, Romain Péchoux

Resource analysis by sup-interpretation

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. We propose a new method to control memory resources by static analysis. For this, we introduce the notion of sup-interpretation which bounds from above the size of function outputs. This method applies to first order functional programming with pattern matching. This work is related to quasi-interpretations but we are now able to determine resources of more algorithms and it is easier to perform an analysis with this new tools.

1 Introduction

This paper deals with general investigation on program complexity analysis. It introduces the notion of sup-interpretation, a new tool 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.

A sup-interpretation of a program is a *partial* assignment of function symbols, which ranges over reals and which bounds the size of the computed values.

The practical issue is to provide program static analysis in order to guarantee space resources that a program consumes during an execution. There is no need to say that this is crucial for at least many critical applications, and have strong impact in computer security. There are several approaches related in [18], which are trying to solve the same problem. The first protection mechanism is by monitoring computations. However, if the monitor is compiled with the program, it could crash unpredictably by memory leak. The second is the testing-based approach, which is complementary to static analysis. Indeed, testing provides a

lower bound on the memory while static analysis gives an upper bound. The gap between both bounds is of some value in practical. Lastly, the third approach is type checking done by a bytecode verifier. In an untrusded environment (like embedded systems), the type protection policy (Java or .Net) does not allow dynamic allocation. Our approach is an attempt to control resources, and provide a proof certificate, of a high-level language in such a way that the compiled code is safe wrt memory overflow. Thus, we capture and deal with memory allocation features. Similar approaches are the one of Hofmann [12, 13] and the one of Aspinall and Compagnoni [5].

For that purpose we consider first order functional programming language with pattern matching but we hardly believe that such a method could be applied to other languages such as resource bytecode verifier by following the lines of [2], language with synchronous cooperative threads as in [3] or first order functional language including streams as in [11] .

The notion of sup-interpretation can be seen as a kind of annotation provided in the code by the programmer. Sup-interpretations strongly inherit from the notion of quasi-interpretation developed by Bonfante, Marion and Moyen in [8, 9, 17]. Consequently the notion of sup-interpretation is heiress of the notion of polynomial interpretation used to prove termination of programs in [10, 14] and more recently in [6, 16]. Quasi-interpretation, like sup-interpretation, provides a bound over function outputs by static analysis for first order functional programs. Quasi-interpretation was 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.

However both notions differ for two reasons. First, the sup-interpretations are partial assignments which does not satisfy the subterm property, and this allows to capture a larger class of algorithms. In fact, programs computing logarithm or division admits a sup-interpretation but have no quasi-interpretation. Second, the sup-interpretation is a partial assignment over the set of function symbols of a program, whereas the quasi-interpretation is a total assignment on function symbols. On the other hand, sup-interpretations come with a companion, which is a weight to measure argument size of recursive calls involved in a program run. Such constraints are developped thanks to the notion of dependency pairs

by Arts and Giesl [4]. The dependency pairs were initially introduced for proving termination of term rewriting systems automatically. Even if this paper no longer focuses on termination, the notion of dependency pair is used for forcing the program to compute in polynomial space. There is a very strong relation between termination and complexity. Indeed,both for proving some complexity bounds and for proving termination, we need to control the arguments occurring in a function recursive call. Since we try to control the arguments of a recursive call together, the sup-interpretation is closer from the dependency pairs method than from the size-change principle method of [15] which consider the arguments of a recursive call separately. Section 2 introduces the first order functional language and its semantics. Section 3 defines the main notions of sup-interpretation and weight using to bound the size of a program outputs. Section 4 presents the notion of dependency pairs by Arts and Giesl [4] and the ensuing notion of fraternity used to control the size of values added by recursive calls. In section 5, we define the notion of polynomial and additive assignments for sup-interpretations and weights. Finally, section 6 introduces the notion of friendly programs and the main theorems of this paper providing a polynomial bound on the values computed by friendly programs. The full paper with all proofs is available at `http://www.loria.fr/~pechoux`.

2 First order functional programming

2.1 Syntax of programs

We define a generic first order functional programming language. The vocabulary $\Sigma = \langle Cns, Op, Fct \rangle$ is composed of three disjoint domains of symbols. The arity of a symbol is the number n of arguments that it takes. The set of programs are defined by the following grammar.

$$
\begin{array}{lll}
\texttt{Programs} \ni \mathbf{p} & ::= & def_1, \cdots, def_m \\
\texttt{Definitions} \ni def & ::= & \mathtt{f}(x_1, \cdots, x_n) = e^{\mathtt{f}} \\
\texttt{Expression} \ni e & ::= & x \mid \mathbf{c}(e_1, \cdots, e_n) \mid \mathbf{op}(e_1, \cdots, e_n) \mid \mathtt{f}(e_1, \cdots, e_n) \\
 & & \mid \textbf{Case } e_1, \cdots, e_n \textbf{ of } \overline{p_1} \to e^1 \ldots \overline{p_\ell} \to e^\ell \\
\texttt{Patterns} \ni p & ::= & x \mid \mathbf{c}(p_1, \cdots, p_n)
\end{array}
$$

where $\mathbf{c} \in \mathit{Cns}$ is a constructor, $\mathbf{op} \in \mathit{Op}$ is an operator, $\mathtt{f} \in \mathit{Fct}$ is a function symbol, and $\overline{p_i}$ is a sequence of n patterns. Throughout, we generalize this notation to expressions and we write $\overline{e}$ to express a sequence of expressions, that is $\overline{e} = e_1, \ldots, e_n$, for some n clearly determined by the context.

The set of variables Var is disjoint from Σ and $x \in \mathit{Var}$. In a definition, $e^{\mathtt{f}}$ is called the body of $\mathtt{f}$. 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. The program's main function symbol is the first function symbol in the program's list of definitions. We usually don't make the distinction between this main symbol and the program symbol $\mathbf{p}$.

Lastly, it is convenient, because it avoids tedious details, to restrict case definitions in such way that an expression involved in a **Case** expression does not contain nested **Case** (In other word, an expression e^j does not contain an expression **Case**). 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.

2.2 Semantics

The set Values is the constructor algebra freely generated from Cns.

$$\mathit{Values} \ni v ::= \mathbf{c} \mid \mathbf{c}(v_1, \cdots, v_n) \qquad\qquad \mathbf{c} \in \mathit{Cns}$$

Put $\mathit{Values}^* = \mathit{Values} \cup \{\mathbf{Err}\}$ where **Err** is the value associated when an error occurs. Each operator **op** of arity n is interpreted by a function $[\![\mathbf{op}]\!]$ from Values^n to Values^*. Operators are essentially basic partial functions like destructors or characteristic functions of predicates like $=$. The destructor **tl** illustrates the purpose of **Err** when it satisfies $[\![\mathbf{tl}(\mathbf{nil})]\!] = \mathbf{Err}$.

The computational domain is $\mathit{Values}^{\#} = \mathit{Values} \cup \{\mathbf{Err}, \bot\}$ where $\bot$ means that a program is non-terminating. The language has a closure-based call-by-value semantics which is displayed in Figure 1. A few comments are necessary. A substitution σ is a finite function from variables to Values. The application of a substitution σ to an expression e is noted $e\sigma$.

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

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

$$\frac{e \downarrow u \quad \exists \sigma,\ i\ :\ p_i\sigma = u \quad e_i\sigma \downarrow w}{\mathbf{Case}\ x\ \mathbf{of}\ p_1 \rightarrow x_1 \ldots p_\ell \rightarrow x_\ell \downarrow w}\ \mathbf{Case} \text{ and } u \neq \mathbf{Err}$$

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

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

The meaning of $e \downarrow w$ is that e evaluates to the value w of *Values*. If no rule is applicable, then an error occurs, and $e \downarrow \mathbf{Err}$. So, a program $\mathbf{p}$ computes a partial function $[\![\mathbf{p}]\!] : \mathit{Values}^n \rightarrow \mathit{Values}^{\#}$ defined as follows. For all $v_i \in \mathit{Values}$, $[\![\mathbf{p}]\!](v_1, \cdots, v_n) = w$ iff $\mathbf{p}(v_1, \cdots, v_n) \downarrow w$. Otherwise $[\![\mathbf{p}]\!](v_1, \cdots, v_n) = \perp$. Throughout, we shall say that $[\![\mathbf{p}]\!](v_1, \cdots, v_n)$ is defined when $[\![\mathbf{p}]\!](v_1, \cdots, v_n)$ is a constructor term of *Values*.

3 Sup-interpretations

3.1 Partial assignment

A partial assignment θ is a partial mapping from a vocabulary Σ such that for each symbol $\mathtt{f}$ of arity n, in the domain of θ, it yields a partial function $\theta(f) : (\mathbb{R})^n \longmapsto \mathbb{R}$. The domain of a partial assignment θ is noted $\mathrm{dom}(\theta)$. Because it is convenient, we shall always assume that partial assignments that we consider, are defined on constructors and operators. That is $\mathit{Cns} \cup \mathit{Op} \subseteq \mathrm{dom}(\theta)$.

An expression e is defined over $\mathrm{dom}(\theta)$ if each symbol belongs to $\mathrm{dom}(\theta)$ or is a variable of *Var*. Take a denumerable sequence $X_1, \ldots, X_n, \ldots$. Assume that an expression e is defined over $\mathrm{dom}(\theta)$ and has n variables. The partial assignment of e wrt θ is the extension of the assignment θ to the expression e that we write $\theta^*(e)$. It denotes a function from $\mathbb{R}^n$ to $\mathbb{R}$ and is defined as follows:

1. If x_i is a variable of *Var*, let $\theta^*(x_i) = X_i$
2. If b is a 0-ary symbol of Σ, then $\theta^*(b) = \theta(b)$.
3. If $\overline{e}$ is a sequence of n expressions, then $\theta^*(\overline{e}) = \max(\theta^*(e_1), \ldots, \theta^*(e_n))$
4. If e is a **Case** expression of the shape **Case** $\overline{e}$ **of** $\overline{p_1} \to e^1 \ldots \overline{p_\ell} \to e^\ell$,

$$\theta^*(e) = \max(\theta^*(\overline{e}), \theta^*(e^1), \ldots, \theta^*(e^\ell))$$

5. If f is a symbol of arity $n > 0$ and $e_1, \cdots, e_n$ are expressions, then

$$\theta^*(f(e_1, \cdots, e_n)) = \theta(f)(\theta^*(e_1), \ldots, \theta^*(e_n))$$

3.2 Sup-interpretation

Definition 1 (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 $f \in dom(\theta)$, the function $\theta(f)$ satisfies*

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

2. *For each $v \in Values$,*

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

   The size of an expression e is noted $|e|$ and is defined by $|\mathbf{c}| = 0$ where $\mathbf{c}$ is a 0-ary symbol and $|\mathbf{b}(e_1, \ldots, e_n)| = 1 + \sum_i |e_i|$ where $\mathbf{b}$ is a n-ary symbol.
3. *For each symbol $f \in dom(\theta)$ of arity n and for each value $v_1, \ldots, v_n$ of Values, if $[\![f]\!](v_1, \ldots, v_n)$ is defined, that is $[\![f]\!](v_1, \ldots, v_n) \in Values$, then*

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

Now an expression e admits a sup-interpretation θ if e is defined over $\mathrm{dom}(\theta)$. 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 approximation from above of the size of the outputs of the function denoted by the program.

Lemma 1. *Let e be an expression with no variable and which admits a sup-interpretation θ. Assume that $[\![e]\!]$ is defined. We then have*

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

Proof. The proof is done by structural induction on expression. The base case is a consequence of Condition 2 of Definition 1.

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

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

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

$$\|e\| = \begin{cases} |[\![e]\!]| & \text{if } [\![e]\!] \text{ is defined} \\ 0 & \text{otherwise} \end{cases}$$

Corollary 1. *Let e be an expression with no variable and which admits a sup-interpretation θ. Assume that $[\![e]\!]$ is defined. We then have*

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

Proof.

$$\begin{aligned}
\theta^*(e) &\geq \theta^*([\![e]\!]) && \text{by Lemma 1} \\
&\geq \|e\| && \text{by Condition 2 of Dfn 1}
\end{aligned}$$

Example 1.

$$\begin{aligned}
\texttt{half}(x) \ = \mathbf{Case}\ x\ \mathbf{of}\ \mathbf{0} &\to \mathbf{0} \\
\mathbf{S}(\mathbf{0}) &\to \mathbf{0} \\
\mathbf{S}(\mathbf{S}(y)) &\to \mathbf{S}(\texttt{half}(y))
\end{aligned}$$

In this example, the function `half` computes $\lfloor n/2 \rfloor$ on an entry of size n. So by taking $\theta(\mathbf{S})(X) = X+1$ and $\theta(\texttt{half})(X) = X/2$, we define a sup-interpretation of the function symbol `half`. In fact, those functions are monotonic. For every unary value v of size n, $\theta^*(v) = n \geq n = |v|$ by definition of $\theta(\mathbf{S})$, so that condition 2 on sup-interpretation is satisfied. Finally, it remains to check that for every value v, $\theta^*(\texttt{half}(v)) \geq \theta^*([\![\texttt{half}(v)]\!])$. For a value v of size n, we have by definition of θ^* that $\theta^*(\texttt{half}(v)) = \theta^*(v)/2 = n/2$ and $\theta^*([\![\texttt{half}(v)]\!]) = \|\texttt{half}(v)\| = \lfloor n/2 \rfloor$. Since $n/2 \geq \lfloor n/2 \rfloor$, condition 3 of sup-interpretation is satisfied. Notice that such a sup-interpretation is not a quasi-interpretation (a fortiori not an interpretation for proof termination) since it has not the subterm property.

3.3 Weight

The weight 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. Intuitively, whereas the sup-interpretation controls the size of the computed values, the weight can be seen as a control point for the computation of recursive calls.

Definition 2 (Weight). *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*

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

The weight of a function is often taken to be the maximum or the sum functions.

The weight is useful to control the number of occurrences of a recursive call . Indeed, the monotonicity property combined with the fact that a weight ranges over function symbols ensures suitable properties on the number of occurrences

of a loop in a program when we consider the constraints given in section 6. Moreover, the subterm property ensures to control the size of each argument in a recursive calls, in opposition to the size-change principle as mentioned in the introduction.

4 Fraternities

In this section we define fraternies which are an important notion based on dependency pairs, that Arts and Giesl [4] introduced to prove termination automatically. Fraternities allows 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 $\diamond_i$. Here, we suppose that the $\diamond_i$'s are new symbols which are not in Σ nor in *Var*. The substitution of each $\diamond_i$ by an expression d_i is noted $\mathsf{C}[d_1, \cdots, d_r]$.

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, $e^{\mathtt{f}} = \mathbf{Case}\ x_1, \cdots, x_n\ \mathbf{of}\ \overline{q_1} \to e_1 \ldots \overline{q_\ell} \to e_\ell$, 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 $\overline{q_j} = q_{j,1} \ldots q_{j,n}$.

At first glance, this definition may look a bit tedious. However, it is practical 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.

Definition 3. *Assume that* $\boldsymbol{p}$ *is a program. A dependency pair*

$$\langle \mathit{f}(p_1, \cdots, p_n), \mathit{g}(t_1, \cdots, t_m) \rangle$$

is a couple such that $\mathit{g}(t_1, \cdots, t_m)$ *is activated by* $\mathit{f}(p_1, \cdots, p_n)$.

The dependency pairs provide a precedence $\geq_{Fct}$ on function symbols. Indeed, set $\mathtt{f} \geq_{Fct} \mathtt{g}$ if $\langle \mathtt{f}(\overline{p}), \mathtt{g}(\overline{t}) \rangle$ is a dependency pair. 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 partial order. 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.

Say that an expression d activated by $\mathtt{f}(p_1, \cdots, p_n)$ is maximal if there is no context $\mathsf{C}[\diamond]$ such that $\mathsf{C}[d]$ is activated by $\mathtt{f}(p_1, \cdots, p_n)$.

Definition 4. *In a program $\boldsymbol{p}$, a maximal expression $d = \mathsf{C}[\boldsymbol{g}_1(\overline{t_1}), \ldots, \boldsymbol{g}_r(\overline{t_r})]$ activated by $\boldsymbol{f}(p_1, \cdots, p_n)$ is a fraternity if*

1. *For each $i \in \{1, r\}$, $\boldsymbol{g}_i \approx_{Fct} \boldsymbol{f}$.*
2. *For every function symbol $\mathtt{h}$ that appears in the context $\mathsf{C}[\diamond_1, \cdots, \diamond_r]$, we have $\boldsymbol{f} >_{Fct} \mathtt{h}$.*

All along, we suppose that there is no nested fraternities, which means that a fraternity d does not contain any fraternity inside it. This restriction prevents definitions of the shape $\mathtt{f}(\mathbf{S}(x)) = \mathtt{f}(\mathtt{f}(x))$. This restriction is not too strong since such functions 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. The following examples illustrate typical fraternity constructions.

Example 2. Consider the program $\mathtt{log}$ computing $\mathtt{log}_2(n) + 1$ on an entry of size n and using the program $\mathtt{half}$ of example 1.

$$\begin{aligned}
\mathtt{log}(x) &= \mathbf{Case}\ x\ \mathbf{of}\ \mathbf{0} \to \mathbf{0} \\
&\qquad\qquad \mathbf{S}(y) \to \mathbf{S}(\mathtt{log}(\mathtt{half}(\mathbf{S}(y)))) \\
\mathtt{half}(x) &= \mathbf{Case}\ x\ \mathbf{of}\ \mathbf{0} \to \mathbf{0} \\
&\qquad\qquad \mathbf{S}(\mathbf{0}) \to \mathbf{0} \\
&\qquad\qquad \mathbf{S}(\mathbf{S}(y)) \to \mathbf{S}(\mathtt{half}(y))
\end{aligned}$$

This program admits two fraternities $\mathbf{S}(\mathtt{log}[\mathtt{half}(\mathbf{S}(y))])$ and $\mathbf{S}[\mathtt{half}(y)]$ since $\mathtt{log} >_{Fct} \mathtt{half}$.

Example 3 (division). Consider the following definitions that encode the division:

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

This program admits two fraternities $\mathtt{minus}(u,v)$ and $\mathbf{S}[\mathtt{q}(\mathtt{minus}(z,u),\mathbf{S}(u))]$ since $\mathtt{q} >_{Fct} \mathtt{minus}$.

Definition 5. *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 $\mathsf{C}[\mathtt{g}(t_1, \cdots, t_k)]$ is activated by $\mathtt{f}(p_1, \cdots, p_n)$. A* transition *is a triplet $\eta_1 \overset{\mathsf{C}[\diamond]}{\rightsquigarrow} \eta_2$ such that:*

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

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. $\overset{+}{\rightsquigarrow}$ is the transitive closure of $\overset{\mathsf{C}[\diamond]}{\rightsquigarrow}$. Moreover we say that a dependency pair is involved in the call-tree, if its evaluation for a given substitution corresponds to $< \eta_1, \eta_2 >$.

5 Polynomial assignments

Definition 6. *A partial assignment θ is polynomial if for each symbol f of arity n of $dom(\theta)$, $\theta(f)$ is bounded by a polynomial of $\mathbb{R}[X_1, \cdots, X_n]$. A polynomial sup-interpretation is a polynomial assignment. A polynomial weight ω of arity n is a weight which is bounded by some polynomial of $\mathbb{R}[X_1, \cdots, X_n]$.*

An assignment of $\mathbf{c} \in \mathrm{dom}(\theta)$ is *additive* (or of kind 0) if

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

We classify *polynomial* assignments by the kind of assignments given to constructors, and not to function symbols. If each constructor of a polynomial assignment is additive then the assignment is additive. Throughout the following paper we consider additive assignments.

Lemma 2. *There is a constant α such that for each value v of Values, the inequality is satisfied :*

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

6 Local criteria to control space resources

Definition 7 (Friendly). *A program $\boldsymbol{p}$ is* friendly *iff there is a polynomial sup-interpretation θ and a polynomial weight ω such that for each fraternity expression $d = \mathsf{C}[\boldsymbol{g}_1(\overline{t_1}), \ldots, \boldsymbol{g}_r(\overline{t_r})]$ activated by $\boldsymbol{f}(p_1, \cdots, p_n)$ we have,*

$$\theta^*(\mathsf{C}[\diamond_1, \ldots, \diamond_r]) = \max_{i=1..r}(\diamond_i + R_i(Y_1, \ldots, Y_m))$$

where each Y_i corresponds to a variable occurring in C.

Moreover, for each $i \in \{1, r\}$, we have that for each substitution σ,

$$\omega_f(\theta^*(p_1\sigma), \ldots, \theta^*(p_n\sigma)) \geq \omega_{g_i}(\theta^*(t_{i,1}\sigma), \ldots, \theta^*(t_{i,m}\sigma))$$

Moreover, if

$$\exists\sigma \quad \omega_f(\theta^*(p_1\sigma), \ldots, \theta^*(p_n\sigma)) = \omega_{g_i}(\theta^*(t_{i,1}\sigma), \ldots, \theta^*(t_{i,m}\sigma))$$

Then $R_i(Y_1, \ldots, Y_m)$ is the null polynomial.

Example 4. The program of example 2 is friendly. We take $\theta(\mathbf{S})(X) = X + 1$ and $\theta(\mathtt{half})(X) = X/2$. The context of the two fraternities involved in this program

are of the shape $\mathbf{S}[\diamond]$, thus having a sup-interpretation of the shape $\diamond + 1$. We have to find $\omega_{\mathtt{log}}$ and $\omega_{\mathtt{half}}$ such that for every σ:

$$\omega_{\mathtt{log}}(\theta^*(\mathbf{S}(y)\sigma)) > \omega_{\mathtt{log}}(\theta^*(\mathtt{half}(\mathbf{S}(y)\sigma)))$$
$$\omega_{\mathtt{half}}(\theta^*(\mathbf{S}(\mathbf{S}(y\sigma)))) > \omega_{\mathtt{half}}(\theta^*(y\sigma))$$

Both inequalities are satisfied by taking $\omega_{\mathtt{log}}(X) = \omega_{\mathtt{half}}(X) = X$. Thus the program is friendly.

Example 5. The program of example 3 is friendly by taking $\theta(\mathbf{S})(X) = X + 1$, $\theta(\mathtt{minus})(X, Y) = X$, $\omega_{\mathtt{minus}}(X, Y) = \max(X, Y)$ and $\omega_{\mathtt{q}}(X, Y) = X + Y$

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

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

Proof. The proof can be found in appendix A.2 relying on definitions and lemmas of appendix A.1. It begins by assigning a polynomial $P_{\mathtt{f}}$ to every function symbol $\mathtt{f}$ of a friendly programs. This polynomial is the sum of a bound on the size of values added by the contexts of recursive calls and of a bound on the size of values added by the calls which are no longer recursive. Then it checks both bounds thus showing that the values computed by the program are polynomially bounded.

The programs presented in examples 2 and 3 are examples of friendly program and thus computing polynomially bounded values. More examples of friendly programs can be found in the appendix.

The next result strengthens Theorem above. Indeed it claims that even if a program is not terminating then the intermediate values are polynomially bounded. This is quite interesting because non-terminating process are common, and moreover it is not difficult to introduce streams with a slight modification of the above Theorem, which is essentially based on the semantics change.

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

call-tree of root $\langle \mathbf{f}, v_1, \cdots, v_n \rangle$,

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

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

Proof. The proof, located in appendix A.3, is a consequence of previous theorem since in every state of the call-tree, the values are computed and thus bounded polynomially. The proof is in the full paper.

Remark 1. As mentionned above, this theorem holds for non-terminating programs and particularly for a class of programs including streams. For that purpose we have to give a new definition of substitutions over streams. In fact, it would be meaningless to consider a substitution over stream variables. Thus stream variables are never subsituted and the sup-interprtation of a stream l is taken to be a new variable L like in definition of the sup-interpretations.

7 Conclusion and Perspectives

The notion of sup-interpretation allows to check that the size of the outputs of a friendly program is bounded polynomially by the size of it inputs. It allows to capture algorithms admitting no quasi-interpretations (division, logarithm, gcd ...). So, our experiments show that is not too difficult to find sup-interpretations for the following reasons. First, we have to guess sup-interpretation and weight of only some, and not all, symbols. Second, a quasi-interpretations for those symbol works pretty well in most of the cases. And so we can use tools to synthesize quasi-interpretations [1, 7].

Sup-interpretation should be easier to synthesize than quasi-interpretations since we have to find fewer functions. Moreover it is not so hard to find a sup-interpretation, since quasi-interpretation often defines a sup-interpretation (except in the case of additive contexts). In the appendix C, we give some results on termination for a subclass of friendly programs. This result strongly inherits in a natural way from the dependency pair method of Arts and Giesl [4]. However, it differs in the sense that the monotonicity of the quasi-ordering and

the inequalities over definitions (rules) of a program are replaced by the notion of sup-interpretation combined to weights. Consequently, it shares the same advantages and disadvantages than the dependency pairs method compared to termination methods such as size-change principle by Jones et al. [15], failing on programs admitting no polynomial orderings (Ackermann function, for example), and managing to prove termination on programs where the size-change principle fails. For a more detailed comparison between both termination criteria see [19]. Finally, an open question concerns characterization of time complexity classes with the use of such a tool, particularly, the characterization of polynomial time by determining a restriction on sup-interpretations.

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. G. Bonfante, A. Cichon, J.-Y. Marion, and H. Touzet. Algorithms with polynomial interpretation termination proof. *Journal of Functional Programming*, 11, 2000.
7. G. Bonfante, J.-Y. Moyen J.-Y. Marion, and R. Péchoux. Synthesis of quasi-interpretations, 2005. `http://www.loria/~pechoux`.
8. G. Bonfante, J.-Y. Marion, and J.-Y. Moyen. On lexicographic termination ordering with space bound certifications. In Dines Bjørner, Manfred Broy, and Alexandre V. Zamulin, editors, *Perspectives of System Informatics, 4th International Andrei Ershov Memorial Conference, PSI 2001, Akademgorodok, Novosibirsk, Russia,*

Ershov Memorial Conference, volume 2244 of *Lecture Notes in Computer Science*. Springer, Jul 2001.

9. 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/~moyen/appsemTCS.ps`.
10. N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, pages 69–115, 1987.
11. S.G. Frankau and A. Mycroft. Stream processing hardware from functional language specifications. In Martin Hofmann, editor, *36th Hawai'i International Conference on System Sciences (HICSS 36)*. IEEE, 2003.
12. 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.
13. 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.
14. D.S. Lankford. On proving term rewriting systems are noetherien. Technical report, 1979.
15. 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.
16. S. Lucas. Polynomials over the reals in proofs of termination: from theory to practice. *RAIRO Theoretical Informatics and Applications*, 39(3):547–586, 2005.
17. 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.
18. J. Regehr. Say no to stack overflow. 2004. `http://www.embedded.com`.
19. R. Thiemann and J. Giesl. Size-change termination for term rewriting. In *14th International Conference on Rewriting Techniques and Applications*, Lecture Notes in Computer Science, Valencia, Spain, 2003. Springer.

A Proofs

A.1 Properties of friendly programs

A friendly program $\mathbf{p}$ has two characteristic factors. The first one $R_{\mathbf{p}}$ is called the *thickness* function. It is defined as the maximum of the R_i's residual functions which appear in the sup-interpretation of a fraternity. Formally, for each fraternity $d = \mathsf{C}[\mathsf{g}_1(\overline{t_1}), \ldots, \mathsf{g}_r(\overline{t_r})]$ activated by some expression, we set $R_d(X) = \max_{i=1..r}(R_i(X, \ldots, X))$ where

$$\theta^*(\mathsf{C}[\diamond_1, \ldots, \diamond_r]) = \max_{i=1..r}(\diamond_i + R_i(Y_1, \ldots, Y_m))$$

Then, we put $R_{\mathbf{p}} = \max_{\forall \text{ fraternity } d}\{R_d(X)\}$

The second one δ is called the *erosion* factor and intuitively it corresponds to the argument quantity which is removed at each recursive step.

In order to define to define the erosion factor δ, we consider a dependency pair $u = \langle \mathsf{g}(q_1, \cdots, q_m), \mathtt{h}(s_1, \cdots, s_k)\rangle$ and we set the erosion factor δ_u implied by u thus. If the friendly criteria is strict, that is

$$\omega_{\mathsf{g}}(\theta^*(q_1\sigma), \ldots, \theta^*(q_m\sigma)) > \omega_{\mathtt{h}}(\theta^*(s_1\sigma), \ldots, \theta^*(s_k\sigma))$$

So the constant $\delta_u > 0$ verifies

$$\omega_{\mathsf{g}}(\theta^*(q_1\sigma), \ldots, \theta^*(q_m\sigma)) \geq 1/\delta_u + \omega_{\mathtt{h}}(\theta^*(s_1\sigma), \ldots \theta^*(s_k\sigma))$$

Otherwise,

$$\omega_{\mathsf{g}}(\theta^*(q_1\sigma), \ldots, \theta^*(q_m\sigma)) \geq \omega_{\mathtt{h}}(\theta^*(s_1\sigma), \ldots, \theta^*(s_k\sigma))$$

and we put $\delta_u = 0$. Next, we set $\delta = \max_{\forall \text{ dep. pair } u}\{\delta_u\}$.

Definition 8 (Lightening). *Given a weight and a sup-interpretation θ, a dependency pair u is called a lightening if $\delta_u > 0$*

Definition 9 (cycle). *Suppose that we have a call-tree, for a given function symbol f, we call a cycle in the evaluation of f, a branch of the call tree of the shape $\langle \mathsf{f}, u_1, \cdots, u_n\rangle \overset{+}{\rightsquigarrow} \langle \mathsf{f}, u'_1, \cdots, u'_n\rangle$ where for every state $\langle \mathsf{g}, v_1, \cdots, v_k\rangle$ of this branch (if there is one) disctinct from $\langle \mathsf{f}, u_1, \cdots, u_n\rangle$ and $\langle \mathsf{f}, u'_1, \cdots, u'_n\rangle$, g is disctinct from f. The number of occurrences of a cycle determines the number of successive cycles in on branch of the call-tree.*

Lemma 3. *For a friendly program and a branch* $\langle \mathtt{f}, u_1, \cdots, u_n\rangle \overset{+}{\rightsquigarrow} \langle \mathtt{g}, v_1, \cdots, v_k\rangle$ *of its call-tree with* $\mathtt{f} \approx_{Fct} \mathtt{g}$ *, then*

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

Proof. We start by induction on the length k of branch were the length represents the number of transitions. If $k = 1$, we have one dependency pair $u = \langle \mathtt{f}(p_1, \cdots, p_m), \mathtt{g}(t_1, \cdots, t_m)\rangle$ involved in the branch of the call-tree. Since our program is friendly, we know that:

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

Suppose that our induction hypothesis is satisfied for a branch of length k and consider a branch of length $k+1$. That is $\langle \mathtt{f}, u_1, \cdots, u_n\rangle \overset{k}{\rightsquigarrow} \langle \mathtt{h}, v'_1, \cdots, v'_l\rangle \overset{1}{\rightsquigarrow} \langle \mathtt{g}, v_1, \cdots, v_k\rangle$. We know, by definition of $\approx_{Fct}$, that $\mathtt{f} \approx_{Fct} \mathtt{h}$. Thus by induction hypothesis,

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

Consider the dependency pair $u = \langle \mathtt{h}(p_1, \cdots, p_l), \mathtt{g}(t_1, \cdots, t_m)\rangle$ involved in the transition $\langle \mathtt{h}, v'_1, \cdots, v'_l\rangle \overset{1}{\rightsquigarrow} \langle \mathtt{g}, v_1, \cdots, v_k\rangle$. By the friendly criteria we know that for each substitution σ,

$$\omega_{\mathtt{h}}(\theta^*(p_1\sigma), \cdots, \theta^*(p_l\sigma)) \geq \omega_{\mathtt{g}}(\theta^*(t_1\sigma), \cdots, \theta^*(t_m\sigma))$$

By lemma 1, we know that $\theta^*(t_i\sigma) \geq \theta^*(\llbracket t_i\sigma \rrbracket),\ \forall i = 1..m$. Thus by monotonicity of the weight, we obtain that:

$$\omega_{\mathtt{g}}(\theta^*(t_1\sigma), \cdots, \theta^*(t_m\sigma)) \geq \omega_{\mathtt{g}}(\theta^*(\llbracket t_1\sigma \rrbracket), \ldots, \theta^*(\llbracket t_m\sigma \rrbracket))$$

Taking σ such that $p_j\sigma = v'_j,\ \forall j = 1..l$, we know by definition of a transition in the call-tree that $\llbracket t_i\sigma \rrbracket = v_i,\ \forall i = 1..m$. Combining the previous inequalities, we obtain that:

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

Lemma 4. *Assume that* $\boldsymbol{p}$ *is a friendly program. For every node* $\langle \mathtt{g}, v_1, \cdots, v_k \rangle$ *of the call-tree of root* $\langle \mathtt{f}, u_1, \cdots, u_n \rangle$*, if* $\mathtt{g} \approx_{Fct} \mathtt{f}$ *then*

$$|v_j| \leq \omega_f(\theta^*(u_1), \cdots, \theta^*(u_n))$$

Proof. Since our program is friendly and $\mathtt{g} \approx_{Fct} \mathtt{f}$ and by lemma 3,

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

By subterm property of weight and definition of sup-interpretations,

$$\omega_{\mathtt{g}}(\theta^*(v_1), \cdots, \theta^*(v_k)) \geq \theta^*(v_j) \geq |v_j|, \ \forall j = 1..k$$

A.2 Proof of theorem 1

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

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

A.2.1 Assignments of function symbols

All along, we consider a friendly program $\mathbf{p}$ which admits a sup-interpretation. We now provide an assignment $P_{\mathtt{f}}$ to each function symbol $\mathtt{f}$ of a program $\mathbf{p}$, which is a function bounded by a polynomial and which satisfies that for every value $v_1, \ldots, v_n$ of *Values*,

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

The assignation is built recursively following the function symbol precedence $\geq_{Fct}$ and on the definition size.

1. Suppose that the definition $\mathtt{f}(x_1, \cdots, x_n) = e^{\mathtt{f}}$ contains only constructors, operators or symbols of rank strictly lower that the rank of $\mathtt{f}$ wrt $\geq_{Fct}$. We put $P_{\mathtt{f}}(X) = \theta^*(e^{\mathtt{f}})[X_1 \leftarrow X, \ldots, X_n \leftarrow X]$. We write $\theta^*(e^{\mathtt{f}})[X_1 \leftarrow X, \ldots, X_n \leftarrow X]$ to mean that we replace $X_1, \ldots, X_n$ by the variable X.

2. Suppose that the definition $\mathtt{f}(x_1, \cdots, x_n) = e^{\mathtt{f}}$ is a case expression of the form :

$$\begin{array}{l} \textbf{Case } x_1, \ldots, x_n \textbf{ of} \\ \qquad\qquad \overline{p_1} \ \rightarrow e_1 \\ \qquad\qquad \ldots \\ \qquad\qquad \overline{p_\ell} \ \rightarrow e_\ell \end{array}$$

Say that I is the set of all indices such that e_i does not contain a fraternity for each $i \in I$. In other words, for $i \in I$, e_i just contains function symbols whose precedences are strictly less than $\mathtt{f}$. So, there is already a polynomially bounded function Q_{e_i} associated to e_i.

$$Q_{\mathtt{f}}(X) = \max_{j \in I}(Q_{e_j}[Y_1 \leftarrow X, \ldots, Y_m \leftarrow X])$$

where Y_i's are the variables of e_j.
Lastly, we assign to $\mathtt{f}$ the function

$$P_{\mathtt{f}}(X) = \delta.R_{\mathbf{p}}(\omega_{\mathtt{f}}(X, \ldots, X)).\omega_{\mathtt{f}}(X, \ldots, X) + \max_{\mathtt{f} \approx_{Fct} \mathtt{g}} Q_{\mathtt{g}}(\omega_{\mathtt{f}}(X, \ldots, X))$$

A.2.2 Upper bound

Theorem 3. *Assume that* $\boldsymbol{p}$ *is a friendly program. For each function symbol* $\boldsymbol{f}$ *of* $\boldsymbol{p}$ *there is a polynomial* P_f *such that for every value* $v_1, \ldots, v_n$ *of Values,*

$$\|f(v_1, \ldots, v_n)\| \leq P_f(\max(\theta^*(v_1), ..., \theta^*(v_n)))$$

375 Now, we establish the upper bound 3 again by induction on function symbols wrt the precedence $\geq_{Fct}$. Consider the definition $\mathtt{f}(x_1, \cdots, x_n) = e^{\mathtt{f}}$.

– Suppose that $\mathtt{f}$ is defined by composition of constructors, operators and symbols of rank strictly less than $\mathtt{f}$ wrt $\geq_{Fct}$.We have

$$\begin{array}{lll} \|\mathtt{f}(v_1, \ldots, v_n)\| = \|e^{\mathtt{f}}\sigma\| & & \text{where } x_i\sigma = v_i \\ \qquad\qquad \leq \theta^*(e^{\mathtt{f}}\sigma) & & \text{By Corollary 1} \\ \qquad\qquad = P_{\mathtt{f}}(\max(\theta^*(v_1), \ldots, \theta^*(v_n))) & & \text{by dfn 1} \end{array}$$

– Suppose that $e^{\mathtt{f}}$ is a case expression of the form.

$$\begin{array}{l}\textbf{Case } x_1, \ldots, x_n \textbf{ of} \\ \qquad\qquad \overline{p_1} \ \rightarrow e_1 \\ \qquad\qquad \ldots \\ \qquad\qquad \overline{p_\ell} \ \rightarrow e_\ell\end{array}$$

Suppose that the computation $\mathtt{f}(v_1, \ldots, v_n)$ matches the pattern sequence $\overline{p_i}$ wrt σ, that is $\mathbf{p}_i\sigma = v_i$. So e_i is triggered.

There are two cases to consider. The main one is when e_i is the fraternity $\mathsf{C}[\mathsf{g}_1(\overline{t_1}), \ldots, \mathsf{g}_r(\overline{t_r})]$. In order to bound the size of the computed value, we have to bound the number of constructor symbols added by the context in the corresponding cycle of the call-tree, noted $E_{\approx_{Fct}}$ and the number of constructor symbols added by the rules that involves function symbols strictly smaller for the precedence (i.e. that leave the cycle), noted $E_{>_{Fct}}$.

We start to bound $E_{\approx_{Fct}}$:

By the friendly criteria, the only contexts which can add constructors are the ones corresponding to lightenings. If $|\mathbf{p}|$ is the size of the program, we have at most $|\mathbf{p}|$ such lightenings in one cycle (In fact, the size of the program bounds the number of dependency pairs). But in this case, by corollary 2 there are at most $\delta\omega(\theta^*(p_1\sigma), \ldots, \theta^*(p_n\sigma))/|\mathbf{p}|$ occurrences of the cycle. So, in all cases, there are at most $\delta\omega(\theta^*(p_1\sigma), \ldots, \theta^*(p_n\sigma))$ contexts adding constructors. By lemma 4, the size of every argument of a function symbol involved in the cycle is bounded by $\omega_{\mathtt{f}}(\theta^*(p_1\sigma), \cdots, \theta^*(p_n\sigma))$. It implies by the friendly criteria that each occurrence of the cycle adds at most $R_{\mathbf{p}}(\omega_{\mathtt{f}}(\ldots, \max_{i=1..n}(\theta^*(p_i\sigma)), \ldots))$. Finally we have:

$$E_{\approx_{Fct}} \leq R_{\mathbf{p}}(\omega_{\mathtt{f}}(\ldots, \max_{i=1..n}(\theta^*(p_i\sigma)), \ldots)).\delta_u\omega_{\mathtt{f}}(\ldots, \max_{i=1..n}(\theta^*(p_i\sigma)), \ldots)$$

constructor symbols. It remains to bound the number of constructors added by function calls that leave the cycle. Since the calls leave the cycle, we know that they involve only function strictly smaller for the precedence. For such a function $\mathtt{g}$, a polynomial bound on the computed value is already given by the polynomial $Q_{\mathtt{g}}$. We know that the arguments of such a call are bounded by $\omega_{\mathtt{f}}(\theta^*(p_1\sigma), \cdots, \theta^*(p_n\sigma))$. So they are bounded by

$\omega_{\mathtt{f}}(\ldots, \max_{i=1..n}(\theta^*(p_i\sigma)), \ldots)$. So the number of constructor symbols added by such a call is bounded by $Q_{\mathtt{g}}(\omega_{\mathtt{f}}(\ldots, \max_{i=1..n}(\theta^*(p_i\sigma)), \ldots))$. Finally, since the sup-interpretations of contexts oblige to take the maximum of such calls, we have:

$$E_{>_{Fct}} \leq \max_{\mathtt{g} \approx_{Fct} \mathtt{f}} (Q_{\mathtt{g}}(\omega_{\mathtt{f}}(\ldots, \max_{i=1..n}(\theta^*(p_i\sigma)), \ldots)))$$

So we have:

$$\begin{aligned} \|\mathtt{f}(v_1, \cdots, v_n)\| &= E_{>_{Fct}} + E_{\approx_{Fct}} \\ &\leq P_{\mathtt{f}}(\max_{i=1..n}(\theta^*(p_i\sigma))) \end{aligned}$$

Lastly, if e_i is not a fraternity. Then we have already a bound $Q_{\mathtt{f}}$ on the size of the computed value. Thus

$$\begin{aligned} \|\mathtt{f}(v_1, \cdots, v_n)\| &\leq Q_{\mathtt{f}}(\omega_{\mathtt{f}}(\ldots, \max_{i=1..n}(\theta^*(p_i\sigma)), \ldots)) \\ &\leq P_{\mathtt{f}}(\max_{i=1..n}(\theta^*(p_i\sigma))) \end{aligned}$$

Thus proving theorem 3. Finally, we have:

$$\begin{aligned} \|\mathtt{f}(v_1, \cdots, v_n)\| &\leq P_{\mathtt{f}}(\max_{i=1..n}(\theta^*(v_i))) && \text{by theorem 3} \\ &\leq P_{\mathtt{f}}(\max_{i=1..n}(\alpha(|v_i|))) && \text{by lemma 2} \\ &\leq P_{\mathtt{f}}(\alpha(\max_{i=1..n}(|v_i|))) \end{aligned}$$

Thus proving theorem 1.

A.3 Proof of theorem 2

We first begin by proving the following lemma and then we define a notion of rank:

Lemma 5. *Given a friendly program, for every rule of the shape* $\mathtt{f}(p_1, \cdots, p_n) \to \mathsf{C}[e]$ *and every subsitution* σ*, there is a polynomial* Q *such that:*

$$\|e\sigma\| \leq Q(\max_{i=1..n}(|p_i\sigma|))$$

Proof. The proof is done by induction on the height $ht(e)$ of a term e. If $ht(e) = 0$ then the term is either a constant or a variable. We have $\|e\sigma\| \leq \max_{i=1..n}(|p_i\sigma|)$. Suppose that the result holds for terms d of height $ht(d) = k$. And consider the term e of height $k+1$. e is of the shape $h(e_1, \ldots, e_n)$ with $\max_{i=1..n}(ht(e_i)) = k$. Thus, by induction hypothesis $\|e_i\sigma\| \leq Q_i(\max_{j=1..n}(|p_j\sigma|))$.

$$
\begin{aligned}
\|e\sigma\| &= \|h([\![e_1\sigma]\!], \ldots, [\![e_n\sigma]\!])\| && \text{for the call by value} \\
&\leq P(\max_{i=1..n} \|e_i\sigma\|) && \text{by Th. 1} \\
&\leq P(\max_{i=1..n} Q_i(\max_{j=1..n}(|p_j\sigma|))) && \text{by monotonicity of P} \\
&\leq Q(\max_{j=1..n}(|p_j\sigma|))
\end{aligned}
$$

Notice that the coefficients of the polynomial Q only depend on the size of the program since the height of a term is strictly bounded by the size of the program.

Definition 10 (rank). *We define a notion of rank rk on the class of equivalent function symbols for $\geq_{Fct}$ in one branch of the call-tree. The calling symbol $\mathtt{f}$ is of rank 0.*

- *If there is a transition $\langle \mathtt{f}, v_1, \cdots, v_n\rangle \rightsquigarrow \langle \mathtt{g}, u_1, \cdots, u_m\rangle$ and $\mathtt{f} >_{Fct} \mathtt{g}$ with $rk(\mathtt{f}) = k$ then $rk(\mathtt{g}) = k + 1$*

- *If there is a transition $\langle \mathtt{f}, v_1, \cdots, v_n\rangle \rightsquigarrow \langle \mathtt{g}, u_1, \cdots, u_m\rangle$ and $\mathtt{f} \approx_{Fct} \mathtt{g}$ with $rk(\mathtt{f}) = k$ then $rk(\mathtt{g}) = k$*

Theorem 2. *Assume that $\mathbf{p}$ is a friendly program. For each function symbol $\mathtt{f}$ of $\mathbf{p}$ 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 defined.

Proof. We show how to build such a polynomial R. If $rk(\mathtt{g}) = 0$ then $\mathtt{g} \approx_{Fct} \mathtt{f}$ and then by lemma 4 and 3,

$$\max_{j=1..m} |u_j| \leq \omega_{\mathtt{g}}(\theta^*(u_1), \cdots, \theta^*(u_m)) \leq \omega_{\mathtt{f}}(\theta^*(v_1), \cdots, \theta^*(v_n))$$

Taking $R_0(X) = \omega_{\mathtt{f}}(\alpha(X), \ldots, \alpha(X))$ we obtain the required polynomial bound. Suppose that we have built a polynomial R_k for the function symbols of rank k and that $rk(\mathtt{g}) = k+1$. There is a dependency pair $\langle \mathtt{f}_1(q_1, \cdots, q_l), \mathtt{g}_1(s_1, \cdots, s_{l'})\rangle$ involved in the branch $\langle \mathtt{f}, v_1, \cdots, v_n\rangle \overset{+}{\rightsquigarrow} \langle \mathtt{g}, u_1, \cdots, u_m\rangle$ of the call-tree, for a given substitution σ, and such that $rk(\mathtt{f}_1) = k$ and $rk(\mathtt{g}_1) = k+1$ (i.e. $\mathtt{g} \approx_{Fct} \mathtt{g}_1$). We know that $\max_{i=1..l} |q_i\sigma| \leq R_k(\max(|v_1|, ..., |v_n|))$.

$$
\begin{aligned}
\max_{i=1..m} |u_i| &\leq \omega_{\mathtt{g}_1}(\theta^*([\![s_1\sigma]\!]), \ldots, \theta^*([\![s_{l'}\sigma]\!])) && \text{by lemma 4}\\
&\leq \omega_{\mathtt{g}_1}(\alpha(\max_{i=1..n} \|s_i\sigma\|), \ldots, \alpha(\max_{i=1..n} \|s_i\sigma\|)) && \text{by lemma 2}\\
&\leq \omega_{\mathtt{g}_1}(\ldots, \alpha(Q(\max_{i=1..l} |q_i\sigma|)), \ldots) && \text{by lemma 5}\\
&\leq \omega_{\mathtt{g}_1}(\ldots, \alpha(Q(R_k(\max_{j=1..n} |v_j|))), \ldots) && \text{by lemma I.H.}
\end{aligned}
$$

Then we put $R_{k+1}(X) = \max(R_k(X), \omega_{\mathtt{g}_1}(\ldots, \alpha(Q(R_k(X))), \ldots)$. Finally, for the highest rank l we put $R(X) = R_l(X)$. Such a l exists since the rank is bounded by the number of function symbols and consequently by the size of the program $|\mathbf{p}|$. Consequently, the polynomial is independant of the inputs. Now we prove by induction on the rank that for every k R_k gives a bound on the size of the values belonging to states whose rank of function symbol is smaller than k. We have shown that it is true for $k = 0$. Suppose that it is true for k. By construction and induction hypothesis, R_{k+1} bounds the values of the states of rank smaller or equal to k. Given a state $\langle \mathtt{h}, w_1, \cdots, w_{l'}\rangle$ of the branch with $rk(\mathtt{h}) = k+1$ (i.e. $\mathtt{h} \approx_{Fct} \mathtt{g}_1$).

$$
\begin{aligned}
\max_{i=1..l'} |w_i| &\leq \omega_{\mathtt{g}_1}(\theta^*([\![s_1\sigma]\!]), \ldots, \theta^*([\![s_m\sigma]\!])) && \text{by lemma 7}\\
&\leq \omega_{\mathtt{g}_1}(\ldots, \alpha(Q(R_k(\max_{j=1..n} |v_j|))), \ldots) && \text{cf. above}\\
&\leq R_{k+1}(\max_{j=1..n} |v_j|) && \text{by construction of } R_{k+1}
\end{aligned}
$$

Finally R gives a bound on the size of every value of the call-tree, even if the call-tree is infinite (In this case, the program is non-terminating and it adds at least an infinite branch of states whose respective function symbols are equivalent for the precedence $\geq_{Fct}$).

B Examples

Definition 11 (Dependency pairs graph). *Let $\boldsymbol{DP}(\boldsymbol{p})$ be the set of all dependency pairs of a program $\boldsymbol{p}$. The dependency pair graph $\boldsymbol{DPG}(\boldsymbol{p})$ of a program is defined by*

- *The set of nodes is $\boldsymbol{DP}(\boldsymbol{p})$.*
- *There is an edge between $\langle \boldsymbol{f}(p_1, \cdots, p_n), \boldsymbol{g}(t_1, \cdots, t_m)\rangle$ and $\langle \boldsymbol{g}(q_1, \cdots, q_m), \boldsymbol{h}(s_1, \cdots, s_k)\rangle$.*

The notion of dependency pairs graph is useful in order to have a clearest view of programs. In fact, the call-tree of a program is no more than an unfolding of the dependency pair graph.

Example 6. Consider the program for multiplication:

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

We define the sup-interpretation θ by taking $\theta(\mathtt{add})(X, Y) = X{+}Y$ and $\theta(\mathbf{S})(X) = X{+}1$. The program owns two fraternities $\mathtt{add}(y, [\mathtt{mult}(z, u)]))$ and $\mathbf{S}(\mathbf{S}[\mathtt{add}(u, v)])$ called respectively by $\mathtt{mult}(\mathbf{S}(z), u)$ and $\mathtt{add}(\mathbf{S}(u), \mathbf{S}(v))$. By taking $\omega_{\mathtt{add}}(X, Y) = \omega_{\mathtt{mult}}(X, Y) = X{+}Y$, the inequalities of the friendly criteria are strictly statisfied for every subsitution σ. So it remains to check the context's sup-interpretations. Since $\theta^*(\mathbf{S}(\mathbf{S}(\diamond))) = \diamond + 2$ and $\theta^*(\mathtt{add}(y, \diamond))) = \diamond + \theta^*(y) = \diamond + Y$, $\mathbf{p}_{\mathtt{mult}}$ is a friendly program. This example illustrates an interesting case where the polynomial $R(X) = X$ added by the contexts and defined in the friendly criteria is no longer a constant.

Example 7. Consider the program for exponential:

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

$\theta(\texttt{double})(X)$ is at least equal to $2X$ and the program contains a fraternity involving `double` in its context. Consequently, the program $\mathbf{p}_{\text{exp}}$ is not friendly.

Example 8. The following program computes the greatest common divisor:

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

This program admits four cycles: two depending on `minus` and `le` which verify the friendly criteria and two other on the `gcd` function. Since the corresponding context to these latter cycles is the function `if` whose sup-interpretation can be taken to be max, and by taking $\theta(\mathbf{S})(X) = X + 1$, we only have to check that, there exist θ and ω such that:

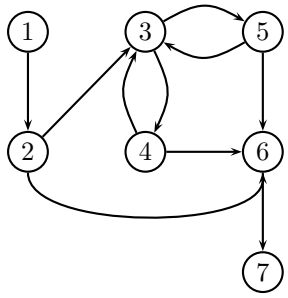
$$\begin{aligned}
\omega_{\texttt{gcd}}(U+1, V+1) &\geq \omega_{\texttt{gcd}}(\theta(\texttt{minus})(V,U), U+1) \\
\omega_{\texttt{gcd}}(V+1, V+1) &\geq \omega_{\texttt{gcd}}(\theta(\texttt{minus})(U,V), V+1)
\end{aligned}$$

We can take $\theta(\texttt{minus})(X,Y)$ equal to X and $\omega_{\texttt{gcd}}(X,Y) = X + Y$ so that the previous inequalities are satisfied over reals, consequently also on substitutions' sup-interpretation and the program is friendly.

Example 9. The following program eliminates duplicates from a list:

$$\begin{aligned}
\texttt{equi}(x,y) = \textbf{Case } x,y \textbf{ of } \mathbf{0},\mathbf{0} &\to \textbf{True}\\
\mathbf{0},\mathbf{S}(v) &\to \textbf{False}\\
\mathbf{S}(u),\mathbf{0} &\to \textbf{True}\\
\mathbf{S}(u),\mathbf{S}(v) &\to \texttt{equi}(u,v)\\
\texttt{remove}(x,y) = \textbf{Case } x,y \textbf{ of } n,\textbf{nil} &\to \textbf{nil}\\
n,\mathbf{c}(m,l) &\to \texttt{if}(\texttt{equi}(n,m),n,\mathbf{c}(m,l))\\
\texttt{if}(x,y,z) = \textbf{Case } x,y,z \textbf{ of } \textbf{True},n,\mathbf{c}(m,l) &\to \texttt{remove}(n,l)\\
\textbf{False},n,\mathbf{c}(m,l) &\to \mathbf{c}(m,\texttt{remove}(n,l))\\
\texttt{elim}(x) = \textbf{Case } x \textbf{ of } \textbf{nil} &\to \textbf{nil}\\
\mathbf{c}(n,l) &\to \mathbf{c}(n,\texttt{elim}(\texttt{remove}(n,l)))
\end{aligned}$$

This program admits the following dependency pair graph:



with

1. $< \texttt{elim}(\mathbf{c}(n,l)), \texttt{elim}(\texttt{remove}(n,l)) >$
2. $< \texttt{elim}(\mathbf{c}(n,l)), \texttt{remove}(n,l) >$
3. $< \texttt{remove}(n,\mathbf{c}(m,l)), \texttt{if}(\texttt{equi}(n,m),n,\mathbf{c}(m,l)) >$
4. $< \texttt{if}(\textbf{False},n,\mathbf{c}(m,l)), \texttt{remove}(n,l) >$
5. $< \texttt{if}(\textbf{True},n,\mathbf{c}(m,l)), \texttt{remove}(n,l) >$

6. $< \texttt{remove}(n, \mathbf{c}(m, l)), \texttt{equi}(n, m) >$
7. $< \texttt{equi}(\mathbf{S}(u), \mathbf{S}(v)), \texttt{equi}(u, v) >$

So the precedence holds as follows $\texttt{elim} >_{Fct} \texttt{remove} \approx_{Fct} \texttt{if} >_{Fct} \texttt{equi}$. Now we want to check the constraint on recursive calls. There are four cycles in the graph and we want to find ω and θ such that:

$\forall\sigma,\ \omega_{\texttt{elim}}(\theta^*(\mathbf{c}(n,l)\sigma)) \geq \omega_{\texttt{elim}}(\theta^*(\texttt{remove}(n,l)\sigma))$

$\forall\sigma,\ \omega_{\texttt{equi}}(\theta^*(\mathbf{S}(u)\sigma), \theta^*(\mathbf{S}(v)\sigma)) \geq \omega_{\texttt{equi}}(\theta^*(u\sigma), \theta^*(v\sigma))$

$\forall\sigma,\ \omega_{\texttt{remove}}(\theta^*(n\sigma), \theta^*(\mathbf{c}(m,l)\sigma)) \geq \omega_{\texttt{if}}(\theta^*(\texttt{equi}(n,m)\sigma), \theta^*(n\sigma), \theta^*(\mathbf{c}(m,l)\sigma))$

$\forall\sigma,\ \omega_{\texttt{if}}(\theta^*(\mathbf{False}), \theta^*(n\sigma), \theta^*(\mathbf{c}(m,l)\sigma)) \geq \omega_{\texttt{remove}}(\theta^*(n\sigma), \theta^*(l\sigma))$

$\forall\sigma,\ \omega_{\texttt{if}}(\theta^*(\mathbf{True}), \theta^*(n\sigma), \theta^*(\mathbf{c}(m,l)\sigma)) \geq \omega_{\texttt{remove}}(\theta^*(n\sigma), \theta^*(l\sigma))$

We take $\omega_{\texttt{elim}}$ as the identity and $\omega = \sum$ otherwise. It remains to prove that :

1. $\forall\sigma,\ \theta^*(\mathbf{c}(n,l)\sigma) \geq \theta^*(\texttt{remove}(n,l)\sigma)$
2. $\forall\sigma,\ \theta^*(\mathbf{S}(u)\sigma) + \theta^*(\mathbf{S}(v)\sigma) \geq \theta^*(u\sigma) + \theta^*(v\sigma)$
3. $\forall\sigma,\ \theta^*(n\sigma) + \theta^*(\mathbf{c}(m,l)\sigma) \geq \theta^*(\texttt{equi}(n,m)\sigma) + \theta^*(n\sigma) + \theta^*(\mathbf{c}(m,l)\sigma)$
4. $\forall\sigma,\ \theta^*(\mathbf{False}) + \theta^*(n\sigma) + \theta^*(\mathbf{c}(m,l)\sigma) \geq \theta^*(n\sigma) + \theta^*(l\sigma)$
5. $\forall\sigma,\ \theta^*(\mathbf{True}) + \theta^*(n\sigma) + \theta^*(\mathbf{c}(m,l)\sigma) \geq \theta^*(n\sigma) + \theta^*(l\sigma)$

Inequalities 2, 4 and 5 are strict since **True**, **False**, **S** and **c** are constructors of respective arities 0, 0, 1 and 2. For inequality 1, we take $\theta^*(\texttt{remove}(n,l)) = L$ since the result of the `remove` function is a list smaller than the input one. So inequality 1 is strictly satisfied. Finaly, we can take $\theta(\texttt{equi}(n,m)) = 0$ since the result is a constructor of arity 0. As a result, the third constraint becomes an equality and is satisfied.

Now, consider the following definitions included in the program:

$$\begin{aligned}
\texttt{remove}(n, \mathbf{c}(m,l)) &= \texttt{if}(\texttt{equi}(n,m), n, \mathbf{c}(m,l)) \\
\texttt{if}(\mathbf{True}, n, \mathbf{c}(m,l)) &= \texttt{remove}(n,l) \\
\texttt{if}(\mathbf{False}, n, \mathbf{c}(m,l)) &= \mathbf{c}(m, \texttt{remove}(n,l))
\end{aligned}$$

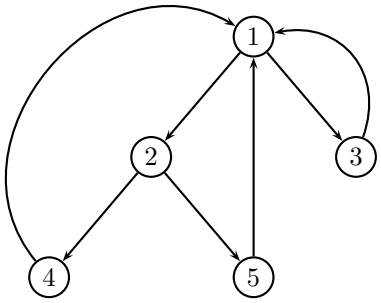
Since $\texttt{if} \approx_{Fct} \texttt{remove}$, the three respective fraternities have the following contexts $C^1[\diamond] = \diamond$, $C^2[\diamond] = \diamond$ and $C^3[\diamond] = \mathbf{c}(m, \diamond)$. $\theta(C^3[\diamond]) = \diamond + M + 1 =$

$\max(\diamond + R(M))$ with $R = M + 1$ and $\theta^*(m) = M$. Since the corresponding inequality is strict our program is friendly. Finally, as a consequence of theorem 1, there exist $P_{\texttt{elim}}$ such that for every value v_1:$\|\texttt{elim}(v_1)\| \leq P_{\texttt{elim}}(|v_1|)$

Example 10. The following program computes the reachability problem for graphs:

$$
\begin{aligned}
\texttt{equi}(x, y) = \textbf{Case } x, y \textbf{ of}& \\
\mathbf{0}, \mathbf{0} &\to \mathbf{True} \\
\mathbf{0}, \mathbf{S}(z) &\to \mathbf{False} \\
\mathbf{S}(z), \mathbf{0} &\to \mathbf{False} \\
\mathbf{S}(u), \mathbf{S}(v) &\to \texttt{equi}(u, v) \\
\texttt{or}(x, y) = \textbf{Case } x, y \textbf{ of}& \\
\mathbf{True}, y &\to \mathbf{True} \\
\mathbf{False}, y &\to \mathbf{False} \\
\texttt{union}(x, y) = \textbf{Case } x, y \textbf{ of}& \\
\epsilon, h &\to h \\
\mathbf{edge}(x, y, i), h &\to \mathbf{edge}(x, y, \texttt{union}(i, h)) \\
\texttt{reach}(x, y, z, u) = \textbf{Case } x, y, z, u \textbf{ of}& \\
x, y, \epsilon, h &\to \mathbf{False} \\
x, y, \mathbf{edge}(u, v, i), h &\to \texttt{if}_1(\texttt{equi}(x, u), x, y, \mathbf{edge}(u, v, i), h) \\
\texttt{if}_1(x, y, z, u, v) = \textbf{Case } x, y, z, u, v \textbf{ of}& \\
\mathbf{True}, x, y, \mathbf{edge}(u, v, i), h &\to \texttt{if}_2(\texttt{equi}(y, v), x, y, \mathbf{edge}(u, v, i), h) \\
\mathbf{False}, x, y, \mathbf{edge}(u, v, i), h &\to \texttt{reach}(x, y, i, \mathbf{edge}(u, v, h)) \\
\texttt{if}_2(x, y, z, u) = \textbf{Case } x, y, z, u \textbf{ of}& \\
\mathbf{True}, x, y, \mathbf{edge}(u, v, i), h &\to \mathit{true} \\
\mathbf{False}, x, y, \mathbf{edge}(u, v, i), h &\to \texttt{or}(\texttt{reach}(x, y, i, h), \\
& \qquad \texttt{reach}(v, y, \texttt{union}(i, h), \epsilon))
\end{aligned}
$$

Here is the corresponding dependency graph (without the `equi` and `union` function symbols):



with the nodes corresponding to the following dependency pairs:

1. $< \texttt{reach}(x, y, \textbf{edge}(u, v, i), h), \texttt{if}_1(\texttt{equi}(x, u), x, y, \textbf{edge}(u, v, i), h) >$
2. $< \texttt{if}_1(\textbf{True}, x, y, \textbf{edge}(u, v, i), h), \texttt{if}_2(\texttt{equi}(y, v), x, y, \textbf{edge}(u, v, i), h)) >$
3. $< \texttt{if}_1(\textbf{False}, x, y, \textbf{edge}(u, v, i), h), \texttt{reach}(x, y, i, \textbf{edge}(u, v, h)) >$
4. $< \texttt{if}_2(\textbf{True}, x, y, \textbf{edge}(u, v, i), h), \texttt{reach}(x, y, i, h) >$
5. $< \texttt{if}_2(\textbf{True}, x, y, \textbf{edge}(u, v, i), h), \texttt{reach}(x, y, i, h) >$

We can see that the graph is composed of 3 smallest cycles. Since each dependency pair involve empty contexts or the `or` function symbol whose sup-interpretation can clearly be taken to be the maximum of its inputs, we only have to check non strict inequalities to ensure the friendly criteria. Since the `equi` function symbol only returns booleans, we can take its sup-interpretation to be null. The sup-interpretation of the **edge** constructor can be assimilated to the size of the value. It remains to find a sup-interpretation θ and a weight ω such that(if the inequalities are satisfied over reals, they are also satisfied for every sup-interpretation of a substitution of a variable):

$$\omega_{\texttt{reach}}(X, Y, U + V + I + 1, H) \geq \omega_{\texttt{if}_1}(0, X, Y, U + V + I + 1, H)$$
$$\omega_{\texttt{if}_1}(0, X, Y, U + V + I + 1, H) \geq \omega_{\texttt{if}_2}(0, X, Y, U + V + I + 1, H)$$
$$\omega_{\texttt{if}_2}(0, X, Y, U + V + I + 1, H) \geq \omega_{\texttt{reach}}(X, Y, I, H)$$
$$\omega_{\texttt{if}_2}(0, X, Y, U + V + I + 1, H) \geq \omega_{\texttt{reach}}(V, Y, \theta(\texttt{union})(I, H), 0)$$
$$\omega_{\texttt{if}_1}(0, X, Y, U + V + I + 1, H) \geq \omega_{\texttt{reach}}(X, Y, I, H + U + V + 1)$$

Since the function symbol `union` makes the union between two graphs, we can take its sup-interpretation to be the sum of the size of the two graphs. FinalLy,

we set $\omega_{\texttt{if}_1}(Z,X,Y,U,V) = \omega_{\texttt{if}_2}(Z,X,Y,U,V) = \max(U+V,X,Y,Z)$ and $\omega_{\texttt{reach}}(X,Y,U,V) = \max(X,Y,U,V)$. Thus obtaining:

$$\begin{aligned}
\max(X,Y,U+V+I+1,H) &\geq \max(X,Y,U+V+I+1+H)\\
\max(X,Y,U+V+I+1+H) &\geq \max(X,Y,U+V+I+1+H)\\
\max(X,Y,U+V+I+1+H) &\geq \max(X,Y,I,H)\\
\max(X,Y,U+V+I+1+H) &\geq \max(V,Y,I+H)\\
\max(X,Y,U+V+I+1+H) &\geq \max(X,Y,I,H+U+V+1)
\end{aligned}$$

All these inequalities are satisfied, so the program is friendly.

C A termination criteria

In this section, using definitions and lemmas introduced in appendix A.1, we prove that the friendly criteria can be used to prove termination of programs.

Lemma 6. *For a friendly program and a branch* $\langle \mathtt{f}, u_1, \cdots, u_n\rangle \overset{+}{\rightsquigarrow} \langle \mathtt{g}, v_1, \cdots, v_k\rangle$ *of its call-tree with* $\mathtt{f} \approx_{Fct} \mathtt{g}$ *involving a lightening , then*

$$\omega_f(\theta^*(u_1), \cdots, \theta^*(u_n)) \geq 1/\delta + \omega_g(\theta^*(v_1), \cdots, \theta^*(v_m))$$

Proof. Suppose that lightening $u = \langle \mathtt{h}(p_1, \cdots, p_l), \mathtt{h'}(t_1, \cdots, t_{l'})\rangle$ corresponding to $\langle \mathtt{h}, v'_1, \cdots, v'_l\rangle \overset{\mathsf{C}[\diamond]}{\rightsquigarrow} \langle \mathtt{h}, w_1, \cdots, w_{l'}\rangle$ is involved in the branch of the call-tree, with $w_j = [\![t_j\sigma]\!]$ and $p_i\sigma = v'_i$ for a given subsitution σ, then by the friendly criteria:

$$\begin{aligned}
\omega_{\mathtt{h}}(\theta^*(p_1\sigma), \cdots, \theta^*(p_l\sigma)) &\geq 1/\delta_u + \omega_{\mathtt{h'}}(\theta^*(t_1\sigma), \cdots, \theta^*(t_{l'}\sigma)) \\
&\geq 1/\delta_u + \omega_{\mathtt{h'}}(\theta^*(w_1), \cdots, \theta^*(w_{l'})) \\
\omega_{\mathtt{h}}(\theta^*(v'_1), \cdots, \theta^*(v'_l)) &\geq 1/\delta + \omega_{\mathtt{h'}}(\theta^*(w_1), \cdots, \theta^*(w_{l'})) \qquad \text{by Dfn of } \delta
\end{aligned}$$

By lemma 3, we have

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

and

$$\omega_{\mathtt{h'}}(\theta^*(w_1), \cdots, \theta^*(w_{l'})) \geq \omega_{\mathtt{g}}(\theta^*(v_1), \cdots, \theta^*(v_m))$$

Finally,

$$\omega_{\mathtt{f}}(\theta^*(u_1), \cdots, \theta^*(u_n)) \geq 1/\delta + \omega_{\mathtt{g}}(\theta^*(v_1), \cdots, \theta^*(v_m))$$

Lemma 7. *Given a subsitution* σ*, suppose that we have a cycle in the evaluation of* $\mathtt{f}$ *starting from* $\langle \mathtt{f}, p_1\sigma, \cdots, p_n\sigma\rangle$ *in the call-tree of a friendly program. If there is a lightening* $u = \langle \mathtt{g}(q_1, \cdots, q_m), \mathtt{h}(s_1, \cdots, s_k)\rangle$ *involved in this cycle then there are at most* $\delta\omega(\theta^*(p_1\sigma), \ldots, \theta^*(p_n\sigma))$ *occurrences of the cycle in the call-tree.*

Proof. The proof is based on lemma 6. If we have a lightening involved in a cycle, we know that we decrease $\omega(\theta^*(p_1\sigma), \ldots, \theta^*(p_n\sigma))$ by at least $1/\delta$ for each occurrence of a cycle. Thus we have at most $\delta\omega(\theta^*(p_1\sigma), \ldots, \theta^*(p_n\sigma))$ occurrences.

Corollary 2. *If k lightenings are involved in a cycle of the call-tree of a friendly program, then there are at most $\delta\omega(\theta^*(p_1\sigma),\ldots,\theta^*(p_n\sigma))/k$ occurrences of the cycle in the call-tree.*

Theorem 4. *If the program is friendly and such that each cycle in its call-tree involved a lightening, then the program is terminating.*

Proof. There are no infinite loops in the program since we bound the number of occurrences of each cycle. Thus it terminates.

Remark 2. Notice that a version of previous theorem could also be applied to non-friendly programs such as exponential in example 7 of appendix A.1, if we forget the condition on sup-interpretation of contexts. In the case of exponential, we have to check the following inequalities:

$$\forall\sigma,\ \omega_{\texttt{double}}(\theta^*(\mathbf{S}(x\sigma))) > \omega_{\texttt{double}}(\theta^*(x\sigma))$$
$$\forall\sigma,\ \omega_{\exp}(\theta^*(\mathbf{S}(x\sigma))) > \omega_{\exp}(\theta^*(x\sigma))$$

This is done by taking $\omega_{\texttt{double}}(X) = \omega_{\exp}(X) = X$ so that exp is also terminating.