Course of value distinguishes the intentionality of programming languages

Guillaume Bonfante

Course of value distinguishes the intentionality of programming languages

Guillaume Bonfante*
Université de Lorraine
Nancy, France
bonfante@loria.fr

ABSTRACT

In this contribution, we propose to study the transformation of first order programs by course of value recursion. Our motivation is to show that this transformation provides a separation criterion for the intentionality of sets of programs. As an illustration, we consider two variants of the multiset path ordering, for the first, terms in recursive calls are compared with respect to the subterm property, for the second with respect to embedding. Under a quasi-interpretation, both characterize Ptime, the latter characterization being a new result. Once applied the transformation, we get respectively Ptime and Pspace thus proving that the latter set of programs contains more algorithms.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Theory of computing, theory of complexity and recursion

Keywords

Implicit computational complexity, Space and time resource evaluation, program interpretation

1. INTRODUCTION

This paper follows a line opened in [3] where we have shown that transformations of languages could be used to compare sets of programs. Let us come back to the key idea. In the field of implicit computational complexity, when characterizing a set of programs, the intentionality measures the 'quantity' of programs, not the 'quantity' of functions. Indeed, one function may be computed by many programs, possibly efficient. Thus, a set of programs may correspond to a small number of functions but with a large choice of programs. The larger choice, the easier to pick up one program. To show the strength of some theory (which defines a set of programs) one usually exhibit some remarkable example. However, the argument is usually rather weak since one has to compare programs, not functions. We point out the work of Asperti [2] who considers such complexity cliques, showing their intrinsic undecidability. See also [4] where we proved that characterizing programs computable in Ptime is Σ_2-complete.

To avoid such direct comparison of programs, we propose to compare sets of programs via some apparel. Let us suppose for the discussion that a program is a rewriting system, in other words, our study focuses on recursion. Let us suppose that we are given two sets S and S' of programs, and that these programs compute the same set U of functions. Suppose that under some transformation (of rewriting systems) T, the two sets $T(S)$ and $T(S')$ of programs yield different function sets $V_1 \neq V_2$, then, $T(S) \neq T(S')$ and consequently S and S' cannot be equivalent from an implicit computational complexity point of view: indeed, would they be the same, then $T(S) = T(S')$! The transformation works as a kind of lens, separating thus indistinguishable programs at the first level.

In [3], we have considered the effect of adding non-determinism to programs. We have shown that this transformation does not add some computational strength to constructor-free programs (see Jones [12]) which characterize functions in Ptime. On contrary, applying it on programs with a multi-set path ordering termination proof admitting a quasi-interpretation, one obtains a characterization of Pspace. Then, if Pspace is different from Ptime, we have shown the separation between the two sets of programs. We propose in this paper to consider a (kind of) course-of-value recursion transformation.

Referring to the study of recursive schema [15, 16], it is well known that course-of-values recursion, that is functions defined by equations of the shape:

$$\begin{aligned} f(0,y) &= g(y) \\ f(x+1,y) &= h(x,y,f(j_1(x),y),\ldots,f(j_k(x),y) \end{aligned}$$

where $g, h, j_1, \ldots, j_k$ are primitive recursive and $j_i(x) \leq x$ for $i \in \{1,\ldots,k\}$, are themselves primitive recursive. A similar remark holds for recursion with parameter substitu-

*Work partially supported by project ANR-08-BLANC-0211-03 (COMPLICE) and by EA Cristal.

SoICT 2011, Hanoi, Vietnam

tions as observed by Leivant and Marion in [14]:

$$\begin{aligned} f(0,y) &= g(y) \\ f(x+1,y) &= h(x,y,f(x,j_1(y)),\ldots,f(x,j_k(y))) \end{aligned}$$

Leivant and Marion have proved that restricted to safe recursion, primitive recursion with parameter substitution corresponds to the functions computable within Pspace. We will rediscover a similar characterization with course-of-value recursion in Section 4, Theorem 3(2).

As observed by Leivant, notice that the transformation of such recursive schemas into primitive recursion involves an encoding function which, in other terms, means that one uses the exponential function. For the study of classes of low complexity such as Ptime, it is then reasonable to treat both schema apart.

Actually, in the context of rewriting systems, we define a notion that subsumes both schemas. A program has a course-of-value recursive schema if a recursive call contains an argument which is computed by an other function symbol. In other words, there is a rule of the form $f(x_1,\ldots,x_n) = C[f(j_1(x_1),\ldots,j_n(x_n))]$ for some context functions $f, j_1, \ldots, j_n$. A typical example of course-of-value is given by the quick-sort recursion:

$$\texttt{qs}(\ell) = \texttt{insert}(\texttt{qs}(\texttt{first-half}(\ell)), \texttt{qs}(\texttt{second-half}(\ell)))$$

for a non empty list ℓ. The functions `first-half` and `second-half` respectively compute the first half and the second half of the list ℓ (with respect to some pivot). These two functions appear as arguments of `qs`, the quick-sort function.

To prove the termination of recursive programs, one shows that arguments decrease according to some well-founded order. A very large number of such orders have been considered so far. We consider two of them, one based on the subterm relation and one on embedding. We recall that for the subterm relation, one erases some of the leading symbols, while for embedding, the erasing can be done at any point. For instance $t_1 = a(a(b(\varepsilon)))$ is a subterm of $t_2 = c(c(a(a(b(\varepsilon)))))$ but not a subterm of $t_3 = a(a(c(b(\varepsilon))))$. But t_1 is both embedded in t_2 and t_3. The subterm relation and the embedding relation are well-founded. Thus, if one or more arguments of a recursive call decrease while the others stay unchanged, the program ends.

Is there a main difference between these two orderings? Let us restrict our attention to programs computing functions of polynomial growth (this will be ensured later by the interpretation). In both cases, for programs terminating by means of the subterm relation or the embedding relation, functions computed are the functions of Ptime. We want to stress the fact that the characterization for the path ordering based on the subterm relation already appeared in [4], but the one based on embedding is a new result.

But, if one applies course-of-value, the situation is different. In the first case (based on the subterm relation), either with or without course-of-value recursion we stick to functions in Ptime. In the second case, we reach Pspace. To conclude, we have established the non-equivalence of these two sets of programs.

As a matter of fact, as a by-product of our result, we show that adding course-of-value recursion to product-path ordering based on embedding together with a quasi-interpretation shifts the computed functions from Ptime to Pspace, a characterization closed to the one of Leivant and Marion mentioned above.

Section 2 defines the syntax of programs. The reader familiar with rewriting, interpretation and path ordering may directly jump to Section 3 which provides the new characterization of Ptime, namely given by programs with an additive quasi-interpretation and a proof by product-path-ordering based on embedding. Section 4 deals with the course-of-value extension of programs.

2. PROGRAMMING LANGUAGES

As said above, we consider rewriting systems as a model of functional programs. Apart from Definition 2, all the material is standard. We briefly recall the main definitions, essentially to fix some notations. Dershowitz and Jouannaud's survey [10] of rewriting is a good entry point for beginners.

Let $\mathcal{X}$ denotes a (countable) set of *variables*. A *signature* is a set of *symbols*, that is a pair of a name and a natural number, each symbol is referred to by its name. The number is called the symbol's arity. Given a signature Σ, the set of *terms* over Σ and $\mathcal{X}$ is denoted by $\mathcal{T}(\Sigma,\mathcal{X})$ and the set of *ground terms*, that is terms without variables, by $\mathcal{T}(\Sigma)$. The size $|t|$ of a term t is defined as the number of symbols in t. For instance, $|f(a,f(a,b))| = 5$. The height of a term, written $||t||$, is the longest path from the root to a leaf. $||f(a,f(a,b))|| = 2$. A *position* is a word in $\mathbf{N}^*$. Given a term t, a position q denotes a sub-term $t_{|q}$ by the equations:

$$\begin{aligned} t_{|\varepsilon} &= t \text{ for the empty word} \varepsilon \\ t_{|q} &= t_{i|q'} \text{ if } t = f(t_1,\ldots,t_n) \text{ and } q = i\cdot q' \end{aligned}$$

Finally, as a general notation, we use $\vec{u}$ as a shorthand to denote sequences of terms $u_1,\ldots,u_n$.

Definition 1 (subterm and embedding relations). *The subterm relation is the smallest binary relation, denoted $\trianglelefteq$, on terms such that* $\dfrac{}{t \trianglelefteq t}$ *and* $\dfrac{t \trianglelefteq u_i}{t \trianglelefteq f(u_1,\ldots,u_n)}$

for all terms $t, u_1, \ldots, u_n$ and f an n-ary symbol. The embedding relation, denoted $\trianglelefteq\!\!\!\!\blacktriangleleft$, is the smallest relation verifying:

$$\frac{}{t \trianglelefteq\!\!\!\!\blacktriangleleft t} \qquad \frac{t \trianglelefteq\!\!\!\!\blacktriangleleft u_i}{t \trianglelefteq\!\!\!\!\blacktriangleleft f(u_1,\ldots,u_n)} \qquad \frac{(t_i \trianglelefteq\!\!\!\!\blacktriangleleft u_i)_{i=1,\ldots,n}}{f(t_1,\ldots,t_n) \trianglelefteq\!\!\!\!\blacktriangleleft f(u_1,\ldots,u_n)}$$

for all terms $t, t_1, \ldots, t_n, u_1, \ldots, u_n$ and f an n-ary symbol.

From the definition, it is clear that $\trianglelefteq \subseteq \trianglelefteq\!\!\!\!\blacktriangleleft$, that is for all terms t, v, $t \trianglelefteq v \Rightarrow t \trianglelefteq\!\!\!\!\blacktriangleleft v$. The inclusion is strict as shown by the example:

Example 1. *Let us suppose that $1,2,3,4,5,6$ are unary constructors. Then,*

$$\begin{aligned} &1(3(5(\bullet))) \trianglelefteq\!\!\!\!\blacktriangleleft 1(2(3(4(5(6(\bullet)))))) \\ &1(3(5(\bullet))) \not\trianglelefteq 1(2(3(4(5(6(\bullet)))))) \\ &4(5(6(\bullet))) \trianglelefteq 1(2(3(4(5(6(\bullet)))))) \\ &4(5(6(\bullet))) \trianglelefteq\!\!\!\!\blacktriangleleft 1(2(3(4(5(6(\bullet)))))) \end{aligned}$$

A *context* is a term t whose variables occur only once. $t[u_1,\ldots,u_n]$ is a shorthand for $t[x_1 \leftarrow u_1,\ldots,x_n \leftarrow u_n]$ where $x_1,\ldots,x_n$ are the variables of t given in some a priori order. Actually, from the context, no confusion should arise from the variables.

We say that two contexts t_1 and t_2 are *compatible* ($t_1 \mid_{com} t_2$) if either t_1 or t_2 is a variable or $t_1 = f(u_1, \cdots, u_n)$, $t_2 = f(v_1, \cdots, v_n)$ and for all $i \leq n$, $u_i \mid_{com} v_i$.

The *extension of two compatible contexts* t_1 and t_2, denoted $t_1 + t_2$, is defined by:

$$\begin{array}{rcl} x + t & = & t \\ t + x & = & t \\ f(u_1, \cdots, u_n) + f(v_1, \cdots, v_n) & = & f(u_1 + v_1, \ldots, u_n + v_n) \end{array}$$

where x denotes a variable and f is a symbol. In the definition above, we suppose without loss of generality that t_1 and t_2 do not share variables.

EXAMPLE 2. *For instance,* $\boldsymbol{cons}(\boldsymbol{cons}(x, y), z)$ *is compatible with* $\boldsymbol{cons}(u, \boldsymbol{cons}(v, t))$. *And* $\boldsymbol{cons}(\boldsymbol{cons}(x, y), z) + \boldsymbol{cons}(u, \boldsymbol{cons}(v, t)) = \boldsymbol{cons}(\boldsymbol{cons}(x, y), \boldsymbol{cons}(v, t))$.

DEFINITION 2 (COMPATIBLE EXTENSION). *Given some terms* t_1', t_1, t_2 *such that* $t_1' \trianglelefteq t_1$ *and* $t_1 \mid_{com} t_2$, *the* compatible extension *of* t_1' *w.r.t.* t_2 *is defined as the largest term* t *such that* $t \trianglelefteq t_1 + t_2$ *and* $t \mid_{com} t_1'$. *It is defined by the following equations.*

- $ext(t_1', t_1, x) = t_1'$ *and* $ext(t_1', x, t_2) = t_2$ *whenever* x *a variable.*
- *Otherwise,* $t_1 = f(v_1, \ldots, v_n)$, $t_2 = f(w_1, \ldots, w_n)$ *and either*
  - $t_1' = t_1$, *then* $ext(t_1, t_1, t_2) = t_1$,
  - *or* $t_1' \trianglelefteq v_i$ *for some* $i \leq n$, *then* $ext(t_1', t_1, t_2) = ext(t_1', v_i, t_2)$,
  - *or* $v_i \trianglelefteq w_i$ *for all* $i \leq n$, *then* $ext(t_1', t_1, t_2) = f(ext(u_1, v_1, w_1), \ldots, ext(u_n, v_n, w_n))$.

EXAMPLE 3. *Consider for instance the terms* $\boldsymbol{cons}(x, z) \trianglelefteq \boldsymbol{cons}(\boldsymbol{cons}(x, y), z)$ *and* $\boldsymbol{cons}(u, \boldsymbol{cons}(v, t))$. *We have:* $ext(\boldsymbol{cons}(x, z), \boldsymbol{cons}(\boldsymbol{cons}(x, y), z), \boldsymbol{cons}(u, \boldsymbol{cons}(v, t)) = \boldsymbol{cons}(x, \boldsymbol{cons}(v, t))$.

PROPOSITION 1. *Given a term* t *and two contexts* $t_1[x_1, \ldots, x_n]$, $t_2[y_1, \ldots, y_m]$ *such that* $t = t_1[u_1, \ldots, u_n] = t_2[v_1, \ldots, v_m]$, *then* t_1 *and* t_2 *are compatible.*

PROOF. By induction on the definition. □

PROPOSITION 2. *Let us suppose given two compatible contexts* $t_1[x_1, \ldots, x_n]$, $t_2[y_1, \ldots, y_m]$. *Given a term* $t_1' \trianglelefteq t_1$, *let* $x_{i_1}, \ldots, x_{i_k}$ *be the subset of variables in* t_1' *occuring in* t_1. *There are positions* $p_1, \ldots, p_h$ *and indices* $j_1, \ldots, j_h \leq m$ *such that for all terms* $u_1, \ldots, u_n$, $v_1, \ldots, v_m$, *if* $t_1[u_1, \ldots, u_n] = t_2[v_1, \ldots, v_m]$, *then*

$$t' = t_1'[u_{i_1}, \ldots, u_{i_k}] = ext(t_1', t_1, t_2)[v_{j_1|_{p_1}}, \ldots, v_{j_h|_{p_h}}].$$

PROOF. By induction on the definitions. □

2.1 Syntax of programs

In this subsection, we briefly present rewriting systems, a simple model of functional programming.

In the sequel of the section, we suppose that we are given an algebra of terms given by a (finite) signature $\mathcal{C}$ on which computations are done. These terms are called *constructor* terms, and accordingly, symbols in the signature are called constructor symbols.

Next, we suppose given a set $\mathcal{F}$ of *function symbols*. A rule is a pair (ℓ, r), next written $\ell \to r$, where:

- $\ell = \mathtt{f}(p_1, \cdots, p_n)$ where $\mathtt{f} \in \mathcal{F}$ and $p_i \in \mathcal{T}(\mathcal{C}, \mathcal{X})$ for all $i = 1, \ldots, n$,
- and $r \in \mathcal{T}(\mathcal{C} \cup \mathcal{F}, \mathcal{X})$ is a term such that any variable occuring in r also occurs in ℓ.

DEFINITION 3. *A* program *is a quadruplet* $\mathtt{f} = \langle \mathcal{C}, \mathcal{F}, \mathtt{f}, \mathcal{E} \rangle$ *such that* $\mathcal{E}$ *is a finite set of rules. We distinguish among* $\mathcal{F}$ *a main function symbol whose name is given by the program name* $\mathtt{f}$.

The set of rules induces a rewriting relation written $\to$. The relation $\xrightarrow{*}$ is the reflexive and transitive closure of $\to$ and $t \xrightarrow{!} u$ denotes the fact that $t \xrightarrow{*} u$ and u is a normal form. All along, when it is not explicitly mentioned, we suppose programs to be *confluent*, that is, the rewriting relation is confluent.

The domain of the computed functions is the constructor term algebra $\mathcal{T}(\mathcal{C})$. The program $\mathtt{f} = \langle \mathcal{C}, \mathcal{F}, \mathtt{f}, \mathcal{E} \rangle$ computes a partial function $[\![\mathtt{f}]\!] : \mathcal{T}(\mathcal{C})^n \to \mathcal{T}(\mathcal{C})$ defined as follows. For every $u_1, \cdots, u_n \in \mathcal{T}(\mathcal{C})$, $[\![\mathtt{f}]\!](u_1, \cdots, u_n) = v$ iff $\mathtt{f}(u_1, \cdots, u_n) \xrightarrow{!} v$ and v is a constructor term. The notation is extended to terms by composition.

EXAMPLE 4. *Equality on binary words in* $\{0,1\}^*$, *boolean operations, membership in a list (built on* $\boldsymbol{cons}, \mathbf{nil}$*) are computed as follows.*

$$\begin{array}{rcl} \boldsymbol{\varepsilon} = \boldsymbol{\varepsilon} & \to & \boldsymbol{tt} \\ \mathbf{i}(x) = \mathbf{i}(y) & \to & x = y \text{ with } \mathbf{i} \in \{0,1\} \\ \mathbf{i}(x) = \mathbf{j}(y) & \to & \boldsymbol{ff} \text{ with } \mathbf{i} \neq \mathbf{j} \in \{0,1\} \\ \mathbf{i}(x) = \boldsymbol{\varepsilon} & \to & \boldsymbol{ff} \text{ with } \mathbf{i} \in \{0,1\} \\ \boldsymbol{\varepsilon} = \mathbf{i}(y) & \to & \boldsymbol{ff} \text{ with } \mathbf{i} \in \{0,1\} \\ \mathtt{not}(\boldsymbol{tt}) & \to & \boldsymbol{ff} \\ \mathtt{not}(\boldsymbol{ff}) & \to & \boldsymbol{tt} \\ \mathtt{or}(\boldsymbol{tt}, y) & \to & \boldsymbol{tt} \\ \mathtt{or}(\boldsymbol{ff}, y) & \to & y \\ \mathtt{and}(\boldsymbol{tt}, y) & \to & y \\ \mathtt{and}(\boldsymbol{ff}, y) & \to & \boldsymbol{ff} \\ \mathtt{if}\ \boldsymbol{tt}\ \mathtt{then}\ y\ \mathtt{else}\ z & \to & y \\ \mathtt{if}\ \boldsymbol{ff}\ \mathtt{then}\ y\ \mathtt{else}\ z & \to & z \end{array}$$

The definitions of a program induce a partial order on symbols, defined as the smallest transitive relation containing $g \preceq f$ for all f, g, such that there is a rule $f(p_1, \cdots, p_n) \to r$ and g occurs in r. Let $\prec$ denotes the strict part of $\preceq$ and $\simeq$ the equivalence relation induced by $\preceq$. The rank is then the equivalence class of a symbol w.r.t. $\simeq$. We define constructors to be symbols of a rank smaller than any other ranks.

Given a program $\langle \mathcal{C}, \mathcal{F}, f, \mathcal{E} \rangle$, we define $\mathcal{E}_{<f}$ to be the subset of $\mathcal{E}$ restricted to rules $g(p_1, \cdots, p_n) \to r$ such that $g \prec f$. The rewriting relation $\to_{<f}$ denotes the corresponding restriction to $\mathcal{E}_{<f}$. Given a symbol f, we define $f(t_1, \ldots, t_n) \to_f u$ iff

$$f(t_1, \ldots, t_n) \to^!_{<f} f(u_1, \ldots, u_n) \to r \to^!_{<f} u.$$

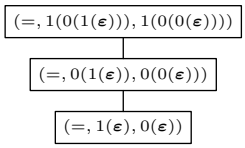
This notion is known as relative rewriting relation.

DEFINITION 4 (CALL-TREE). *Suppose we are given a program $\langle \mathcal{C}, \mathcal{F}, f, \mathcal{R}\rangle$. Let $\rightsquigarrow$ be the relation*

$$(f, t_1, \ldots, t_n) \rightsquigarrow (g, u_1, \ldots, u_m)$$
$$\Leftrightarrow$$
$$f(t_1, \ldots, t_n) \to C[g(w_1, \ldots, w_m)] \wedge \forall i : w_i \xrightarrow{!} u_i$$

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

EXAMPLE 5. *The call tree of $0(1(\varepsilon)) = 0(0(\varepsilon))$ is*



2.2 Interpretations of programs

The result presented in the following sections are correct whenever all computed functions have polynomial growth rate. However, to ensure that property (see corollary 1) in an effective way, we propose to use interpretations. Other methods could be used.

Given a signature Σ, a Σ-algebra on $\mathbf{R}$ is a mapping $(\!|-|\!)$ which associates to every n-ary symbol $f \in \Sigma$ an n-ary function $(\!|f|\!) : \mathbf{R}^n \to \mathbf{R}$.

DEFINITION 5. *Given a program $\langle \mathcal{C}, \mathcal{F}, f, \mathcal{E}\rangle$, let us consider a $(\mathcal{C} \cup \mathcal{F})$-algebra $(\!|-|\!)$ on $\mathbf{R}$. It is said to:*

1. *be weakly monotonic if for any symbol f, the function $(\!|f|\!)$ is a weakly monotonic function, that is if $x_i \geq x_i'$, then*
$$(\!|f|\!)(x_1, \ldots, x_n) \geq (\!|f|\!)(x_1, \ldots, x_i', \ldots, x_n),$$
2. *have the weak sub-term property if for any symbol f, the function $(\!|f|\!)$ verifies $(\!|f|\!)(x_1, \ldots, x_n) \geq x_i$ with $i \in 1, \ldots, n$,*
3. *to be weakly compatible if for all rules $\ell \to r$, $(\!|\ell|\!) \geq (\!|r|\!)$,*

An algebra verifying both three hypotheses is a called a quasi-interpretation.

Finally, we restrict the interpretations over the real numbers to be *Max-Poly functions*, that is functions obtained by finite compositions of the constant functions, maximum, addition and multiplication. In [5], we have proved that it is decidable to compute (if it exists!) an interpretation given a program.

The generalized geography game suppose that we are given a direct acyclic graph g, a node $r \in g$. Alternatively, player and opponent play a successor of the current node (player begins with r). The first who can't play (there is no more successors) looses. The question is: "does player has a winning strategy[1]? Even if the graph is binary, that is each node has at most two successors, the problem is known to be PSPACE-complete. The problem is solved by the following program.

[1]That is, he cannot loose if playing properly.

EXAMPLE 6.

$$\begin{aligned}
\mathtt{player}(g) &\to \mathtt{if}\ \mathtt{snd}(g) = \varepsilon\ \mathtt{then} \\
&\qquad \mathtt{if}\ \mathtt{fst}(g) = \varepsilon\ \mathtt{then}\ \mathbf{ff} \\
&\qquad \mathtt{else}\ \mathtt{not}(\mathtt{opp}(\mathtt{rem}_1(g))) \\
&\quad \mathtt{else}\ \mathtt{not}(\mathtt{and}(\mathtt{opp}(\mathtt{rem}_1(g)), \mathtt{opp}(\mathtt{rem}_2(g)))) \\
\mathtt{opp}(g) &\to \mathtt{if}\ \mathtt{snd}(g) = \varepsilon\ \mathtt{then} \\
&\qquad \mathtt{if}\ \mathtt{fst}(g) = \varepsilon\ \mathtt{then}\ \mathbf{ff} \\
&\qquad \mathtt{else}\ \mathtt{not}(\mathtt{opp}(\mathtt{rem}_1(g))) \\
&\quad \mathtt{else}\ \mathtt{not}(\mathtt{and}(\mathtt{player}(\mathtt{rem}_1(g)), \mathtt{player}(\mathtt{rem}_2(g))))
\end{aligned}$$

$$\begin{aligned}
\mathtt{fst}(\mathbf{cons}(T(x), \ell)) &\to \mathtt{fst}(\ell) \\
\mathtt{fst}(\mathbf{cons}(V(x,y,z), \ell) &\to y \\
\mathtt{fst}(\mathbf{cons}(\mathbf{i}(x), \ell)) &\to \mathtt{fst}(\ell) \\
\mathtt{fst}(\mathbf{nil}) &\to \varepsilon \\
\mathtt{snd}(\mathbf{cons}(T(x), \ell)) &\to \mathtt{snd}(\ell) \\
\mathtt{snd}(\mathbf{cons}(V(x,y,z), \ell) &\to z \\
\mathtt{snd}(\mathbf{cons}(\mathbf{i}(x), \ell)) &\to \mathtt{snd}(\ell) \\
\mathtt{snd}(\mathbf{nil}) &\to \varepsilon \\
\mathtt{rem}_1(g) &\to \mathtt{goto}(\mathtt{rem}(g), \mathtt{fst}(g)) \\
\mathtt{rem}_2(g) &\to \mathtt{goto}(\mathtt{rem}(g), \mathtt{snd}(g)) \\
\mathtt{rem}(\mathbf{cons}(T(x), \ell)) &\to \mathbf{cons}(T(x), \mathtt{rem}(\ell)) \\
\mathtt{rem}(\mathbf{cons}(V(x,y,z), \ell)) &\to \mathbf{cons}(x, \ell) \\
\mathtt{rem}(\mathbf{cons}(\mathbf{i}(x), \ell)) &\to \mathbf{cons}(\mathbf{i}(x), \mathtt{rem}(\ell)) \\
\mathtt{rem}(\mathbf{nil}) &\to \varepsilon \\
\mathtt{goto}(\mathbf{cons}(T(V(x,y,z)), \ell, m)) &\to \mathtt{if}\ m = x \\
&\qquad \mathtt{then}\ \mathbf{cons}(V(x,y,z), \ell) \\
&\qquad \mathtt{else} \\
&\qquad\quad \mathbf{cons}(T(V(x,y,z)), \mathtt{goto}(\ell, m)) \\
\mathtt{goto}(\mathbf{cons}(V(x,y,z), \ell), m) &\to \varepsilon \\
\mathtt{goto}(\mathbf{cons}(\mathbf{i}(x), \ell), m) &\to \mathbf{cons}(\mathbf{i}(x), \mathtt{goto}(\ell, m)) \\
\mathtt{goto}(\mathbf{nil}, m) &\to \varepsilon
\end{aligned}$$

Such a program admits a quasi-interpretation. Set

$$\begin{aligned}
(\!|\mathbf{nil}|\!) = (\!|\varepsilon|\!) = (\!|\mathbf{ff}|\!) = (\!|\mathbf{tt}|\!) &= 1 \\
(\!|T|\!)(x) = (\!|\mathbf{i}|\!)(x) &= x + 1 \\
(\!|\mathbf{cons}|\!)(x, y) &= x + y + 1 \\
(\!|V|\!)(x, y, z) &= x + y + z + 1
\end{aligned}$$

with $\mathbf{i} \in \{0, 1\}$. All other n-ary symbols f are mapped to $(\!|f|\!)(x_1, \ldots, x_n) = \max(x_1, \ldots, x_n)$.

At the beginning of the computation, the graph is given by an adjacent list of the form $\mathbf{cons}(e, \ell)$ where ℓ denotes the rest of the list and $e = T(V(x, y, z))$ denotes the (not yet visited) node x with two successors y and z.[2] In the computation process, the current node is denoted by $e = V(x, y, z)$ and a visited node by $e = x$. Finally, at the beginning of the computation, the root node is supposed to be denoted by $V(r, y, z)$ with y and z its successors.

Few notes on the program. $\mathtt{fst}(g)$ (respectively $\mathtt{snd}$) computes the first (resp. second) successor of the current node. $\mathtt{goto}(g, m)$ puts m as the current node of g, $\mathtt{rem}_1(g)$ removes the current node and puts its first successor as the current

[2]If x has one successor, z is supposed to be ε, if it has zero successor, then $y = z = \varepsilon$.

node. $\mathtt{rem}_2(g)$ *is analogous.* $\mathtt{rem}(g)$ *marks the current node as a visited node.*

DEFINITION 6. *The interpretation of a symbol f is said to be additive if it has the shape $\sum_i x_i + c$ for some constant $c \geq 1$. A program with an interpretation is said to be additive when its* constructors *are additive.*

For instance, the program given in Example 6 is additive. We recall a proposition and a corollary taken from [4]:

PROPOSITION 3. *Let $\langle \mathcal{C}, \mathcal{F}, f, \mathcal{E} \rangle$ be a program with an additive quasi-interpretation. Then, there is a constant $K > 0$ such that $(\!|t|\!) \leq K \times |t|$ for all constructor terms $t \in \mathcal{T}(\mathcal{C})$.*

Given a program with an additive quasi-interpretation, suppose that $u_1, \ldots, u_n \in \mathcal{T}(\mathcal{C})$. Whenever $t[u_1, \ldots, u_n] \xrightarrow{*} v$ with $v \in \mathcal{T}(\mathcal{C})$, then $|v| \leq (\!|t(u_1, \ldots, u_n)|\!)$. So, the size of the outputs of programs is directly linked to the value of the initial interpretation. For additive programs, Proposition 3 leads to $|v| \leq (\!|t|\!)(K|u_1|, \ldots, K|u_n|)$. Consequently,

COROLLARY 1. *The output of additive program has a polynomial size w.r.t. to its corresponding input.*

2.3 Termination proofs

Proving the termination of a program is a very common issue of software engineering. Many methods have been considered so far, and we are considering one of the most simple, namely path orderings which are decidable simplification orderings. The proof of the well-foundedness of $\prec_{mpo}$ is due to Dershowitz [9] and Kamin and Levy [13] and relies on Kruskal theorem. An formalized proof of the well-foundedness of the multiset path ordering by Coupet-Grimal and Delobel can be found in [8].

A precedence $\preceq_{\mathcal{F}}$ (strict precedence $\prec_{\mathcal{F}}$) is a quasi-ordering (strict partial ordering) on a given set $\mathcal{F}$ of function symbols. We define the equivalence relation $\approx_{\mathcal{F}}$ as $\mathtt{f} \approx_{\mathcal{F}} \mathtt{g}$ iff $\mathtt{f} \preceq_{\mathcal{F}} \mathtt{g}$ and $\mathtt{g} \preceq_{\mathcal{F}} \mathtt{f}$. This precedence $\preceq_{\mathcal{F}}$ on function symbols extends canonically to $\mathcal{C} \cup \mathcal{F}$ with the following relations $\forall \mathtt{f} \in \mathcal{F}$, $\forall \mathbf{c} \in \mathcal{C}$, $\mathbf{c} \prec_{\mathcal{F}} \mathtt{f}$.

DEFINITION 7. *Given an ordering $\prec$ on terms, its product extension $\prec$ over sequences (of equal length) is defined by: $(m_1, \cdots, m_k) \prec (n_1, \cdots, n_k)$ if and only if (i) $\forall i \leq k, m_i \preceq n_i$ and (ii) $\exists j \leq k$ such that $m_j \prec n_j$.*

It is clear that if the ordering $\prec$ on terms is well-founded, then its extension to sequences is also well-founded. This is the core point of the following ordering.

DEFINITION 8. *Given a program $\langle \mathcal{C}, \mathcal{F}, f, \mathcal{E} \rangle$ and a precedence $\preceq_{\mathcal{F}}$, we define the subterm product path ordering $\prec^{\trianglelefteq}_{mpo}$ by the rules of Figure 1. A program is ordered by $\prec^{\trianglelefteq}_{mpo}$ iff there is a precedence $\preceq_{\mathcal{F}}$ such that for each rule $l \to r$, the inequality $r \prec^{\trianglelefteq}_{mpo} l$ holds.*

REMARK 1. *The product path ordering is closed under substitutions and context.*

EXAMPLE 7. *The program of Example 4 can be ordered by product path ordering. Set $\prec$ to be* $\mathtt{or} \prec \mathtt{and} \prec \mathtt{not} \prec = \prec$ *it-the-else* $\prec \mathtt{in}$.

However, Example 6 cannot be ordered by the ordering. As we will see later, this is due to the rule

$$\begin{aligned} \mathtt{player}(g) \quad \to \quad & \mathtt{if}\ \mathtt{snd}(g) = \varepsilon\ \mathtt{then} \\ & \quad \mathtt{if}\ \mathtt{fst}(g) = \varepsilon\ \mathtt{then}\ \mathtt{ff} \\ & \quad \mathtt{else}\ \mathtt{not}(\mathtt{opp}(\mathtt{rem}_1(g))) \\ & \mathtt{else}\ \mathtt{not}(\mathtt{and}(\mathtt{opp}(\mathtt{rem}_1(g)), \mathtt{opp}(\mathtt{rem}_2(g)))). \end{aligned}$$

DEFINITION 9. *The embedding product path ordering $\prec^{\blacktriangleleft}_{mpo}$ is defined in exactly the same manner, but with constructors compared as in Figure 2.*

Since the subterm relation is included in the embedding relation, it is clear that programs ordered by $PPO_{\trianglelefteq}$ are ordered by $PPO_{\blacktriangleleft}$.

The main difference between $PPO_{\trianglelefteq}$ and $PPO_{\blacktriangleleft}$ is that $t \prec^{\trianglelefteq}_{mpo} u$ iff $t \triangleleft u$ for all constructor terms t, u. For $PPO_{\blacktriangleleft}$, we get a larger set $t \prec^{\blacktriangleleft}_{mpo} u$ iff $t \blacktriangleleft u$. As a consequence, a rule such as:

$$f(1(0(x))) \to f(1(x))$$

is not compatible with $\prec^{\trianglelefteq}_{mpo}$ but it is compatible with $\prec^{\blacktriangleleft}_{mpo}$.

3. CHARACTERIZATIONS OF PROGRAMS

In this section, we address the following problem. Is there a fundamental difference between the two product path orderings? We answer to the question by a characterization of the functions computed by means of the two orderings. Let us begin recalling the following theorem,

THEOREM 1. *[Bonfante, Marion and Moyen [4]] Functions computed by programs with an additive quasi-interpretation and a termination proof by $PPO_{\trianglelefteq}$ are exactly* PTIME *functions.*

Actually, it is safe to replace $PPO_{\trianglelefteq}$ by $PPO_{\blacktriangleleft}$ as shown by the Theorem:

THEOREM 2. *Functions computed by programs with an additive quasi-interpretation and a termination proof by $PPO_{\blacktriangleleft}$ are exactly* PTIME *functions.*

Since programs ordered by $PPO_{\trianglelefteq}$ are ordered by $PPO_{\blacktriangleleft}$, it is immediate that PTIME functions can be computed by programs ordered by $PPO_{\blacktriangleleft}$ with an additive quasi-interpretation. So, the issue is to prove that the converse holds.

The main issue with the embedding relation is that the set of terms $U_t = \{u \mid u \prec^{\blacktriangleleft}_{mpo} t\}$ does not have a polynomial size w.r.t. the size of the constructor term t. This property is a key feature used in the proof of [4] to apply memoisation. For the embedding relation, one may observe that all the sequences of booleans of size n verify: $\{0,1\}^n \subset U_{\underbrace{01\cdots01}_{n \text{ times}}}$. Thus, $|U_{\underbrace{01\cdots01}_{n \text{ times}}}| \geq 2^n$ leads to an exponential lower bound. This is bad news.

The remaining of the Section is devoted to the proof of Theorem 2. The core point of the proof is to show that actually, when evaluating $f(t_1, \ldots, t_n)$ for some constructor terms $t_1, \ldots, t_n$, not all the terms $f(u_1, \ldots, u_n)$ with $u_1 \blacktriangleleft t_1, \ldots, u_n \blacktriangleleft t_n$ will be evaluated, only a polynomial number of them. This is Proposition 6. After this crucial step, the proof follows the line of the proof of Theorem 1 in [4].

$$\frac{s \blacktriangleleft t}{s \prec^{\trianglelefteq}_{mpo} t} s,t \in \mathcal{T}(\mathcal{C}) \qquad \frac{s \preceq^{\trianglelefteq}_{mpo} t_i}{s \prec^{\trianglelefteq}_{mpo} f(\ldots,t_i,\ldots)} f \in \mathcal{F}\bigcup\mathcal{C} \qquad \frac{\forall i \leq n : s_i \prec^{\trianglelefteq}_{mpo} f(t_1,\cdots,t_n) \qquad g \prec_{\mathcal{F}} f}{g(s_1,\cdots,s_m) \prec^{\trianglelefteq}_{mpo} \mathtt{f}(t_1,\cdots,t_n)} g \in \mathcal{F}\bigcup\mathcal{C}, f \in \mathcal{F}$$

$$\frac{(s_1,\cdots,s_n) \prec^{\trianglelefteq}_{mpo} (t_1,\cdots,t_n) \qquad \mathtt{f} \approx_{\mathcal{F}} \mathtt{g}}{\mathtt{g}(s_1,\cdots,s_n) \prec^{\trianglelefteq}_{mpo} \mathtt{f}(t_1,\cdots,t_n)} g, f \in \mathcal{F}$$

Figure 1: The subterm product path ordering

$$\frac{s \blacktriangleleft t}{s \prec^{\blacktriangleleft}_{mpo} t} s,t \in \mathcal{T}(\mathcal{C}) \qquad \frac{s \preceq^{\blacktriangleleft}_{mpo} t_i}{s \prec^{\blacktriangleleft}_{mpo} f(\ldots,t_i,\ldots)} f \in \mathcal{F}\bigcup\mathcal{C} \qquad \frac{\forall i \leq n : s_i \prec^{\blacktriangleleft}_{mpo} f(t_1,\cdots,t_n) \qquad g \prec_{\mathcal{F}} f}{g(s_1,\cdots,s_m) \prec^{\blacktriangleleft}_{mpo} \mathtt{f}(t_1,\cdots,t_n)} g \in \mathcal{F}\bigcup\mathcal{C}, f \in \mathcal{F}$$

$$\frac{(s_1,\cdots,s_n) \prec^{\blacktriangleleft}_{mpo} (t_1,\cdots,t_n) \qquad \mathtt{f} \approx_{\mathcal{F}} \mathtt{g}}{\mathtt{g}(s_1,\cdots,s_n) \prec^{\blacktriangleleft}_{mpo} \mathtt{f}(t_1,\cdots,t_n)} g, f \in \mathcal{F}$$

Figure 2: The embedded product path ordering

In the rest of the Section, we suppose given a program $\langle \mathcal{C}, \mathcal{F}, f, \mathcal{E} \rangle$ which has a termination proof by $PPO_{\underline{\blacktriangleleft}}$ and a quasi-interpretation.

PROPOSITION 4.

1. *For each constructor terms t and s, $s \prec^{\blacktriangleleft}_{mpo} t$ iff $s \blacktriangleleft t$.*
2. *If t is a constructor term and u contains a function symbol, then $u \not\prec^{\blacktriangleleft}_{mpo} t$.*

PROOF. The proof is by induction on the definition of the path ordering. □

PROPOSITION 5. *For all rules $f(p_1,\cdots,p_n) \to r$, there is a context C such that*

- *$r = C[f_1(\vec{u_1}),\ldots,f_k(\vec{u_k})]$,*
- *C does not contain any symbol of rank of f,*
- *$f_1,\ldots,f_k$ have the rank of f,*
- *the $\vec{u_i}$'s are constructor terms.*

PROOF. We define C to be the largest context such that $r = C[t_1,\ldots,t_n]$ and

- C does not contain any symbol of rank of f
- for all i, $t_i = f_i(\vec{u_i})$ with f_i of rank of f.

The key point is to verify that the $\vec{u_i}$'s are actually sequences of constructor terms. Let us remind that the patterns $p_1,\ldots,p_n$ are themselves constructor terms. Notice also that for all $i \leq n$, since $t_i = f_i(\vec{u_i}) \preceq^{\blacktriangleleft}_{mpo} r \prec^{\blacktriangleleft}_{mpo} f(p_1,\cdots,p_n)$, it is necessarily the case that $(\vec{u_i}) \prec^{\blacktriangleleft}_{mpo} (\vec{p})$. Then, by definition $u_i \underline{\blacktriangleleft} p_i$ for all $i \leq n$ and consequently, by Proposition 4 (2), the u_i's contain no function symbols. □

PROPOSITION 6. *There is a finite set of terms Θ such that:*

- *for all symbol $f \in \mathcal{F}$,*
- *for all constructor terms $t_1,\ldots,t_n$,*
- *for all constructor terms $u_1,\ldots,u_k$,*
- *for all function symbol g of rank of f such that $f(t_1,\ldots,t_n) \xrightarrow{*} C[g(u_1,\ldots,u_k)]$,*
- *for all $i \leq k$,*

there is an element $e \in \Theta$, some positions $q_1,\ldots,q_h$ and some indices $i_1,\ldots,i_h$ such that $u_i = e[t_{i_1|q_1},\ldots,t_{i_h|q_h}]$.

PROOF. Let Θ be the following set.

- $x \in \Theta$,
- for all $f(p_1,\cdots,p_n) \to r$, for all $i \leq n$, $p_i \in \Theta$,
- for all $q, r \in \Theta$, for all $p \underline{\blacktriangleleft} q$, $\mathrm{ext}(p,q,r) \in \Theta$.

This set is finite. Indeed, the height $||\mathrm{ext}(p,q,r)|| \leq \max(||p||,||r||)$, so that $\Theta \subseteq \{v \mid ||v|| \leq d\}$ with $d = \max\{||p_i|| \mid f(p_1,\cdots,p_n) \to r \in \mathcal{R}\}$.

Now, we proceed by induction on the length of the derivation.

For the base case, $f(t_1,\cdots,t_n) \xrightarrow{*} f(t_1,\cdots,t_n)$, the result is trivial (take $e = x, q = \varepsilon$).

Otherwise, $f(t_1,\cdots,t_n) \xrightarrow{*} C[g(u_1,\cdots,u_n)] \to C[C'[h(v_1,\cdots,v_n)]]$ after application of the rule $g(p_1,\cdots,p_n) \to C'[h(s_1,\cdots,s_n)]$ with the substitution σ. That is, $u_i = p_i\sigma$ and $v_i = s_i\sigma$ with $i \leq n$. By Proposition 4, $s_i \underline{\blacktriangleleft} p_j$ for some j and $v_i = s_i[\sigma(x_{j_1}),\ldots,\sigma(x_{j_\ell})]$ for the set $x_{j_1},\ldots,x_{j_\ell}$ of the variables of s_i, that is a subset of the variable of p_i. By induction, we can state that $u_i = p_i[\sigma(x_1),\ldots,\sigma(x_\ell)] = e[t_{i_1|q_1},\ldots,t_{i_h|q_h}]$ for some $e \in \Theta$ and some positions $q_1,\ldots,q_h$ and indices $i_1,\ldots,i_h$. Notice that $p_i \in \Theta$, and since $s_i \underline{\blacktriangleleft} p_i$, $\mathrm{ext}(s_i,p_i,e) \in \Theta$. According to Proposition 2, there are positions $r_1,\ldots,r_{h'}$ and indices $j_1,\ldots,j_{h'}$ such that:

$$v_i = \mathrm{ext}(s_i,p_i,e)[t_{j_1|q_{j_1}.r_1},\ldots,t_{j_{h'}|q_{j_{h'}}.r_{h'}}]$$

Since $\mathrm{ext}(s_i,p_i,e) \in \Theta$), we have the desired property. □

COROLLARY 2. *Given a program $\langle \mathcal{C}, \mathcal{F}, f, \mathcal{E} \rangle$ ordered by $PPO_{\underline{\blacktriangleleft}}$, $f \in \mathcal{F}$ and constructor terms $t_1,\ldots,t_n$, we denote*

by $T_{(f,t_1,\ldots,t_n)}$ the call tree rooted by $(f,t_1,\ldots,t_n)$ and by $R_{(f,t_1,\ldots,t_n)}$ the subset of nodes $(g,u_1,\ldots,u_m) \in T_{(f,t_1,\ldots,t_n)}$ such that $g \simeq f$. There is a polynomial P such that: $|R_{(f,t_1,\ldots,t_n)}| \leq P(\max(|t_1|,\ldots,|t_n|))$.

PROOF. As seen above, if $(g,u_1,\ldots,u_m) \in R_{(f,t_1,\ldots,t_n)}$, then for all $i \leq m$, $u_i = e[t_{i_1|_{q_1}},\ldots,t_{i_h|_{q_h}}]$ for some $e \in \Theta$ as defined above. One may observe that the number of positions for each q_i is bounded by $\sum_i |t_i|$. Let d be the maximal arity of the contexts $e \in \Theta$. Without loss of generality, we suppose d to be greater than the arity of any symbols in $\mathcal{F}$. We conclude, taking $P(x) = |\mathcal{F}| \times |\Theta| \times (d \times x)^{2d}$. □

LEMMA 1. *Let $\mathfrak{f}$ be a functional program with an additive quasi-interpretation. Then, there is a polynomial $Q_{\mathfrak{f}}$ such that for any equation $\ell \to r$ and all $e \trianglelefteq r$, $|[\![e]\!](t_1,\ldots,t_n)| \leq Q(\max(|t_1|,\ldots,|t_n|))$ with $t_1,\ldots,t_n$ constructor terms.*

PROOF. It is proved by induction on the structure of e, using Corollary 1. □

PROOF THEOREM 2. A naive implementation would lead to an exponential number of function calls. This issue is easily skipped if we switch from a call-by-value semantics to a call-by-value semantics with cache, see Figure 3. In the definition, we set $\mathfrak{S}$ to be the set of substitutions whose range is a subset of constructor terms.

This technique is very common in algorithmic studies, it is known as dynamic programming, a technique inspired from Andersen and Jones' rereading (cf. [1]) of a simulation technique over 2 way push-down automata by Cook (see [7]) also known as memoization.

Our general purpose is to show that the size of the cache remains polynomial w.r.t. the size of the input, that is

- there is a polynomial number of entries and
- each entry has polynomial size, in other words:
  - the size of the arguments of calls are polynomial
  - the size of output terms is polynomial

Since each step of the call-by-value semantics with cache can be performed in polynomial (actually linear) time w.r.t. the size of the cache, the items above are sufficient to ensure that the computation can be done in polynomial time.

Actually, one must also note that functions computed by additive program have output of polynomial size w.r.t. their inputs (Corollary 1). Consequently, to ensure that the cache has polynomial size, one only need to verify that arguments of function calls in the cache have polynomial size.

We will proceed by induction on the rank of symbols, and consequently, we provide a polynomial P_f to bound the size of the cache of the function f, depending only on the rank of f. Without loss of generality, we can suppose that $P_f \geq P_g$ if $g \prec f$.

First, let us define D to be the maximal arity of functions, A to be the number of function symbols and R to be the maximal size of the right-hand side of a rule. Finally, let us suppose we want to compute $f(t_1,\ldots,t_n)$. Elements in the cache will be terms $g(u_1,\ldots,u_m)$ such that $f(t_1,\ldots,t_n) \xrightarrow{*} C[g(u_1,\ldots,u_m)]$. Consequently, g has rank smaller or equal to the rank of f.

Base case.

Consider the evaluation of functions with minimal rank. Let us consider such a function f. Suppose that we are given a term $f(t_1,\ldots,t_n)$. Consider a term $g(u_1,\ldots,u_m)$ in the cache, since g has rank equivalent to the one of f, by Corollary 2, there are at most $P_D(\max(|t_1|,\ldots,|t_n|))$ such nodes. Due to Proposition 4(1), each of these argument has size bounded by $\max(|t_1|,\ldots,|t_n|)$. We set accordingly the polynomial $P_f(x) = x \times P_D(x)$.

Induction step.

Now, suppose f being of a higher rank. Suppose we want to compute $f(t_1,\ldots,t_n)$. Consider its call-tree $T_{f(t_1,\ldots,t_n)}$. There is actually a bijection between the set $S_{(f,t_1,\ldots,t_n)} = \{(g,v_1,\ldots,v_n) \mid (g,v_1,\ldots,v_n) \in T_{(f,t_1,\ldots,t_n)}\}$ and the set of entries in the cache $\text{Cache}_{f(t_1,\ldots,t_n)}$. Each element of the cache $(g,v_1,\ldots,v_k,v) \in \text{Cache}_{(f,t_1,\ldots,t_n)}$ corresponds to some node $(g,v_1,\ldots,v_k) \in T_{(f,t_1,\ldots,t_n)}$. To sum up, it is sufficient to count the number of elements in $S_{(f,t_1,\ldots,t_n)}$ verifying that their arguments have polynomial size.

Suppose that $f(t_1,\ldots,t_n) \xrightarrow{*} C[g(u_1,\ldots,u_m)]$ with g of equal rank to the one of f. Again, by Proposition 2, there are only $P(\max(|t_1|,\ldots,|t_n|))$ entries in the cache for some polynomial P. Each of these arguments having size smaller than $\max(|t_1|,\ldots,|t_n|)$ by embedding.

So, the main issue is to evaluate the number of entries in the cache involving functions of smaller rank. Let $\rightsquigarrow_f$ be the following binary relation. $(g,v_1,\ldots,v_k) \rightsquigarrow_f (h,w_1,\ldots,w_m)$ holds iff g has the rank of f and there exists some function symbol $h' \prec f$ and some terms $w'_1,\ldots,w'_k$ such that

$$(g,v_1,\ldots,v_k) \rightsquigarrow (h',w'_1,\ldots,w'_k) \rightsquigarrow^* (h,w_1,\ldots,w_\ell).$$

One may observe that $(g,\vec{u}) \rightsquigarrow_f (h,\vec{v})$ implies that $h \prec f$. Let $D_{(g,u_1,\ldots,u_k)} = \{(h,w_1,\ldots,w_\ell) \mid (g,v_1,\ldots,v_k) \rightsquigarrow_f (h,w_1,\ldots,w_\ell)\}$. The elements in $S_{(f,t_1,\ldots,t_n)}$ splits as follows:

$$S_{(f,t_1,\ldots,t_n)} = R_{(f,t_1,\ldots,t_n)} \cup \bigcup_{(g,\vec{u}) \in R_{(f,t_1,\ldots,t_n)}} D_{(g,\vec{u})}. \tag{1}$$

Let us compute the size of $D_{(g,u_1,\ldots,u_m)}$ for some node $(g,u_1,\ldots,u_m) \in R_{(f,t_1,\ldots,t_n)}$. The relation $\rightsquigarrow_f$ can be reformulated as $(g,u_1,\ldots,u_m) \rightsquigarrow_f (h,w_1,\ldots,w_k)$ iff one of the two statement holds:

$$(g,\vec{u}) \rightsquigarrow (h,\vec{w}) \wedge h \prec f \tag{2}$$

$$(g,\vec{u}) \rightsquigarrow (h',\vec{u'}) \rightsquigarrow_f (h,\vec{w}) \text{ with } h' \prec f \tag{3}$$

Let $G_{(g,u_1,\ldots,u_m)}$ denotes the set of nodes verifying Equation (2). Let us consider a node $(h,w_1,\ldots,w_k) \in G_{(g,u_1,\ldots,u_m)}$. Let us make the hypothesis (which we prove a little bit further) that there is a polynomial Q such that for all $i \leq k$, we have $|w_i| \leq Q(\max(|t_1|,\ldots,|t_n|))$. Then, by induction, the cache corresponding to the computation of $h(w_1,\ldots,w_k)$ is bounded by $P_h(Q(\max(|t_1|,\ldots,|t_n|)))$. Consequently, the size of $D_{(g,u_1,\ldots,u_m)}$ is bounded by

$$\sum_{(h,w_1,\ldots,w_k) \in G_{(g,u_1,\ldots,u_m)}} P_h(Q(\max(|t_1|,\ldots,|t_n|))).$$

Recall that $|G_{(g,u_1,\ldots,u_m)}| \leq R$. Consequently, taking P_H a polynomial bounding P_h for all such h, we get the bound $|D_{(g,u_1,\ldots,u_m)}| \leq R \times P_H(Q(\max(|t_1|,\ldots,|t_n|)))$.

(Constructor)

$$\frac{\mathbf{c} \in \mathcal{C} \quad \langle C_{i-1}, t_i\rangle \Downarrow_c \langle C_i, v_i\rangle}{\langle C_0, \mathbf{c}(t_1, \cdots, t_n)\rangle \Downarrow_c \langle C_n, \mathbf{c}(v_1, \cdots, v_n)\rangle}$$

(Read)

$$\frac{\langle C_{i-1}, t_i\rangle \Downarrow_c \langle C_i, v_i\rangle \quad (\mathtt{f}, v_1, \cdots, v_n, v) \in C_n}{\langle C_0, \mathtt{f}(t_1, \cdots, t_n)\rangle \Downarrow_c \langle C_n, v\rangle}$$

(Update)

$$\frac{\langle C_{i-1}, t_i\rangle \Downarrow_c \langle C_i, v_i\rangle \quad \mathtt{f}(p_1, \cdots, p_n) \to r \in \mathcal{E} \quad \sigma \in \mathfrak{S} \quad p_i\sigma = v_i \quad \langle C_n, r\sigma\rangle \Downarrow_c \langle C, v\rangle}{\langle C_0, \mathtt{f}(t_1, \cdots, t_n)\rangle \Downarrow_c \langle C \cup (\mathtt{f}, v_1, \cdots, v_n, v), v\rangle}$$

Figure 3: Call-by-value interpreter with Cache of $\langle \mathcal{C}, \mathcal{F}, f, \mathcal{E}\rangle$.

Recalling Equation (1), we can state that the cache of $f(t_1, \ldots, t_n)$ is bounded by

$$P(\max(|t_1|, \ldots, |t_n|)) \times (1 + R \times P_H(Q(\max(|t_1|, \ldots, |t_n|))))$$

which is a polynomial in $\max(|t_1|, \ldots, |t_n|)$, thus completing the induction.

So, it remains to prove the existence of Q as defined above. Recalling the definition of $\rightsquigarrow$, for all $(h, w_1, \ldots, w_k) \in G_{(g,u_1,\ldots,u_m)}$, we have $g(u_1, \ldots, u_m) \to r\sigma = C[h(e_1, \ldots, e_k)]\sigma$ for some context C and some terms e_i with $e_i\sigma \xrightarrow{!} w_i$. Observe that terms in the range of σ have size bounded by $\max(|u_1| \ldots, |u_m|)$, themselves bounded by $\max(|t_1|, \ldots, |t_n|)$. So, employing Lemma 1, we can state that $e_i\sigma$ has size polynomially bounded by $Q_{e_i}(\max(|t_1|, \ldots, |t_n|))$. Since there are only finitely many (there are finitely many rules) such e_i, the proof is done. □

4. COURSE OF VALUE EXTENSION

In this section, we suppose that we have a well-founded partial ordering $<$ on terms which is closed by substitution and context, think of $<$ has $\prec^{\trianglelefteq}_{mpo}$ or $\prec^{\blacktriangleleft}_{mpo}$.[3] Let us suppose that a theory T is defined as the set of program such that for all rules $\ell \to r$, $\ell > r$. The ordering being closed by context and substitution, $u \to v$ implies $u > v$ thus leading to termination.

Definition 10. *The course-of-value (cov) extension of T induced by a relation $>$ is the set of programs such that for all terms $u_1, \ldots, u_n$, $f(u_1, \ldots, u_n) \to_f r$ implies $f(u_1, \ldots, u_n) > r$.*

By induction on the rank of symbols, it is easy to prove that such a property provides termination (one may observe that the property is actually a weak form of termination proof by dependency pairs. A good complexity analysis in the context of depedency pairs has been done by Hirokawa and Moser [11]).

We can presently state our second main theorem:

Theorem 3.

1. *Functions computed by cov-programs with an additive quasi-interpretation and a termination proof by $PPO_{\trianglelefteq}$ are exactly* Ptime *functions.*
2. *Functions computed by cov-programs with an additive quasi-interpretation and a termination proof by $PPO_{\blacktriangleleft}$ are exactly* Pspace *functions.*

The theorem can be interpreted as follows. Even if $PPO_{\trianglelefteq}$ and $PPO_{\blacktriangleleft}$ characterize the same class of functions (namely Ptime), they contain different algorithms. In other words, we gave an indirect proof of an essential difference between the two program sets.

One may observe first that the program of the generalized geography given in Example 6 cannot be proven terminating within $PPO_{\trianglelefteq}$. The issue comes from the rule[4]

$$\begin{aligned}\mathtt{player}(g) \quad \to \quad & \mathtt{if\ snd}(g) = \varepsilon\ \mathtt{then} \\ & \quad \mathtt{if\ fst}(g) = \varepsilon\ \mathtt{then\ ff} \\ & \quad \mathtt{else\ not}(\mathtt{opp}(\mathtt{rem}_1(g))) \\ & \mathtt{else\ not}(\mathtt{and}(\mathtt{opp}(\mathtt{rem}_1(g)), \mathtt{opp}(\mathtt{rem}_2(g)))).\end{aligned}$$

It cannot be ordered by $PPO_{\trianglelefteq}$ even with course-of-value extension. Indeed, $\mathtt{player}(g)$ cannot be compared to $\mathtt{opp}(\mathtt{rem}_1(g))$. But even $[\![\mathtt{rem}_1]\!](g)$ cannot be compared to the term g whatever the constructor term g. Consider for instance the term $\mathtt{rem}_1(\mathbf{cons}(T(V(1(\varepsilon), \varepsilon, \varepsilon)), \mathbf{cons}(V(0(\varepsilon), 1(\varepsilon), \varepsilon), \mathbf{nil})))$. We have

$$[\![\mathtt{rem}_1]\!](\mathbf{cons}(T(V(1(\varepsilon), \varepsilon, \varepsilon)), \mathbf{cons}(V(0(\varepsilon), 1(\varepsilon), \varepsilon), \mathbf{nil}))) = \mathbf{cons}(V(1(\varepsilon), 0(\varepsilon), \varepsilon), \mathbf{cons}(0(\varepsilon), \mathbf{nil}))$$

but

$$\mathbf{cons}(V(1(\varepsilon), 0(\varepsilon), \varepsilon), \mathbf{cons}(0(\varepsilon), \mathbf{nil})) \not\trianglelefteq \mathbf{cons}(T(V(1(\varepsilon), \varepsilon, \varepsilon)), \mathbf{cons}(V(0(\varepsilon), 1(\varepsilon), \varepsilon), \mathbf{nil})).$$

However, it is clear that

$$\mathbf{cons}(V(1(\varepsilon), 0(\varepsilon), \varepsilon), \mathbf{cons}(0(\varepsilon), \mathbf{nil})) \underline{\blacktriangleleft} \mathbf{cons}(T(V(1(\varepsilon), \varepsilon, \varepsilon)), \mathbf{cons}(V(0(\varepsilon), 1(\varepsilon), \varepsilon), \mathbf{nil}))$$

and more generally speaking, it is obvious that $[\![\mathtt{rem}_1]\!](g)$ is embedded in g! Thus, the relation $\to_{\mathtt{player}}$ can be ordered by $PPO_{\underline{\blacktriangleleft}}$. More generally, we will see that the program can be ordered by cov-$PPO_{\underline{\blacktriangleleft}}$.

4.1 Proof of Theorem 3

[3] A reminder for the reader: Remark 1.

[4] Which is typically a course-of-value/parameter substitutions recursion schema.

The end of the section is devoted to the proof of the Theorem.

Since program with an additive quasi-interpretation and a termination proof by $PPO_{\trianglelefteq}$ are part of cov-programs with an additive quasi-interpretation and a termination proof by $PPO_{\trianglelefteq}$, it is clear that these latter program contain at least all PTIME functions. Thus, the first item of the Theorem is a direct corollary of the following proposition:

PROPOSITION 7. *Functions computed by cov-programs with an additive quasi-interpretation and a termination proof by $PPO_{\trianglelefteq}$ can be computed in polynomial time.*

PROOF. The proof goes by induction on the rank of symbols. It is actually an adaptation of the Lemma 44 in [4]. The main issue is to prove that the number of entries in the cache remain polynomial. Consider a rule application $f(p_1, \cdots, p_n)\sigma \to r\sigma$ with a constructor substitution σ. Let us consider g of rank of f such that $r = C[g(q_1, \ldots, q_n)]$. According to the ordering, $q_i\sigma \xrightarrow{!} v$ implies that $v \prec^{\trianglelefteq}_{mpo} p_i\sigma$. Thus, recursive calls corresponding to symbols of rank of f are done on terms which are subterms of the input terms.

At that point, we essentially recovered the properties we had for programs with a standard proof of termination by $PPO_{\trianglelefteq}$. We recall that there are at most $\sum_{i=1,\ldots,n} |t_i|$ constructor terms that are subterms of $t_1, \ldots, t_n$. Thus, there are at most $(\sum_{i=1,\ldots,n} |t_i|)^d$ entries in the cache where d is the maximal arity of symbols. We define $S(x) = (d \times x)^d$. One may observe that there are at most $S(\max(|t_1|, \ldots, |t_n|))$ entries in $R_{(f,t_1,\ldots,t_n)}$ with the notation R as in the section above.

Function calls corresponding to symbols of lower ranks can be treated as in [4]. Due to the subterm property, we can state that $f(t_1, \ldots, t_n) \xrightarrow{*} C[g(u_1, \ldots, u_m)]$ for some constructor terms $t_1, \ldots, t_n, u_1, \ldots, u_m$ implies the inequality $(\!|g(u_1, \ldots, u_m)|\!) \leq (\!|f(t_1, \ldots, t_n)|\!)$. Then, $|g(u_1, \ldots, u_m)| \leq (\!|f|\!)(t_1, \ldots, t_n) \leq P_f(\max(|t_1|, \ldots, |t_n|))$ where the latter inequality comes from Proposition 3. Thus, inputs of subcalls have polynomial size with respect to the size of the inputs, say $Q(x)$.

To end the induction, suppose that the function calls of symbols of lower rank needs a cache of size $P(x)$ where x is the size of the input. Then, each of the calls fired by symbols of rank of f can be performed in $P(Q(x))$ which is a polynomial. To sum up, the tree has size $S(x) \times P(Q(x))$ where x denotes the size of the input.

□

For the second statement of the Theorem 3, we define $\mathcal{T}$ to the be set of programs with a termination proof by $PPO_{\trianglelefteq}$ having a quasi-interpretation. The easiest part is to prove the completeness with respect to PSPACE. To justify this, let us observe that the program of the generalized geography given in Example 6 is actually a cov-program with an additive quasi-interpretation and a termination proof by $PPO_{\trianglelefteq}$. Indeed, it has a quasi-interpretation. And clearly, the fact that $[\![\texttt{rem}_1]\!](g) \trianglelefteq g$ shows that the program is catched by $PPO_{\trianglelefteq}$. Set the precedence to be the rank relation on symbols.

From this completeness result, the fact that PTIME functions can be computed in $\mathcal{T}$, that $\mathcal{T}$ is closed by composition, it is tedious, but not difficult to prove that all PSPACE predicates can be computed by cov-program with an additive quasi-interpretation and a termination proof by $PPO_{\trianglelefteq}$.

In the other direction, the proof is done by induction on the rank of symbols. The principle is to show that the height of the call tree is bounded by a polynomial. Thus, the evaluation can be performed by a Parallel Random Access Machine (PRAM) running in polynomial time. Then, following Chandra, Kozen and Stockmeyer [6], we can state the the computation can be done in polynomial space.

Consider, within some computation, a rule application $f(p_1, \cdots, p_n)\sigma \to C[g(u_1, \ldots, u_m)]\sigma$ with g of rank of f and σ a constructor substitution. Consider the normal forms $u_i\sigma \xrightarrow{!} v_i$. By definition of the ordering, for all $i \leq n$, $v_i \trianglelefteq p_i\sigma$ and for one of them $v_j \triangleleft p_i\sigma$. Consequently, the size of one of the inputs is decreased by at least one, the other being not bigger. Consequently, the depth of the function calls of rank of f is bounded linearly by the size of the input.

We can actually go back to the proof of Theorem 2 and do exactly the same induction on symbol's rank, but replacing the size of the call tree by its depth. For symbols of lower rank, we use the trick above: these calls are polynomial and involve arguments of polynomial size. It is then routine to verify that globally the call tree has polynomial depth.

5. CONCLUSION

At first sight, we could have concluded that termination proofs by sub-term or embedding correspond to the same set of algorithms. This can be justified by Theorems 1 and 2. But, as Theorem 3 shows it, this is only a partial view on sets of programs. One knows that there are several ways to compute functions, by a lot of programs. The skill of programmers is to find the right one. Thus, it is important that tools performing resource analysis are as wide as possible. We have shown that a tool based on embedding is more powerful than a tool based on the subterm relation. It would be nice to draw a general map of termination tools with such methods. To discriminate two sets of programs, we shifted the problem from a direct comparison of two programs to an indirect one. It seems that the latter is much easier in practice.

6. REFERENCES

[1] Nils Andersen and Neil D. Jones. Generalizing cook's transformation to imperative stack programs. In *Proceedings of the Colloquium in Honor of Arto Salomaa on Results and Trends in Theoretical Computer Science*, pages 1–18, London, UK, 1994. Springer-Verlag.

[2] Andrea Asperti. The intensional content of rice's theorem. In George C. Necula and Philip Wadler, editors, *POPL*, pages 113–119. ACM, january 2008.

[3] Guillaume Bonfante. Observation of implicit complexity by non confluence. In *DICE*, 2010.

[4] Guillaume Bonfante, Jean-Yves Marion, and Jean-Yves Moyen. Quasi-interpretation: a way to control ressources. *Theoretical Computer Science*, 412(25):2776–2796, 2011. accepted for publication.

[5] Guillaume Bonfante, Jean-Yves Marion, and Romain Péchoux. Synthesis of quasi-interpretations. In *Logic and Complexity in Computer Science*, 2005.

[6] A. K. Chandra, D. J. Kožen, and L. J. Stockmeyer. Alternation. *Journal ACM*, 28:114–133, 1981.

[7] Stephen Cook. Characterizations of pushdown machines in terms of time-bounded computers. *Journal of the ACM*, 18(1):4–18, January 1971.

[8] S. Coupet-Grimal and W. Delobel. An effective proof of the well-foundedness of the multiset path ordering. Rapport de recherche, LIF, 2005.

[9] Nachum Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17(3):279–301, 1982.

[10] Nachum Dershowitz and Jean-Pierre Jouannaud. *Handbook of Theoretical Computer Science vol.B*, chapter Rewrite systems, pages 243–320. 1990.

[11] Nao Hirokawa and Georg Moser. Automated complexity analysis based on the dependency pair method. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *IJCAR*, volume 5195 of *Lecture Notes in Computer Science*, pages 364–379. Springer, 2008.

[12] Neil Jones. Logspace and ptime characterized by programming languages. *Theroretical Computer Science*, 228:151–174, 1999.

[13] Samuel Kamin and Jean-Jacques Lévy. Attempts for generalising the recursive path orderings. Technical report, Univerity of Illinois, Urbana, 1980. Unpublished note. Accessible on http://perso.ens-lyon.fr/pierre.lescanne/not_accessible.html.

[14] Daniel Leivant and Jean-Yves Marion. Ramified recurrence and computational complexity II: substitution and poly-space. In L. Pacholski and J. Tiuryn, editors, *Computer Science Logic, 8th Workshop, CSL '94*, volume 933 of *Lecture Notes in Computer Science*, pages 486–500, Kazimierz,Poland, 1995. Springer.

[15] Rósza Péter. *Rekursive Funktionen*. Akadémiai Kiadó, Budapest, 1966. English translation: *Recursive Functions*, Academic Press, New York, 1967.

[16] H.E. Rose. *Subrecursion*. Oxford university press, 1984.