Under consideration for publication in Math. Struct. in Comp. Science

On Quasi-Interpretations, Blind Abstractions, and Implicit Complexity†

PATRICK BAILLOT[1]

UGO DAL LAGO[2]

JEAN-YVES MOYEN[3]

[1] *LIP, UMR 5668 CNRS, ENS-Lyon, UCBL, INRIA, Université de Lyon.*
[2] *Università di Bologna.*
[3] *LIPN, UMR 7030 CNRS, Université Paris 13.*

Received 23 July 2008; Revised 23 September 2011

Quasi-interpretations are a technique to guarantee complexity bounds on first-order functional programs: with termination orderings they give in particular a sufficient condition for a program to be executable in polynomial time (Marion & Moyen 2000), called here the P-criterion. We study properties of the programs satisfying the P-criterion, in order to improve the understanding of its intensional expressive power. Given a program, its blind abstraction is the non-deterministic program obtained by replacing all constructors with the same arity by a single one. A program is blindly polytime if its blind abstraction terminates in polynomial time. We show that all programs satisfying a variant of the P-criterion are in fact blindly polytime. Then we give two extensions of the P-criterion: one by relaxing the termination ordering condition, and the other one (the bounded value property) giving a necessary and sufficient condition for a program to be polynomial time executable, with memoisation.

1. Introduction

Implicit computational complexity (ICC) explores machine-free characterisations of complexity classes, without referring to explicit resource bounds but instead by seeing these bounds as consequences of restrictions on program structures. It has been mainly developed in the functional programming paradigm, by taking advantage of ideas from primitive recursion (Bellantoni & Cook 1992, Leivant 1994), proof-theory and linear logic (Girard 1998), rewriting systems or functional programming (Jones 1999, Bonfante, Cichon, Marion & Touzet 2001, Bonfante, Marion & Moyen 2001, Marion 2003, Bonfante, Marion & Moyen 2011), type systems (Leivant & Marion 1993, Hofmann 1999), etc.

Usually, ICC results include both a soundness and a completeness statement: the first

† Work partially supported by projects CRISS (ACI), NO-CoST (ANR, JC05_43380).

one says that all programs of a given language (or those satisfying a criterion) satisfy a certain quantitative property, the latter one says that all *functions* (or *problems*) of the corresponding functional complexity class can be programmed in this language. For instance in the case of polynomial time complexity the first statement refers to the possibility of evaluating programs in polynomial time, whereas the second one, which is of an *extensional* nature, refers to the class FP of functions computable in polynomial time (or the class P of problems solvable in polynomial time). Theorems of this kind have been given for many systems, like for instance ramified recursion (Bellantoni & Cook 1992, Leivant 1994), variants of linear logic (Girard 1998), fragments of functional languages (Jones 1999), etc. Our main aim here is *not* defining another characterisation of polynomial time computable functions. Rather, we would like to start studying *existing* characterisations in order to better understand their properties, intensional expressivity *in primis*.

Expressivity. One of the main motivations behind ICC comes from programming language theory, because ICC suggests ways to control complexity properties of programs, which is a difficult issue because of its infinite nature. However, extensional correspondence with complexity classes is usually not enough: a programming language (or a static analysis methodology for it) offering guarantees in terms of program safety is plausible only if it captures enough interesting and natural *algorithms*. In turn, this cannot be guaranteed by merely requiring the system to extensionally correspond to a complexity class. This issue has been pointed out by several authors (Marion & Moyen 2000, Hofmann 1999, Marion 2003) and advances have been made in the direction of more liberal ICC systems: some examples are type systems for *non-size-increasing* computation (Hofmann 1999) and quasi-interpretations (Bonfante, Marion & Moyen 2001, Bonfante et al. 2011). However it is not always easy to measure the improvement a new ICC system brings up: this is usually illustrated by providing examples. Note that some experiments (Avanzini & Moser 2008) have been done to compare some rewriting-based criteria on data bases of examples from term rewriting.

We think that to compare in a more appropriate way the algorithmic aspects of ICC systems and in particular to understand their limitations, specific analytic methods should be developed. Indeed, what we would need are *sharp results* on the *intensional* expressive power of existing systems and on the intrinsic limits of implicit complexity as a way to isolate large (but decidable) classes of programs with bounded complexity. For that we aim to establish properties (like necessary conditions) of the *programs* captured by an ICC system. Such a characterisation is provided in (Dal Lago 2007), where Cobham's definitions by bounded recursion on notation are shown to exactly correspond to hereditarily polytime primitive recursive programs. In this work we undertake an analogous study, but for an ICC system closer to programming practice. More specifically, we give a sufficient condition that all programs in some existing ICC systems must satisfy.

Quasi-interpretations (QI) can be considered as a static analysis methodology for inferring asymptotic resource bounds for first-order functional programs written as constructor term rewriting systems. Used with termination orderings they allow to define various criteria to guarantee either space or time complexity bounds (Bonfante, Marion & Moyen 2005, Amadio 2005, Bonfante, Marion & Péchoux 2007, Bonfante et al. 2011).

They present several advantages: the language for which they are defined is simple to use, and more importantly the class of programs captured by this approach is large compared to that of other ICC systems. For instance, this class contains all primitive recursive programs from Bellantoni and Cook's function algebra (Moyen 2003), but also general recursive programs (non primitive recursive). One of the proposed criteria, that we will call here the *P-criterion*, says that programs with certain QIs and recursive path orderings can be evaluated in polynomial time (Bonfante et al. 2011). A key point when proving that programs satisfying the P-criterion can be evaluated in polynomial time is the adoption of a caching mechanism, following (Marion 2003).

In this paper, we focus our attention on QIs, and prove a strong necessary condition for first-order functional programs to have a (uniform) QI. More precisely, a program transformation called *blind abstraction* is presented. It consists in collapsing the constructors of a given arity to just one constructor, modifying rewriting rules accordingly. This produces in general non-confluent programs, the efficiency of which can be measured by considering all possible evaluations of the program.

In general, the blind abstraction of a polytime first-order functional program is not itself polytime: blinding introduces many execution paths that are not available in the original program. However, we show that under certain assumptions, blinding a program satisfying the P-criterion with a (uniform) QI always produces a polytime program, independently from non-confluence.

We cannot claim the obtained result to be breakthrough. However, this is one of the very first attempts to delineate the class of programs captured by a mainstream ICC system. Indeed, it implies that any polynomial time program whose blind abstraction is *not* polytime *cannot* have a quasi-interpretation. As we will show in Section 4, there are many natural programs in this class.

A preliminary version of this work was presented at the Eighth International Workshop on Logic and Computational Complexity, in 2006.

Outline. We first describe the syntax and operational semantics of programs (Section 2), before termination orderings and QIs (Section 3). Then blind abstractions are introduced (Section 4) and we give the main property of the P-criterion (w.r.t. blinding) in Section 5, with applications to safe recursion. Finally we define a generalisation of the previous termination ordering and of QIs (bounded values property) which also guarantees polytime soundness (Section 7).

2. Programs as Term Rewriting Systems

In Section 2 and Section 3 we will recall some Definitions and results from (Bonfante et al. 2011), and adapt them to non-deterministic programs.

Let us stress that we are here actually interested in the study of deterministic programs. However as soon as we will consider abstractions of programs we will obtain non-deterministic rewriting systems, and to analyse them we thus need our definitions and results to handle non-deterministic systems.

2.1. *Syntax and Semantics of Programs*

We consider first-order term rewriting systems (TRS) with disjoint sets $\mathcal{X}$, $\mathcal{F}$, $\mathcal{C}$ resp. of variables, function symbols and constructor symbols.

Definition 1 (Syntax). The sets of terms and the equations are defined by:

$$\begin{array}{lrcl}
(\textit{constructor terms}) & \mathcal{T}(\mathcal{C}) \ni v & ::= & \mathbf{c}(v_1, \cdots, v_n) \\
(\textit{terms}) & \mathcal{T}(\mathcal{C}, \mathcal{F}, \mathcal{X}) \ni t & ::= & x \mid \mathbf{c}(t_1, \cdots, t_n) \mid \mathtt{f}(t_1, \cdots, t_n) \\
(\textit{patterns}) & \mathcal{P} \ni p & ::= & x \mid \mathbf{c}(p_1, \cdots, p_n) \\
(\textit{equations}) & \mathcal{D} \ni d & ::= & \mathtt{f}(p_1, \cdots, p_n) \to t
\end{array}$$

where $x \in \mathcal{X}$, $\mathtt{f} \in \mathcal{F}$, and $\mathbf{c} \in \mathcal{C}$. We shall use a type writer font for function symbols and a bold face font for constructors.

Definition 2 (Programs). A program is a tuple $\mathtt{f} = \langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E} \rangle$ where $\mathcal{E}$ is a set of equations in $\mathcal{D}$. Each variable in the right hand side (rhs) of an equation also appears in the left hand side (lhs) of the same equation. The program has a main function symbol in $\mathcal{F}$, which we shall also call $\mathtt{f}$.

Our programs, in other words, are constructor (term-rewriting) systems (Klop & de Vrijer 2003).

The domain of computation is the constructor algebra $\mathcal{T}(\mathcal{C})$. A substitution σ is a mapping from variables to terms. We note $\mathfrak{S}$ the set of *constructor substitutions*, *i.e.* substitutions σ with range $\mathcal{T}(\mathcal{C})$. Our programs are not necessarily deterministic, that is to say the TRS is not necessarily confluent. Non-confluent programs will here be interpreted by relations.

Firstly, we consider a call by value semantics which is displayed in Figure 1. The meaning of $t \downarrow v$ is that t evaluates to the constructor term v. A derivation π of the judgement $t \downarrow v$ will be called a *reduction proof.* The program $\mathtt{f}$ computes a relation $[\![\mathtt{f}]\!] \subseteq \mathcal{T}(\mathcal{C})^n \times \mathcal{T}(\mathcal{C})$ defined by: for all $u_i \in \mathcal{T}(\mathcal{C}), v \in [\![\mathtt{f}]\!](u_1, \cdots, u_n)$ iff there is a derivation for $\mathtt{f}(u_1, \cdots, u_n) \downarrow v$. The size $|J|$ of a judgement $J = t \downarrow v$ is the sum $|t| + |v|$ of the two terms composing the judgement. The size $|\pi|$ of any reduction proof π is the sum of $|J|$ over all the occurrences of judgements J in π. In deterministic programs, the size $|\pi|$ of $\pi : t \downarrow v$ can be safely considered as the *cost* of computing v from t.

$$\frac{\mathbf{c} \in \mathcal{C} \quad t_i \downarrow v_i}{\mathbf{c}(t_1, \cdots, t_n) \downarrow \mathbf{c}(v_1, \cdots, v_n)}\ (\text{Constructor}) \qquad \frac{\exists j, t_j \notin \mathcal{T}(\mathcal{C}) \quad t_i \downarrow v_i \quad \mathtt{f}(v_1, \cdots, v_n) \downarrow v}{\mathtt{f}(t_1, \cdots, t_n) \downarrow v}\ (\text{Split})$$

$$\frac{\mathtt{f}(p_1, \cdots, p_n) \to r \in \mathcal{E} \quad \sigma \in \mathfrak{S} \quad p_i\sigma = v_i \quad r\sigma \downarrow v}{\mathtt{f}(v_1, \cdots, v_n) \downarrow v}\ (\text{Function})$$

Fig. 1. Call by value semantics of a program $\langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E} \rangle$

The following is an example of program that we will use throughout the paper ($i \in \{0,1\}$):

$$\begin{array}{ll}
\mathtt{f}(\mathbf{s}_0\mathbf{s}_i x) & \to \mathtt{append}(\mathtt{f}(\mathbf{s}_1 x), \mathtt{f}(\mathbf{s}_1 x)) \\
\mathtt{f}(\mathbf{s}_1 x) & \to x \\
\mathtt{f}(\mathbf{nil}) & \to \mathbf{nil} \\
\mathtt{append}(\mathbf{s}_i x, y) & \to \mathbf{s}_i \mathtt{append}(x, y) \\
\mathtt{append}(\mathbf{nil}, y) & \to y
\end{array}$$

Figure 2 reports a derivation π.

$$\dfrac{\dfrac{\dfrac{\mathbf{nil} \downarrow \mathbf{nil}}{\mathtt{f}(\mathbf{s}_1\mathbf{nil}) \downarrow \mathbf{nil}} \quad \dfrac{\mathbf{nil} \downarrow \mathbf{nil}}{\mathtt{f}(\mathbf{s}_1\mathbf{nil}) \downarrow \mathbf{nil}} \quad \dfrac{\mathbf{nil} \downarrow \mathbf{nil}}{\mathtt{append}(\mathbf{nil}, \mathbf{nil}) \downarrow \mathbf{nil}}}{\mathtt{append}(\mathtt{f}(\mathbf{s}_1\mathbf{nil}), \mathtt{f}(\mathbf{s}_1\mathbf{nil})) \downarrow \mathbf{nil}}}{\mathtt{f}(\mathbf{s}_0\mathbf{s}_1\mathbf{nil}) \downarrow \mathbf{nil}}$$

Fig. 2. Example of reduction proof.

Semantic rules in Figure 1 can be either active or passive:

Definition 3 (Classifying Terms, Rules and Judgements). The *passive* semantic rules are (Constructor) and (Split). The only *active* rule is (Function). Let $\pi : t \downarrow v$ be a reduction proof. If we have:

$$\frac{e = \mathtt{g}(q_1, \cdots, q_n) \to r \quad \sigma \in \mathfrak{S} \quad q_i\sigma = u_i \quad r\sigma \downarrow u}{s = \mathtt{g}(u_1, \cdots, u_n) \downarrow u}$$

then we say that term s (resp. judgement $J = s \downarrow u$) is *active* in π. e is the equation *activated* by s (resp. J) in π and $r\sigma$ (resp. $r\sigma \downarrow u$) is the *activation* of s (resp. J) in π. Other judgements (conclusions of (Split) or (Constructor) rules) are called *passive*.

Notice that the set of active terms in a derivation π is exactly the set of terms of the form $\mathtt{f}(v_1, \cdots, v_n)$, where v_i are constructor terms, appearing in π. Since the program may be non deterministic, the equation activated by a term s depends on the reduction and on the occurrence of s in π, and not only on s.

Our aim now is proving that the size $|\pi|$ of any reduction proof $\mathtt{f}(v_1, \cdots, v_n) \downarrow u$ does not really depend on passive judgements in π. This will somehow motivate our study of the combinatorics of reduction by way of *call-trees*, which we shall define later.

Proposition 1. For all programs, there exists a polynomial $p : \mathbb{N}^2 \to \mathbb{N}$ such that for every derivation proof $\pi : \mathtt{f}(v_1, \cdots, v_n) \downarrow u$, if A is the number of active judgements in π and S is the maximum size of any active judgement in π, then $|\pi| \leq p(A, S)$.

Proof. First of all, observe that for each program, there exists a polynomial $r : \mathbb{N} \to \mathbb{N}$ such that for any reduction proof π, for any active term t in it and for any activation s of t, $|s| \leq r(|t|)$. That is, the size of s is polynomially bounded by the size of t. This holds because any program consists of a *finite* set of equations, each of them leading to at most a polynomial increase in size. Now, consider any occurrence of a passive judgement H inside a reduction proof π. Clearly, in the path from H to the root of π, there is at least

an active judgement (by hypothesis, $\pi : \mathtt{f}(v_1, \cdots, v_n) \downarrow u$, so the conclusion of π is an active judgement itself): take the one which is immediately below H. This way we can associate to any active judgement J in π a set of passive judgements D_J such that any passive judgement is in some D_J. Now, let us show that in D_J there are at most $r(S)$ judgement occurrences. Consider the (Function) rule with J as conclusion. The premise H of this (Function) rule is simply an activation of its conclusion, so its size is bounded by $r(S)$. Moreover the lhs of any passive judgements in D_J is a subterm of the lhs of H, therefore there are at most $r(S)$ such passive judgements. This implies that the number of rule instances in the reduction proof π is at most

$$A + A \cdot r(S).$$

But what about the size of passive judgements in π? The lhs of any passive judgement, has size at most $r(S)$. Indeed, any passive judgement in D_J can be written as $t \downarrow q$, where t is a subterm of an activation of the lhs of J. Now, notice that if q is the rhs of any passive judgement in π, then $|q| \leq \max\{S, |u|\}$, where u is the rhs of the conclusion of π (this by induction on π). As a consequence, $|q| \leq S$, because the rhs of the conclusion of π is part of of an active judgement itself. Putting everything together, we get

$$|\pi| \leq (A + A \cdot r(S))(r(S) + S)$$

so $p(x, y)$ is simply $(x + xr(y))(r(y) + 2y)$. This completes the proof. □

Next, we also consider a call by value semantics with memoisation for confluent programs, following (Marion & Moyen 2000). The idea is to maintain a cache C to avoid recomputing the same results several times. Each time a function call is performed, the semantics looks in the cache C . If the same call has already been computed, then the result can be given immediately, otherwise we need to compute the corresponding value and store the result in the cache (for later reuse). The memoisation semantics is displayed in Figure 3. The (Update) rule can only be triggered if the (Read) rule cannot, that is if there is no v such that $(\mathtt{f}, v_1, \cdots, v_n, v) \in C$. In other words, memoisation corresponds to an automation of the algorithmic technique of dynamic programming. The expression $\langle C, t\rangle \Downarrow \langle D, v\rangle$ means that the result of reducing t is v, given a program $\mathtt{f}$ and an initial cache C. The final cache D contains C and each call which has been necessary to complete the computation. The size $|J|$ of a judgement $\langle C, t\rangle \Downarrow \langle D, v\rangle$ is now the sum $|t| + |v|$. The size $|\pi|$ of any reduction proof $\pi : \langle C, t\rangle \Downarrow \langle D, v\rangle$ is the sum of $|J|$ over all the occurrences of judgements J in π.

Figure 4 depicts a reduction proof π of our example program in the memoisation semantics, where we used the following abbreviations:

$$\begin{aligned}
C &= \{(\mathtt{f}(\mathbf{s}_1\mathbf{nil}), \mathbf{nil})\} \\
D &= \{(\mathtt{f}(\mathbf{s}_1\mathbf{nil}), \mathbf{nil}), (\mathtt{append}(\mathbf{nil}, \mathbf{nil}), \mathbf{nil})\} \\
E &= \{(\mathtt{f}(\mathbf{s}_1\mathbf{nil}), \mathbf{nil}), (\mathtt{append}(\mathbf{nil}, \mathbf{nil}), \mathbf{nil}), (\mathtt{f}(\mathbf{s}_0\mathbf{s}_1\mathbf{nil}), \mathbf{nil})\}
\end{aligned}$$

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

$$\frac{\exists j, t_j \notin \mathcal{T}(\mathcal{C}) \quad \langle C_{i-1}, t_i \rangle \Downarrow \langle C_i, v_i \rangle \quad \langle C_n, \mathtt{f}(v_1, \cdots, v_n) \rangle \Downarrow \langle C, v \rangle}{\langle C_0, \mathtt{f}(t_1, \cdots, t_n) \rangle \Downarrow \langle C, v \rangle} \text{(Split)}$$

$$\frac{(\mathtt{f}, v_1, \cdots, v_n, v) \in C}{\langle C, \mathtt{f}(v_1, \cdots, v_n) \rangle \Downarrow \langle C, v \rangle} \text{(Read)}$$

$$\frac{\mathtt{f}(p_1, \cdots, p_n) \to r \in \mathcal{E} \quad \sigma \in \mathfrak{S} \quad p_i\sigma = v_i \quad \langle C, r\sigma \rangle \Downarrow \langle D, v \rangle}{\langle C, \mathtt{f}(v_1, \cdots, v_n) \rangle \Downarrow \langle D \cup \{(\mathtt{f}, v_1, \cdots, v_n, v)\}, v \rangle} \text{(Update)}$$

Fig. 3. Memoisation semantics of a program $\langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E} \rangle$.

$$\dfrac{\dfrac{\dfrac{\langle \emptyset, \mathbf{nil} \rangle \Downarrow \langle \emptyset, \mathbf{nil} \rangle}{\langle \emptyset, \mathtt{f}(\mathbf{s}_1\mathbf{nil}) \Downarrow \langle C, \mathbf{nil} \rangle} \quad \dfrac{}{\langle C, \mathtt{f}(\mathbf{s}_1\mathbf{nil}) \rangle \Downarrow \langle C, \mathbf{nil} \rangle} \quad \dfrac{\langle C, \mathbf{nil} \rangle \Downarrow \langle C, \mathbf{nil} \rangle}{\langle C, \mathtt{append}(\mathbf{nil}, \mathbf{nil}) \rangle \Downarrow \langle D, \mathbf{nil} \rangle}}{\langle \emptyset, \mathtt{append}(\mathtt{f}(\mathbf{s}_1\mathbf{nil}), \mathtt{f}(\mathbf{s}_1\mathbf{nil})) \rangle \Downarrow \langle D, \mathbf{nil} \rangle}}{\langle \emptyset, \mathtt{f}(\mathbf{s}_0\mathbf{s}_1\mathbf{nil}) \rangle \Downarrow \langle E, \mathbf{nil} \rangle}$$

Fig. 4. Example of reduction proof.

We here need to classify rules, terms and judgements as in the memoisation semantics:

Definition 4 (Classifying Terms, Rules and Judgements). Constructor and Split are *passive* semantic rules. Update is an *active* rule. Read is a *semi-active* rule. Active terms and judgements, activated equations, activations and dependencies are defined similarly as for the call by value case. Semi-active terms and judgements are similarly defined from semi-active rules.

As in the call-by-value semantics, we can concentrate our attention on active judgements when reasoning about the size of π of a reduction:

Proposition 2. Consider the memoisation semantics of Figure 3. For all programs, there exists a polynomial p such that for all derivation proof $\pi : \langle \emptyset, \mathtt{f}(v_1, \cdots, v_n) \rangle \Downarrow \langle C, u \rangle$, if A is the number of distinct active judgements in π and S is the maximum size of an active judgement in π, then $|\pi| \leq p(A, S)$.

Proof. First of all, notice that the conclusions of active rule instances in π are distinct from each other. So, A is indeed the number of active rule instances in π. The total number of active and passive rule instances can be bound as in Proposition 1 by

$$A + A \cdot r(S).$$

Since semi-active judgements form a subset of the set of leaves in π and since the number of premises of a rule is statically bounded (by $(k+1)$, where k is the maximum arity of a

symbol), the number of semi-active judgements is polynomially bounded by the number of non-leaves judgements in π.
Bounding the size of any judgement in π requires a little more care than in Proposition 1. We can proceed as follows ; consider a sub-derivation $\rho : \langle D, t\rangle \Downarrow \langle E, q\rangle$ in π:
— The size $|t|$ of t is at most $r(S)$.
— The cardinals of D and E are bounded by A.
— The size of elements of D and E is bounded by S
— The size $|q|$ of q is at most S.
This concludes the proof. □

Lemma 3. Consider the memoisation semantics of Figure 3. Let $J = \langle D, t\rangle \Downarrow \langle E, v\rangle$ be a semi-active judgement in a proof $\pi : \langle \emptyset, t\rangle \Downarrow \langle C, v\rangle$, then there exists an active judgement $H = \langle F, t\rangle \Downarrow \langle G, v\rangle$ in π. We say that H is the active judgement corresponding to J.

Proof. Because the pair can only be in the cache if an active judgement put it there. □

There is no obvious way to use memoisation with non-confluent programs. Indeed, the same function call can lead to several different results. Several ideas could be used to define a memoisation semantics for non-confluent programs, but they all have their problems, hence we will not use any of them here and only use memoisation when the program is confluent. For sufficient conditions to check if a program is confluent, one can for instance refer to Huet's work (Huet 1980).

2.2. *Call Trees, Call Dags*

Following (Bonfante et al. 2011), we now present call-trees which are a tool that we shall use all along. Note that this is not a new kind of semantics for programs but an analysis tool, used to study both call by value semantics and memoisation semantics of execution. A call-tree indeed provides a static view of an execution and describes all function calls. Hence, we can study dependencies between function calls without taking care of the extra details provided by the underlying rewriting relation.

Definition 5 (States). A *state* is a tuple $(\mathtt{f}, v_1, \cdots, v_n, v)$ where $\mathtt{f}$ is a function symbol of arity n, $v_1, \cdots, v_n$ are constructor terms and $v \in [\![\mathtt{f}]\!](v_1, \cdots, v_n)$. Assume that $\eta = (\mathtt{f}, v_1, \cdots, v_n, v)$ and $\mu = (\mathtt{g}, u_1, \cdots, u_n, u)$ are two states. A *transition* is a triplet $\eta \overset{e}{\rightsquigarrow} \mu$ such that:
1. e is an equation $\mathtt{f}(p_1, \cdots, p_n) \to t$ of $\mathcal{E}$,
2. there is a substitution σ such that $p_i\sigma = v_i$ for all $1 \le i \le n$,
3. there is a subterm $\mathtt{g}(s_1, \cdots, s_m)$ of t such that $s_i\sigma \downarrow u_i$ for all $1 \le i \le m$.

$\overset{*}{\rightsquigarrow}$ is the reflexive transitive closure of $\cup_{e\in\mathcal{E}} \overset{e}{\rightsquigarrow}$.

The above definition of state is slightly different although essentially equivalent to the one from (Bonfante et al. 2011).

Definition 6 (Call Trees). Consider the call by value semantics. Let $\pi : t \downarrow v$ be a reduction proof. Its set of *call trees* is the set of trees Θ_π obtained by only keeping active terms in π.

That is, if t is passive:

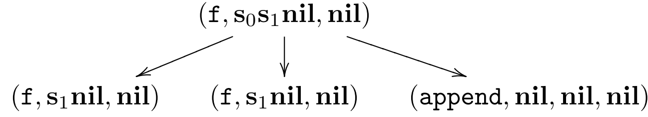
$$\frac{b \in \mathcal{F} \bigcup \mathcal{C} \quad \pi_i : t_i \downarrow v_i}{b(t_1, \cdots, t_n) \downarrow b(v_1, \cdots, v_n)} \text{(Constructor) or (Split)},$$

then $\Theta_\pi = \bigcup \Theta_{\pi_i}$. If t is active:

$$\frac{\mathtt{f}(p_1, \cdots, p_n) \to r \in \mathcal{E} \quad \sigma \in \mathfrak{S} \quad p_i\sigma = v_i \quad \rho : r\sigma \downarrow v}{\mathtt{f}(v_1, \cdots, v_n) \downarrow v} \text{(Function)},$$

then Θ_π only contains the tree whose root is $(\mathtt{f}, v_1, \cdots, v_n, v)$ and children are Θ_ρ.

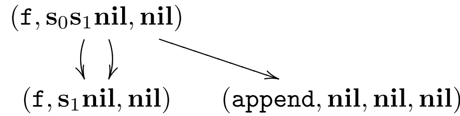
The following is the only element of Θ_π, where π is the reduction proof in Figure 2:



When using the memoisation semantics, as some function calls are not recomputed but instead fetched from the cache, it is convenient to represent the dependencies of function calls by a directed acyclic graph (dag) rather than by a tree.

Definition 7 (Call Dags). Consider the memoisation semantics. Let $\pi : \langle C, t\rangle \Downarrow \langle D, v\rangle$ be a reduction proof. Its set of *call dags* Θ_π is defined as in Def. 6 by keeping active terms, and by replacing each semi-active term by a link to the corresponding active term, according to Lemma 3.

For instance, the following dag is the only element of Θ_π, where π is the reduction proof in Figure 4:



Fact 1 (Call Tree Arity). Let $\mathtt{f}$ be a program and consider the call by value semantics. There exists a fixed integer k, such that given a derivation π of a term of the program, and a tree $\mathcal{T}$ of Θ_π, all nodes in $\mathcal{T}$ have at most k children.

Proof. For each rhs term r of an equation of $\mathtt{f}$ consider the number of maximal subterms of r with a function as head symbol; let then k be the maximum of these integers over the (finite) set of equations of $\mathtt{f}$. □

Lemma 4. Consider the call by value semantics. Let π be a reduction proof, $\mathcal{T}$ be a call-tree in Θ_π and consider two states $\eta = (\mathtt{f}, v_1, \cdots, v_n, v)$ and $\mu = (\mathtt{g}, u_1, \cdots, u_n, u)$ such that μ is a child of η. Then there is an equation e such that $t = \mathtt{f}(v_1, \cdots, v_n)$ activates e in π and $\eta \overset{e}{\leadsto} \mu$. Conversely, if $\eta \overset{e}{\leadsto} \mu$ then μ is a child of η in $\mathcal{T}$.

Proof. A trivial induction on the structure of π. □

This implies that in the deterministic case our definition of call trees is equivalent to the one in (Bonfante et al. 2011). However, we did need a new definition in order to deal with non determinism.

Recall that the size $|\pi|$ of a reduction proof π (either in the call by value or the memoisation semantics) takes into account all the symbol occurrences in π: this way, it can be considered as a reasonable approximation on execution time. As a consequence, bounding execution time in a computation boils down to (i) bound the size (number of nodes) in the call tree or call dag and (ii) bound the size of the states appearing in the call tree (dag):

Proposition 5. For any program $\mathtt{f}$:

1. (Call by value semantics) There is a polynomial $p_{\mathtt{f}} : \mathbb{N}^2 \to \mathbb{N}$ such that whenever $\pi : \mathtt{f}(u_1, \ldots, u_n) \downarrow v$ and the set Θ_π of call trees contains $a_\pi \in \mathbb{N}$ states whose size is at most $s_\pi \in \mathbb{N}$, then $|\pi| \leq p_{\mathtt{f}}(a_\pi, s_\pi)$.
2. (Memoisation semantics) Assume $\mathtt{f}$ is deterministic. There is a polynomial $q_{\mathtt{f}} : \mathbb{N}^2 \to \mathbb{N}$ such that whenever $\pi : \langle \emptyset, \mathtt{f}(u_1, \ldots, u_n)\rangle \Downarrow \langle C, v\rangle$ and the set Θ_π of call dags contains $a_\pi \in \mathbb{N}$ states whose size is at most $s_\pi \in \mathbb{N}$, then $|\pi| \leq q_{\mathtt{f}}(a_\pi, s_\pi)$.

Proof. This is an easy corollary of Proposition 1 and Proposition 2. □

The naive model where each rule takes unary time to be executed is not very realistic with the memoisation semantics. Indeed, each (Read) and (Update) rule needs to perform a lookup in the cache and this would in practice take time proportional to the size of the cache (and the size of elements in it). However, the size of the final cache is exactly the number of (Update) rules in the proof (because only (Update) modifies the cache) and the size of terms in the cache is bounded by the size of active terms (only active terms are stored in the cache). So Proposition 5 yields to a polynomial bound on the execution time.

3. Orderings, Quasi-Interpretations

This section presents in a novel way concepts and results which have already appeared in the literature (Bonfante et al. 2011).

3.1. *Termination Orderings*

Definition 8 (Precedence). Let $\mathtt{f} = \langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E}\rangle$ be a program. A *precedence* $\preceq_{\mathcal{F}}$ is a preorder over $\mathcal{F} \bigcup \mathcal{C}$. We note $\approx_{\mathcal{F}}$ the associated equivalence relation. A precedence is *compatible* with $\mathtt{f}$ if for each equation $\mathtt{g}(p_1, \cdots, p_n) \to r$ and each symbol b appearing in r, $b \preceq_{\mathcal{F}} \mathtt{g}$. It is *separating* if for each $\mathbf{c} \in \mathcal{C}, \mathtt{f} \in \mathcal{F}$, $\mathbf{c} \prec_{\mathcal{F}} \mathtt{f}$ (that is constructors are the smallest elements of $\prec_{\mathcal{F}}$ while functions are the biggest). It is *fair* if for each constructors $\mathbf{c}, \mathbf{d}$ with the same arity, $\mathbf{c} \approx_{\mathcal{F}} \mathbf{d}$ and it is *strict* if for each distinct constructors $\mathbf{c}, \mathbf{d}$, $\mathbf{c}$ and $\mathbf{d}$ are incomparable.

Any strict precedence can be canonically extended into a fair precedence.

Definition 9 (Product Extension). Let $\prec$ be an ordering over a set S. Its *product extension* is an ordering $\prec^p$ over tuples of elements of S such that $(m_1, \cdots, m_k) \prec^p (n_1, \cdots, n_k)$ if and only if (i) $\forall i, m_i \preceq n_i$ and (ii) $\exists j$ such that $m_j \prec n_j$.

Definition 10 (PPO). Given a separating precedence $\preceq_{\mathcal{F}}$, the recursive path ordering $\prec_{rpo}$ is defined in Figure 5. If $\prec_{\mathcal{F}}$ is strict (resp. fair) and separating, then the ordering is the *Product Path Ordering* PPO (resp. the *Extended Product Path Ordering* EPPO).

Of course, it is possible to consider other extensions of orderings. Usual choices are the lexicographic extension, thus leading to Lexicographic Path Ordering or the Multiset extension, leading to Multiset Path Ordering. It is also possible to add a notion of *status* to function symbols (Kamin & Lévy 1980) indicating with which extension the parameters must be compared. This leads to the more general Recursive Path Ordering (RPO). However, here we only use the (extended) Product Path Ordering so we do not describe others.

$$\frac{s = t_i \textit{ or } s \prec_{rpo} t_i}{s \prec_{rpo} \mathtt{f}(\ldots, t_i, \ldots)} \ \mathtt{f} \in \mathcal{F} \bigcup \mathcal{C} \qquad \frac{\forall i\ s_i \prec_{rpo} \mathtt{f}(t_1, \cdots, t_n) \qquad \mathtt{g} \prec_{\mathcal{F}} \mathtt{f}}{g(s_1, \cdots, s_m) \prec_{rpo} \mathtt{f}(t_1, \cdots, t_n)} \ \mathtt{f}, \mathtt{g} \in \mathcal{F} \bigcup \mathcal{C}$$

$$\frac{(s_1, \cdots, s_n) \prec^p_{rpo} (t_1, \cdots, t_n) \qquad \mathtt{f} \approx_{\mathcal{F}} \mathtt{g} \qquad \forall i\ s_i \prec_{rpo} \mathtt{f}(t_1, \cdots, t_n)}{\mathtt{g}(s_1, \cdots, s_n) \prec_{rpo} \mathtt{f}(t_1, \cdots, t_n)} \ \mathtt{f}, \mathtt{g} \in \mathcal{F} \bigcup \mathcal{C}$$

Fig. 5. Definition of $\prec_{rpo}$

Given a precedence $\preceq_{\mathcal{F}}$, an equation $l \to r$ is *decreasing* w.r.t. to $\preceq_{\mathcal{F}}$ if we have $r \prec_{rpo} l$. A program is ordered by PPO if there is a separating precedence $\preceq_{\mathcal{F}}$ on $\mathcal{F}$ such that each equation is decreasing w.r.t. $\preceq_{\mathcal{F}}$. Recall that $\prec_{rpo}$ guarantees termination (Dershowitz 1982). Notice that in our case, since patterns cannot contain function symbols, if there is a precedence such that the program is ordered by the corresponding $\prec_{rpo}$, then there is also a compatible one with the same condition.

Lemma 6. Let f be a program and $\prec_{\mathcal{F}}$ be a separating precedence compatible with it. Let $\eta = (\mathtt{f}, v_1, \cdots, v_n, v)$ be a state in a call tree (resp. dag) $\mathcal{T}$ and $\mu = (\mathtt{g}, u_1, \cdots, u_m, u)$ be a descendant of η in $\mathcal{T}$. Then $\mathtt{g} \preceq_{\mathcal{F}} \mathtt{f}$

Proof. Because the precedence is compatible with f. □

Proposition 7 (Computing by Rank). Let f be a program and $\prec_{\mathcal{F}}$ be a separating precedence compatible with it. Let $\mathcal{T}$ be a call dag. Let $A_{\mathcal{T}}$ be the maximum number of descendants of a node with the same precedence:

$$A_{\mathcal{T}} = \max_{\eta = (\mathtt{f}, v_1, \cdots, v_n, v) \in \mathcal{T}} |\{\mu = (\mathtt{g}, u_1, \cdots, u_m, u), \ \eta \text{ is an ancestor of } \mu \text{ and } \mathtt{g} \approx_{\mathcal{F}} \mathtt{f}\}|$$

The size of $\mathcal{T}$ (the number of nodes) is polynomially bounded by $A_{\mathcal{T}}$.

Proof. Let $\mathtt{f}$ be a function symbol. Its rank is $rk(\mathtt{f}) = \max_{\mathtt{g} \prec_{\mathcal{F}} \mathtt{f}} rk(\mathtt{g}) + 1$. Let d be the maximum number of function symbols in a rhs of $\mathtt{f}$ and k be the maximum rank. We will prove by induction on $k-i$ that there are at most $B_i = k^{k-i} \times d^{k-i} \times A_{\mathcal{T}}^{k-i+1}$ nodes in $\mathcal{T}$ at rank i:

— If $k-i=0$, then $i=k$. There are at most $A_{\mathcal{T}} = k^{k-k} d^{k-k} A_{\mathcal{T}}^{k-k+1} = B_k$ nodes at rank k.

— Suppose that the hypothesis is true for all ranks $j > i$. Each node has at most d children. Hence, there are at most $d \sum_{k \geq h > i} B_h$ nodes at rank i whose parent has rank different from i. Each of these nodes has at most $A_{\mathcal{T}}$ descendants at rank i, hence there are at most $d \times A_{\mathcal{T}} \times \sum_{k \geq h > i} B_h$ nodes at rank i. But

$$\begin{aligned} d \cdot A_{\mathcal{T}} \cdot \sum_{k \geq h > i} B_h &\leq d \cdot A_{\mathcal{T}} \cdot \sum_{k \geq h > i} k^{k-h} \cdot d^{k-h} \cdot A_{\mathcal{T}}^{k-h+1} \\ &\leq d \cdot A_{\mathcal{T}} \cdot \sum_{k \geq h > i} k^{k-(i+1)} \cdot d^{k-(i+1)} \cdot A_{\mathcal{T}}^{k-(i+1)+1} \\ &\leq d \cdot A_{\mathcal{T}} \cdot k \cdot k^{k-(i+1)} \cdot d^{k-(i+1)} \cdot A_{\mathcal{T}}^{k-(i+1)+1} \\ &= k^{k-i} \cdot d^{k-i} \cdot A_{\mathcal{T}}^{k-i+1}. \end{aligned}$$

B_i is thus polynomially bounded in $A_{\mathcal{T}}$ and so is the size of the call tree (dag). This concludes the proof. □

Thus to bound the number of nodes in a call-dag (hence bound the derivation's size by Proposition 5) it suffices to establish the bound rank by rank.

Proposition 8. Let $\mathtt{f}$ be a program terminating by PPO, $\mathcal{T}$ be a call dag and $\eta = (\mathtt{f}, v_1, \cdots, v_n, v)$ be a node in $\mathcal{T}$. The number of descendants of η in $\mathcal{T}$ with the same rank as η is polynomially bounded by $|\eta|$.

Proof. Because of the PPO termination ordering, if $\mu = (\mathtt{g}, u_1, \cdots, u_m, u)$ is a descendant of η with $\mathtt{f} \approx_{\mathcal{F}} \mathtt{g}$, then u_i is a subterm of some v_j. There are at most $|v_j|$ subterms of v_j and thus $c \cdot d \cdot \max |v_i|$ possible nodes (where c is the number of functions with the same precedence as $\mathtt{f}$ and d is the maximum arity of symbols in $\mathtt{f}$). □

This is point (2) in the proof of Lemma 51 in (Bonfante et al. 2011). Notice that it only works on a call dag (because identical nodes collapse) and not on a call tree.

3.2. *Quasi-interpretations*

We restrict ourselves to additive QIs as defined in (Bonfante et al. 2011).

Definition 11 (Assignment). An *assignment* of a symbol $b \in \mathcal{F} \bigcup \mathcal{C}$ whose arity is n is a function $(\!|b|\!) : (\mathbb{R})^n \to \mathbb{R}$ such that:

(Subterm) $(\!|b|\!)(X_1, \cdots, X_n) \geq X_i$ for all $1 \leq i \leq n$.

(Weak Monotonicity) $(\!|b|\!)$ is increasing with respect to each variable:

$$\text{for all } 1 \leq i \leq n, X_i \leq Y_i \Rightarrow (\!|b|\!)(X_1, \cdots, X_n) \leq (\!|b|\!)(Y_1, \cdots, Y_n)$$

(Additivity) $(\!|\mathbf{c}|\!)(X_1, \cdots, X_n) = \sum_{i=1}^{n} X_i + a$ if $\mathbf{c} \in \mathcal{C}$ (where $a \geq 1$).
(Polynomial) $(\!|b|\!)$ is bounded by a polynomial.

We extend assignments $(\!|.|\!)$ to terms in the canonical way. Given a term t with n variables, the assignment $(\!|t|\!)$ is a function $(\mathbb{R})^n \to \mathbb{R}$ defined by the rules:

$$(\!|b(t_1, \cdots, t_n)|\!) = (\!|b|\!)((\!|t_1|\!), \cdots, (\!|t_n|\!))$$
$$(\!|x|\!) = X$$

Given two functions $f : (\mathbb{R})^n \to \mathbb{R}$ and $g : (\mathbb{R})^m \to \mathbb{R}$ such that $n \geq m$, we say that $f \geq g$ iff $\forall X_1, \ldots, X_n \in \mathbb{R} : f(X_1, \ldots, X_n) \geq g(X_1, \ldots, X_m)$. There are some well-known and useful consequences of such definitions. We have $(\!|s|\!) \geq (\!|t|\!)$ if t is a subterm of s. Then, for every substitution σ, $(\!|s|\!) \geq (\!|t|\!)$ implies that $(\!|s\sigma|\!) \geq (\!|t\sigma|\!)$.

Definition 12 (Quasi-interpretation). A program assignment $(\!|.|\!)$ is an assignment of each program symbol. An assignment $(\!|.|\!)$ of a program is a quasi-interpretation (QI) if for each equation $l \to r$,

$$(\!|l|\!) \geq (\!|r|\!).$$

In the following, unless explicitly specified, $(\!|.|\!)$ will always denote a QI and not an assignment.

Lemma 9. There exists a constant a such that for any constructor term v, we have $|v| \leq (\!|v|\!) \leq a|v|$.

Proof. By induction. the constant a depends on the constants in the QI of constructors. □

Lemma 10. Assume $\mathtt{f}$ has a QI. Let $\eta = (\mathtt{f}, v_1, \cdots, v_n, v)$ and $\mu = (\mathtt{g}, u_1, \cdots, u_m, u)$ be two states such that $\eta \overset{*}{\rightsquigarrow} \mu$. Then, $(\!|\mathtt{g}(u_1, \cdots, u_m)|\!) \leq (\!|\mathtt{f}(v_1, \cdots, v_n)|\!)$.

Proof. Because $\mathtt{g}(u_1, \cdots, u_m)$ is a subterm of a term obtained by reduction from $\mathtt{f}(v_1, \cdots, v_n)$. □

Proposition 11. Let $\mathtt{f}$ be a program (of arity n) admitting a QI. Then there is a polynomial $p_{\mathtt{f}} : \mathbb{N}^n \to \mathbb{N}$ such that for every $\pi : \langle \emptyset, \mathtt{f}(v_1, \cdots, v_n) \rangle \Downarrow \langle D, v \rangle$, the size of any active judgement in π is bounded by $p_{\mathtt{f}}(|v_1|, \ldots, |v_n|)$.

Proof. Let $\mathtt{g}(u_1, \cdots, u_m) \downarrow q$ be an active judgement in π. The size of $s = \mathtt{g}(u_1, \cdots, u_m)$ is bounded by

$$m \cdot \max |u_i| + 1 \leq m \cdot \max (\!|u_i|\!) + 1 \leq m \cdot (\!|s|\!) + 1.$$

By Lemma 10, $(\!|s|\!) \leq (\!|t|\!)$, where $t = \mathtt{f}(v_1, \cdots, v_n)$. But by polynomiality of QIs there is a polynomial p such that $(\!|t|\!) \leq p((\!|v_1|\!), \ldots, (\!|v_n|\!))$. Since v_i are constructor terms, $(\!|v_i|\!) \leq a|v_i|$. Moreover $|q| \leq (\!|q|\!) \leq (\!|s|\!)$. The claim follows easily. □

Now, if we combine this bound on the size of active terms with the bound on the number of active terms of Proposition 8, we can apply Proposition 5 and conclude that programs terminating by PPO and admitting a QI are polynomial time computable. Actually, all polytime functions can be obtained in this way:

Theorem 12 (P-criterion, (Bonfante et al. 2011)). Any deterministic program $\mathtt{f}$ which (i) terminates by PPO and (ii) admits a QI, is executable in polynomial time with the memoisation semantics. Conversely, any function in FP can be represented by a program of this kind.

4. Blind Abstractions of Programs

4.1. *Definitions*

Our idea is to associate to a given program $\mathtt{f}$ an abstract program $\overline{\mathtt{f}}$ obtained by forgetting each piece of data and replacing it by its *shape*: for instance strings over a finite alphabet are replaced by tally integers (their length), binary trees over a finite alphabet by binary trees without labels etc. Note that in this way, even if $\mathtt{f}$ is deterministic, the associated $\overline{\mathtt{f}}$ will in general not be deterministic.

For that we first define a target language:

— variables: $\overline{\mathcal{X}} = \mathcal{X}$,

— function symbols: we define the map $\overline{(.)}$ on function symbols by: $\overline{\mathtt{f}} = \overline{\mathtt{g}}$ iff $\mathtt{f} = \mathtt{g}$. Then $\overline{\mathcal{F}} = \{\overline{\mathtt{f}} \ / \ \mathtt{f} \in \mathcal{F}\}$.

— constructor symbols: we define the map $\overline{(.)}$ on constructor symbols by: $\overline{\mathbf{c}} = \overline{\mathbf{d}}$ iff $\mathbf{c}$ and $\mathbf{d}$ have the same arity. Then $\overline{\mathcal{C}} = \{\overline{\mathbf{c}} \ / \ \mathbf{c} \in \mathcal{C}\}$.

This language defines a set of values $\mathcal{T}(\overline{\mathcal{C}})$, a set of terms $\mathcal{T}(\overline{\mathcal{C}}, \overline{\mathcal{F}}, \overline{\mathcal{X}})$ and a set of patterns $\overline{\mathcal{P}}$.

The *blinding map* is the natural map $\mathcal{B} : \mathcal{T}(\mathcal{C}, \mathcal{F}, \mathcal{X}) \longrightarrow \mathcal{T}(\overline{\mathcal{C}}, \overline{\mathcal{F}}, \overline{\mathcal{X}})$ induced by the maps above: basically it just identifies constructors of same arity. It induces similar maps on values and patterns. We will write $\overline{t}$ (resp. $\overline{p}$) for $\mathcal{B}(t)$ (resp. $\mathcal{B}(p)$).

For instance, binary strings built from constructors $\{\mathbf{s}_0, \mathbf{s}_1, \mathbf{nil}\}$ will be mapped to unary strings (or tally integers) corresponding to their length, built from $\{\mathbf{s}, \mathbf{0}\}$, where $\overline{\mathbf{s}_0} = \overline{\mathbf{s}_1} = \mathbf{s}$ and $\overline{\mathbf{nil}} = \mathbf{0}$.

The blinding map extends to *equations* in the expected way: given an equation $d = p \rightarrow t$ of the language $(\mathcal{X}, \mathcal{F}, \mathcal{C})$, we set $\overline{d} = \mathcal{B}(d) = \overline{p} \rightarrow \overline{t}$. Finally, given a program $\mathtt{f} = (\mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E})$, its blind image is $\overline{\mathtt{f}} = (\overline{\mathcal{X}}, \overline{\mathcal{C}}, \overline{\mathcal{F}}, \overline{\mathcal{E}})$ where the equations are obtained by: $\overline{\mathcal{E}} = \{\overline{d}, d \in \mathcal{E}\}$. Observe that even if $\mathtt{f}$ is an orthogonal program, this will not necessarily be the case of $\overline{\mathtt{f}}$, because some patterns are identified by $\mathcal{B}$. The denotational semantics of $\overline{\mathtt{f}}$ can be seen as a relation over the domain $\mathcal{T}(\overline{\mathcal{C}})$.

We mentioned strings over a binary alphabet, but note that on some other data-types, like for instance lists over tally integers, the blinding map might just act as the identity and thus not have much interest. Other notions of abstractions are probably worth considering, but we stick here to the blinding map for simplicity.

4.2. *Complexity Definitions*

Definition 13 (Strongly polytime). We say a (possibly) non-deterministic program $\mathtt{f}$ (of arity n) is *strongly polytime* if there exists a polynomial $p_{\mathtt{f}} : \mathbb{N}^n \rightarrow \mathbb{N}$ such that for any sequence $v_1, \cdots, v_n$ and any $\pi : \mathtt{f}(v_1, \cdots, v_n) \downarrow u$, it holds that $|\pi| \leq p_{\mathtt{f}}(v_1, \ldots, v_n)$.

		f			$\overline{\texttt{f}}$
$\texttt{f}(\mathbf{s}_0\mathbf{s}_i x)$	$\to$	$\texttt{append}(\texttt{f}(\mathbf{s}_1 x), \texttt{f}(\mathbf{s}_1 x))$	$\overline{\texttt{f}}(\mathbf{ss}x)$	$\to$	$\overline{\texttt{append}}(\overline{\texttt{f}}(\mathbf{s}x), \overline{\texttt{f}}(\mathbf{s}x))$
$\texttt{f}(\mathbf{s}_1 x)$	$\to$	x	$\overline{\texttt{f}}(\mathbf{s}x)$	$\to$	x
$\texttt{f}(\mathbf{nil})$	$\to$	$\mathbf{nil}$	$\overline{\texttt{f}}(0)$	$\to$	0
$\texttt{append}(\mathbf{s}_i x, y)$	$\to$	$\mathbf{s}_i\texttt{append}(x, y)$	$\overline{\texttt{append}}(\mathbf{s}x, y)$	$\to$	$\mathbf{s}\ \overline{\texttt{append}}(x, y)$
$\texttt{append}(\mathbf{nil}, y)$	$\to$	y	$\overline{\texttt{append}}(0, y)$	$\to$	y

Fig. 6. Blind abstraction of our running example

Of course similar definitions would also make sense for other complexity bounds than polynomial time. In the case of a deterministic program, this definition coincides with that of a polynomial time program.

Definition 14 (Blindly Polytime). A program f is *blindly polytime* if its blind abstraction $\overline{\texttt{f}}$ is strongly polytime.

Observe that:

Fact 2. If a program f is blindly polytime, then it is polynomial time in the call by value semantics.

Indeed, it is sufficient to observe that any execution (derivation proof) of f can be mapped by $\mathcal{B}$ to an execution of $\overline{\texttt{f}}$. But of course not all $\overline{\texttt{f}}$ executions are obtained in this way. Observe for that our running example in Figure 6: note that f terminates in polynomial time but this is not the case for $\overline{\texttt{f}}$. Indeed if we denote $\underline{n} = \underbrace{\mathbf{s}\ldots\mathbf{s}}_{n}\ \mathbf{0}$, we have that $\overline{\texttt{f}}(\underline{n})$ can be reduced in an exponential number of steps, with a $\pi : \overline{\texttt{f}}(\underline{n}) \downarrow 0$.

Note that the property of being blindly polytime is indeed a strong condition, because it means in some sense that the program will terminate with a polynomial bound for reasons which are indifferent to the actual content of the input but only depend on its size. For instance, it can be checked that the insertion sort algorithm, written in a natural way, gives a blindly polytime program (see Example 1).

Another way to think of blinding is to imagine that some *errors* might occur during the execution of the program, where by an error we mean that a constructor in the current term is replaced by another constructor of same arity (for instance in a string a 0-bit is replaced by a 1-bit): blindly polytime programs are then programs that remain polynomial time, no matter the number of errors occurring during the execution.

Our motivation for introducing blind abstraction and the notion of blindly polytime program is to provide a more abstract method to compare different ICC systems than the one consisting simply in providing examples handled by one system and not by the other. This kind of abstraction indeed gives a way to subdivide the class of polytime programs into smaller classes.

To illustrate our idea consider the picture of Figure 7. The fat lines represent classes of programs defined by an intensional property (being polytime or blindly polytime). Imagine we have three ICC criteria (A), (B) and (C) implying that any program satisfying

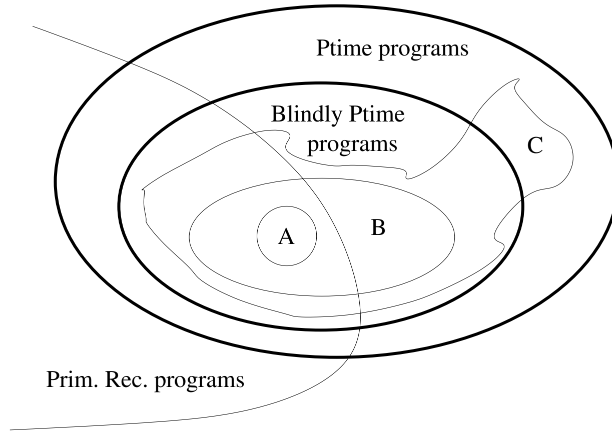


Fig. 7. Subclasses of programs

one of these criteria is polytime. If one criterion, say (B), is only able to validate blindly polytime programs while another one, (C), can handle non blindly polytime programs, then one can argue that (C) is more accurate than (B). Moreover if (A) only validates primitive recursive programs while (B) validates some blindly polytime programs that are not primitive recursive, then (B) is more accurate than (A).

Example 1. Here are two programs for sorting binary strings. The first one uses insertion sort while the second one uses bubble sort. In the first case, since we are working on binary strings there it is sufficient to have an auxiliary function to insert 1 into a sorted string and pre-pend 0 at the beginning of the result. In the second case, we need to introduce pairing (and `let in`) as well as boolean value as a flag to check whether a permutation occurred or not. This is a quite straightforward extension of the language. In `bsort` the flag is initially set to `true`, and is turned to `false` when an exchange is performed. Booleans and boolean tests remain unchanged by the blinding map.

isort			$\overline{\texttt{isort}}$		
$\texttt{insert1}(\mathbf{nil})$	$\rightarrow$	$\mathrm{s}_1\mathbf{nil}$	$\overline{\texttt{insert1}}(0)$	$\rightarrow$	$\mathrm{s}0$
$\texttt{insert1}(\mathrm{s}_0 x)$	$\rightarrow$	$\mathrm{s}_0\texttt{insert1}(x)$	$\overline{\texttt{insert1}}(\mathrm{s}x)$	$\rightarrow$	$\mathrm{s}\overline{\texttt{insert1}}(x)$
$\texttt{insert1}(\mathrm{s}_1 x)$	$\rightarrow$	$\mathrm{s}_1\mathrm{s}_1 x$	$\overline{\texttt{insert1}}(\mathrm{s}x)$	$\rightarrow$	$\mathrm{ss}x$
$\texttt{isort}(\mathbf{nil})$	$\rightarrow$	$\mathbf{nil}$	$\overline{\texttt{isort}}(0)$	$\rightarrow$	0
$\texttt{isort}(\mathrm{s}_0 x)$	$\rightarrow$	$\mathrm{s}_0\texttt{isort}(x)$	$\overline{\texttt{isort}}(\mathrm{s}x)$	$\rightarrow$	$\mathrm{s}\overline{\texttt{isort}}(x)$
$\texttt{isort}(\mathrm{s}_1 x)$	$\rightarrow$	$\texttt{insert1}(\texttt{isort}(x))$	$\overline{\texttt{isort}}(\mathrm{s}x)$	$\rightarrow$	$\overline{\texttt{insert1}}(\overline{\texttt{isort}}(x))$

	bsort	
$\texttt{bubble}(\mathbf{nil}, b)$	$\to$	$\langle \mathbf{nil}, b\rangle$
$\texttt{bubble}(\mathbf{s}_i\mathbf{nil}, b)$	$\to$	$\langle \mathbf{s}_i\mathbf{nil}, b\rangle$
$\texttt{bubble}(\mathrm{s}_0\mathrm{s}_i x, b)$	$\to$	$\texttt{let } \langle y, c\rangle = \texttt{bubble}(\mathrm{s}_i x, b) \texttt{ in } \langle \mathrm{s}_0 y, c\rangle$
$\texttt{bubble}(\mathrm{s}_1\mathrm{s}_1 x, b)$	$\to$	$\texttt{let } \langle y, c\rangle = \texttt{bubble}(\mathrm{s}_1 x, b) \texttt{ in } \langle \mathrm{s}_1 y, c\rangle$
$\texttt{bubble}(\mathrm{s}_1\mathrm{s}_0 x, b)$	$\to$	$\texttt{let } \langle y, c\rangle = \texttt{bubble}(\mathrm{s}_1 x, b) \texttt{ in } \langle \mathrm{s}_0 y, \texttt{false}\rangle$
$\texttt{bsort}(x)$	$\to$	$\texttt{let } \langle y, b\rangle = \texttt{bubble}(x, \texttt{true}) \texttt{ in}$ $\texttt{if } b \texttt{ then } y \texttt{ else } \texttt{bsort}(y)$

	$\overline{\texttt{bsort}}$	
$\overline{\texttt{bubble}}(0, b)$	$\to$	$\langle 0, b\rangle$
$\overline{\texttt{bubble}}(\mathrm{s}0, b)$	$\to$	$\langle \mathrm{s}0, b\rangle$
$\overline{\texttt{bubble}}(\mathrm{ss}x, b)$	$\to$	$\texttt{let } \langle y, c\rangle = \overline{\texttt{bubble}}(\mathrm{s}x, b) \texttt{ in } \langle \mathrm{s}y, c\rangle$
$\overline{\texttt{bubble}}(\mathrm{ss}x, b)$	$\to$	$\texttt{let } \langle y, c\rangle = \overline{\texttt{bubble}}(\mathrm{s}x, b) \texttt{ in } \langle \mathrm{s}y, c\rangle$
$\overline{\texttt{bubble}}(\mathrm{ss}x, b)$	$\to$	$\texttt{let } \langle y, c\rangle = \overline{\texttt{bubble}}(\mathrm{s}x, b) \texttt{ in } \langle \mathrm{s}y, \texttt{false}\rangle$
$\overline{\texttt{bsort}}(x)$	$\to$	$\texttt{let } \langle y, b\rangle = \overline{\texttt{bubble}}(x, \texttt{true}) \texttt{ in}$ $\texttt{if } b \texttt{ then } y \texttt{ else } \overline{\texttt{bsort}}(y)$

Both programs are polytime. The first one terminates by PPO and admits a QI, and hence satisfies the P-criterion. However, it does not follow the safe recursion schemata (Bellantoni & Cook 1992) because the result of `isort` is used to control the recursion of *insert*. The insertion sort is also blindly polytime. Indeed, every recursive call decreases the length of the string whatever the precise value of the elements is.

The bubble sort, on the other hand, is not blindly polytime or even blindly terminating. Indeed, the length of the string does not decrease and termination is ensured only because the elements inside the string are reordered. Actually, the same comparison can be done several times and termination of the algorithm is due to the fact that the same comparison always yields the same result. This is no longer true after blinding the program because then comparison between elements of the string becomes non deterministic.

Now we want to discuss the behaviour of the blinding map with respect to criteria on TRS based on recursive path orderings (RPO) and QIs ((Bonfante et al. 2011)).

4.3. *Blinding and Recursive Path Orderings*

Now we want to examine what happens with the ordering through the blinding operation. We have:

Lemma 13. Let `f` be a program: if `f` terminates by PPO then $\overline{\texttt{f}}$ terminates by PPO.

Indeed $\mathcal{F}$ and $\overline{\mathcal{F}}$ are in one-one correspondence, and it is easy to observe that: if $\prec_{\mathcal{F}}$ is a precedence which gives a PPO ordering for `f`, then the corresponding $\prec_{\overline{\mathcal{F}}}$ does the same for $\overline{\texttt{f}}$.

The converse is not true, see for example Figure 6 where the first equation does terminate by PPO on the blind side but not on the non-blind side. However, we have:

Proposition 14. Let f be a program. The three following statements are equivalent: (i) `f` terminates by EPPO, (ii) $\overline{\texttt{f}}$ terminates by EPPO, (iii) $\overline{\texttt{f}}$ terminates by PPO.

Proof. Just observe that on $\mathcal{T}(\overline{\mathcal{C}}, \overline{\mathcal{F}}, \overline{\mathcal{X}})$, PPO and EPPO coincide, and that $\bar{t} \prec_{EPPO} \bar{s}$ implies $t \prec_{EPPO} s$. □

4.4. *Blinding and Quasi-interpretations*

Assume the program `f` admits a quasi-interpretation $(\!|.|\!)$. Then in general this does not imply that $\overline{\texttt{f}}$ admits a quasi-interpretation. Indeed one reason why $(\!|.|\!)$ cannot be simply converted into a quasi-interpretation for $\overline{\texttt{f}}$ is because a quasi-interpretation might in general give different assignments to several constructors of the same arity, for instance when $(\!|\mathbf{s}_0|\!)(X) = X+1$ and $(\!|\mathbf{s}_1|\!)(X) = X+2$. Then when considering $\overline{\texttt{f}}$ there is no natural choice for $(\!|\mathbf{s}|\!)$.

However, in most examples in practice, a restricted class of quasi-interpretations is used:

Definition 15 (Uniform assignments). An assignment for $\texttt{f} = \langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E} \rangle$ is *uniform* if all constructors of same arity have the same assignment: for each $\mathbf{c}, \mathbf{d} \in \mathcal{C}$, $\mathit{arity}(\mathbf{c}) = \mathit{arity}(\mathbf{d})$ implies $(\!|\mathbf{c}|\!) = (\!|\mathbf{d}|\!)$. A quasi-interpretation of `f` is uniform if it is defined by a uniform assignment.

Now we have:

Proposition 15. The program `f` admits a uniform quasi-interpretation *iff* $\overline{\texttt{f}}$ admits a quasi-interpretation.

5. Linear Programs and Call by Value Semantics

5.1. *Definitions and Main Property*

Now we want to use the blinding transformation to examine properties of programs satisfying the P-criterion (Theorem 12). Actually for that we will restrict our attention to particular programs, *linear* ones. Essentially this means that in recursive definitions (equations) only one recursive call is allowed. This is a rather natural restriction when one has in mind to control the complexity, because it prevents getting an exponential blow-up on the number of recursive calls at execution time. Indeed many common programs are linear. This notion was used in (Bonfante et al. 2011), in combination with lexicographic path ordering (LPO). Let us now give a formal definition:

Definition 16 (Linearity). Let `f` be a program terminating by a recursive path ordering given by a precedence $\preceq_{\mathcal{F}}$ and `g` be a function symbol in `f`. We say `g` is *linear* in `f` w.r.t. $\preceq_{\mathcal{F}}$ if, in the rhs term of any equation for `g`, there is at most one occurrence of a function symbol `h` with same precedence as `g`. The program `f` is linear w.r.t. $\preceq_{\mathcal{F}}$ if all its function symbols are linear w.r.t. $\preceq_{\mathcal{F}}$.

We will omit mentioning the precedence $\preceq_{\mathcal{F}}$ when there is no ambiguity.

Proposition 16. Let `f` be a (possibly non deterministic) program which i) terminates by PPO, ii) admits a quasi-interpretation, iii) is linear. Then `f` is strongly polytime.

Before starting the proof, note that the differences with the P-criterion theorem from (Bonfante et al. 2011) (Theorem 12) are that: the program here does not need to be deterministic, but linearity is assumed for all function symbols.

We have considered here the linearity assumption because we are interested in non-deterministic programs and memoisation is problematic in presence of non-determinism (see Section 2.1); linearity is a sufficient condition to avoid the use of memoisation and keep complexity properties.

Proof of Prop. 16. We consider a derivation proof $\pi : \mathtt{f}(v_1, \cdots, v_n) \downarrow u$. As the program has a quasi-interpretation, Proposition 11 gives a bound S on the size of active judgements which depends polynomially on $|v_1|, \ldots, |v_n|$. Let $\mathcal{T}$ be a call tree associated to π and $\eta = (\mathtt{f}, v_1, \cdots, v_n, v)$ be a node in it. Consider now the subset of nodes of $\mathcal{T}$ obtained by keeping η, the sons of η with same precedence as `f` and proceeding hereditarily in this way. We call this set the set of descendants of $\eta = (\mathtt{f}, v_1, \cdots, v_n, v)$ with same precedence as `f`, and the linearity of the program actually implies that this set is a branch; so its number of elements corresponds to its depth. Termination by PPO ensures that if $\mu = (\mathtt{g}, u_1, \cdots, u_m, u)$ is the child of η with $\mathtt{f} \approx_{\mathcal{F}} \mathtt{g}$ then $|u_i| \leq |v_i|$ and that there is at least one j such that $|u_j| < |v_j|$. So, the number of descendants of η with the same precedence is bounded by $\sum_{i=1}^{n} |v_i|$. This thus bounds the number of active judgements by rank. So we can now use Proposition 1 and conclude that $|\pi|$ is polynomially bounded in A, S, hence also in $|v_1|, \ldots, |v_n|$. Therefore `f` is strongly polytime. □

Theorem 17. Let `f` be a (possibly non deterministic) program which i) terminates by PPO, ii) admits a *uniform* quasi-interpretation, iii) is linear. Then `f` is blindly polytime.

Proof. Note that:
— `f` terminates by PPO, so $\overline{\mathtt{f}}$ also, by Lemma 13;
— the quasi-interpretation for `f` is uniform, so $\overline{\mathtt{f}}$ admits a quasi-interpretation, by Prop. 15;
— `f` is linear, so $\overline{\mathtt{f}}$ is also linear.

So by Proposition 16 we deduce that $\overline{\mathtt{f}}$ is strongly polytime. Therefore `f` is blindly polytime. □

5.2. *Bellantoni-Cook Programs*

To show an example of application of Theorem 17 to a simple class of programs, let us consider the class of Bellantoni-Cook safe recursive programs (Bellantoni & Cook 1992). Of course this is a very particular case but we think it is interesting because this class is an important reference in the implicit computational complexity literature.

Let BC be the class of Bellantoni-Cook programs written in a Term Rewriting System framework as in (Moyen 2003). Function arguments are separated into either *safe* or *normal* arguments, recurrences can only occur over normal arguments and their result can only be used in a safe position. We use here a semi-colon to distinguish between normal (on the left) and safe (on the right) parameters.

Definition 17 (Bellantoni-Cook programs). The class BC is the smallest class of programs containing the initial functions:
— (**Constant**) $\mathbf{0}$
— (**Successors**) $\mathtt{s}_i(x), i \in \{0,1\}$
— (**Projection**) $\pi_j^{n,m}(x_1, \cdots, x_n; x_{n+1}, \cdots, x_{n+m}) \to x_j$
— (**Predecessor**) $\mathtt{p}(;\mathbf{0}) \to \mathbf{0} \quad p(;\mathtt{s}_i(x)) \to x$
— (**Conditional**) $\mathtt{C}(;\mathbf{0}, x, y) \to x \quad \mathtt{C}(;\mathtt{s}_0(z), x, y) \to x \quad \mathtt{C}(;\mathtt{s}_1(z), x, y) \to y$
and being closed by:
— (**Safe recursion**)

$$\begin{aligned}\mathtt{f}(\mathbf{0}, x_1, \cdots, x_n; y_1, \cdots, y_m) \to& \mathtt{g}(x_1, \cdots, x_n; y_1, \cdots, y_m)\\ \mathtt{f}(\mathtt{s}_i(z), x_1, \cdots, x_n; y_1, \cdots, y_m) \to& \mathtt{h}_i(z, x_1, \cdots, x_n; y_1, \cdots, y_m,\\ & \mathtt{f}(z, x_1, \cdots, x_n; y_1, \cdots, y_m)), i \in \{0,1\}\end{aligned}$$

with $\mathtt{g}, \mathtt{h}_i \in$ BC (previously defined) ;
— (**Safe composition**)

$$\mathtt{f}(x_1, \cdots, x_n; y_1, \cdots, y_m) \to \mathtt{g}(\mathtt{h}_1(x_1, \cdots, x_n;), \ldots, \mathtt{h}_p(x_1, \cdots, x_n;);$$
$$\mathtt{l}_1(x_1, \cdots, x_n; y_1, \cdots, y_m), \ldots, \mathtt{l}_q(x_1, \cdots, x_n; y_1, \cdots, y_m))$$

with $\mathtt{g}, \mathtt{h}_i, \mathtt{l}_j \in$ BC ;

It is easy to see that any BC program terminates by PPO and is linear.

Proposition 18 (Quasi-interpretations for BC-programs). A BC-program admits the following quasi-interpretation:
— $(\!|\mathbf{0}|\!) = 1$;
— $(\!|\mathtt{s}_i|\!)(X) = X + 1$;
— $(\!|\pi|\!)(X_1, \cdots, X_{n+m}) = \max(X_1, \cdots, X_{n+m})$;
— $(\!|\mathtt{p}|\!)(X) = X$;
— $(\!|\mathtt{C}|\!)(X, Y, Z) = \max(X, Y, Z)$;
For functions defined by safe recursion or composition, $(\!|\mathtt{f}|\!)(X_1, \cdots, X_n; Y_1, \cdots, Y_m) = q_{\mathtt{f}}(X_1, \cdots, X_n) + \max(Y_1, \cdots, Y_m)$ with $q_{\mathtt{f}}$ defined as follows:
— $q_{\mathtt{f}}(A, X_1, \cdots, X_n) = A(q_{\mathtt{h}_0}(A, X_1, \cdots, X_n) + q_{\mathtt{h}_1}(A, X_1, \cdots, X_n)) + q_{\mathtt{g}}(X_1, \cdots, X_n)$
if $\mathtt{f}$ is defined by safe recursion ;

— $q_{\mathtt{f}}(X_1, \cdots, X_n) = q_{\mathtt{g}}(q_{\mathtt{h}_1}(X_1, \cdots, X_n), \ldots, q_{\mathtt{h}_p}(X_1, \cdots, X_n)) + \sum_i q_{\mathtt{l}_i}(X_1, \cdots, X_n)$ if `f` is defined by safe composition.

Proof. It is easy to check that all conditions of Definitions 11 and 12 are satisfied. □

6. Semi-lattice of Quasi-Interpretations

The study of necessary conditions on programs satisfying the P-criterion has drawn our attention to uniform quasi-interpretations. This suggests to consider quasi-interpretations with fixed assignments for constructors and to examine their properties as a class.

Definition 18 (Compatible assignments). Let `f` be a program and $(\!|\cdot|\!)_1$, $(\!|\cdot|\!)_2$ be two assignments for `f`. We say that they are *compatible* if for any constructor symbol **c** we have: $(\!|\mathbf{c}|\!)_1 = (\!|\mathbf{c}|\!)_2$. A family of assignments for `f` is compatible if its elements are pairwise compatible. We use these same definitions for quasi-interpretations.

Each choice of assignments for constructors thus defines a *maximal compatible family* of quasi-interpretations for a program `f`: all quasi-interpretations for `f` which take these values on $\mathcal{C}$. We consider on assignments the extensional order $\leq$: $(\!|\cdot|\!)_1 \leq (\!|\cdot|\!)_2$ iff

$$\forall b \in \mathcal{C} \cup \mathcal{F}, \forall \vec{x} \in (\mathbb{R}^+)^k, (\!|b|\!)_1(\vec{x}) \leq (\!|b|\!)_2(\vec{x}).$$

Given two compatible assignments $(\!|\cdot|\!)_1$, $(\!|\cdot|\!)_2$ we denote by $(\!|\cdot|\!)_1 \wedge (\!|\cdot|\!)_2$ the assignment $(\!|\cdot|\!)_0$ defined by:

$$\begin{array}{lcl} \forall \mathbf{c} \in \mathcal{C}, \quad (\!|\mathbf{c}|\!)_0 & = & (\!|\mathbf{c}|\!)_1 = (\!|\mathbf{c}|\!)_2 \\ \forall \mathsf{g} \in \mathcal{F}, \quad (\!|\mathsf{g}|\!)_0 & = & (\!|\mathsf{g}|\!)_1 \wedge (\!|\mathsf{g}|\!)_2 \end{array}$$

where $\alpha \wedge \beta$ denotes the greatest lower bound of $\{\alpha, \beta\}$ in the point-wise order. Then we have:

Proposition 19. Let `f` be a program and $(\!|\cdot|\!)_1$, $(\!|\cdot|\!)_2$ be two compatible quasi-interpretations for it, then $(\!|\cdot|\!)_1 \wedge (\!|\cdot|\!)_2$ is also a quasi-interpretation for `f`.

To establish this Proposition we need intermediary Lemmas. We continue to denote $(\!|\cdot|\!)_0 = (\!|\cdot|\!)_1 \wedge (\!|\cdot|\!)_2$:

Lemma 20. For any `g` of $\mathcal{F}$ we have that $(\!|\mathsf{g}|\!)_0$ is monotone and satisfies the subterm property.

Proof. To prove monotonicity, assume $\vec{x} \leq \vec{y}$, for the product ordering. Then, for $i = 1$ or 2: $(\!|\mathsf{g}|\!)_0(\vec{x}) = (\!|\mathsf{g}|\!)_1(\vec{x}) \wedge (\!|\mathsf{g}|\!)_2(\vec{x}) \leq (\!|\mathsf{g}|\!)_i(\vec{x}) \leq (\!|\mathsf{g}|\!)_i(\vec{y})$, using monotonicity of $(\!|\mathsf{g}|\!)_i$. As this is true for $i = 1$ and 2 we thus have: $(\!|\mathsf{g}|\!)_0(\vec{x}) \leq (\!|\mathsf{g}|\!)_1(\vec{y}) \wedge (\!|\mathsf{g}|\!)_2(\vec{y}) = (\!|\mathsf{g}|\!)_0(\vec{y})$. It is also easy to check that $(\!|\mathsf{g}|\!)_0$ satisfies the subterm property. □

Lemma 21. Let t be a term. We have: $(\!|t|\!)_0 \leq (\!|t|\!)_i$, for $i = 1, 2$.

Proof. By induction on t, using the definition of $(\!|\mathsf{g}|\!)_0$ and $(\!|\mathbf{c}|\!)_0$, and the monotonicity property of Lemma 20. □

Lemma 22. Let $\mathtt{g}(p_1, \cdots, p_n) \to t$ be an equation of the program $\mathtt{f}$. We have:

$$(\!|\mathtt{g}|\!)_0 \circ ((\!|p_1|\!)_0, \ldots, (\!|p_n|\!)_0) \geq (\!|t|\!)_0.$$

Proof. As patterns only contain constructor and variable symbols, and by definition of $(\!|\cdot|\!)_0$, if p is a pattern we have: $(\!|p|\!)_0 = (\!|p|\!)_1 = (\!|p|\!)_2$. Let $i = 1$ or 2; we have:

$$\begin{array}{rcll} (\!|\mathtt{g}|\!)_i((\!|p_1|\!)_i(\vec{x}), \ldots, (\!|p_n|\!)_i(\vec{x})) & \geq & (\!|t|\!)_i(\vec{x}) & \text{because } (\!|\cdot|\!)_i \text{ is a quasi-interpretation,} \\ & \geq & (\!|t|\!)_0(\vec{x}) & \text{with Lemma 21.} \end{array}$$

So:

$$(\!|\mathtt{g}|\!)_i((\!|p_1|\!)_0(\vec{x}), \ldots, (\!|p_n|\!)_0(\vec{x})) \geq (\!|t|\!)_0(\vec{x}), \text{ as } (\!|p_j|\!)_i = (\!|p_j|\!)_0.$$

Write $\vec{y} = ((\!|p_1|\!)_0(\vec{x}), \ldots, (\!|p_n|\!)_0(\vec{x}))$. As $(\!|\mathtt{g}|\!)_1(\vec{y}) \geq (\!|t|\!)_0(\vec{x})$ and $(\!|\mathtt{g}|\!)_2(\vec{y}) \geq (\!|t|\!)_0(\vec{x})$, by definition of $(\!|\mathtt{g}|\!)_0$ we get $(\!|\mathtt{g}|\!)_0(\vec{y}) \geq (\!|t|\!)_0(\vec{x})$, which ends the proof. □

Now we can proceed with the proof of Prop. 19:

Proof of Prop. 19. We need to check that the conditions of Definitions 11 and 12 of assignment and quasi-interpretation respectively are satisfied. Observe that Lemma 20 ensures that $(\!|\cdot|\!)_0$ satisfies the (Weak Monotonicity) and the (Subterm) conditions, and Lemma 22 that it satisfies the condition w.r.t. the equations of the program. Condition (Additivity) is satisfied because $(\!|\cdot|\!)_1$ and $(\!|\cdot|\!)_2$ are compatible and satisfy this condition by assumption. Finally condition (Polynomial) is satisfied because if $(\!|b|\!)_1$ and $(\!|b|\!)_2$ are bounded by polynomials then so is $(\!|b|\!)_1 \wedge (\!|b|\!)_2$. Therefore $(\!|\cdot|\!)_0$ is a quasi-interpretation. □

Proposition 23. Let $\mathtt{f}$ be a program and $\mathcal{Q}$ be a non-empty family of compatible quasi-interpretations for $\mathtt{f}$, then $\wedge_{(\!|\cdot|\!) \in \mathcal{Q}} (\!|\cdot|\!)$ is a quasi-interpretation for $\mathtt{f}$. Therefore maximal compatible families of quasi-interpretations for $\mathtt{f}$ have an inferior semi-lattice structure for $\leq$.

Proof. It is sufficient to generalise Lemmas 20 and 22 to the case of an arbitrary family $\mathcal{Q}$ and to apply the same argument as for the proof of Prop. 19. □

Remark 1. Note that if, starting from two quasi-interpretations $(\!|\cdot|\!)_1$ and $(\!|\cdot|\!)_2$, we define the assignment $(\!|\cdot|\!) = (\!|\cdot|\!)_1 \vee (\!|\cdot|\!)_2$ in a similar way as $(\!|\cdot|\!)_1 \wedge (\!|\cdot|\!)_2$ with the point-wise lowest upper bound, then in general $(\!|\cdot|\!)$ is not a quasi-interpretation. Indeed the following counter-example shows that the statement of Proposition 22 is not satisfied in this case.

Take $\mathtt{f}, \mathtt{g}, \mathtt{h} \in \mathcal{F}$, $\mathbf{c} \in \mathcal{C}$, and the equation:

$$\mathtt{g}(\mathbf{c}(x)) \to \mathtt{f}(\mathtt{h}(\mathtt{g}(x))).$$

Consider then $(\!|\cdot|\!)_1$ and $(\!|\cdot|\!)_2$ given by:

$$\begin{array}{lll} (\!|\mathtt{g}|\!)_1(x) = x, & (\!|\mathtt{f}|\!)_1(x) = x + 1, & (\!|\mathtt{h}|\!)_1(x) = x, \\ (\!|\mathtt{g}|\!)_2(x) = x, & (\!|\mathtt{f}|\!)_2(x) = x, & (\!|\mathtt{h}|\!)_2(x) = x + 1. \end{array}$$

Thus we have, $(\!|\mathtt{g}|\!)(x) = x$, $(\!|\mathtt{g}|\!)(x) = x + 1$, $(\!|\mathtt{h}|\!)(x) = x + 1$. Hence: $(\!|\mathtt{g}(\mathbf{c}(x))|\!) = x + 1$, but $(\!|\mathtt{f}(\mathtt{h}(\mathtt{g}(x)))|\!) = x + 2$.

7. Extending the P-criterion

Blind abstraction suggests to consider not only the PPO ordering from the P-criterion, but also an extension which is invariant by the blinding map, the EPPO ordering (see Subsection 4.3). It is thus natural to ask whether EPPO enjoys the same property as PPO. We prove in this section that if we consider programs over strings, with EPPO we can still bound the size of the call-dag and thus generalise the P-criterion. Then, we will also consider the *bounded value property* which is an extension of the notion of QI. Here, we bound the number of nodes in the call-dag with a given precedence. Then, Proposition 7 bounds the total number of nodes in the call-dag.

In this Section we restrict to deterministic programs handling strings over a finite alphabet (*i.e.* words), that is to say we take $\mathcal{C} = \{\mathbf{s}_0, \dots, \mathbf{s}_m, \mathbf{nil}\}$, where each $\mathbf{s}_i$ has arity 1 and **nil** has arity 0. We call these programs over unary constructors.

Example 2. First, let us show that EPPO is (intensionally) strictly more powerful than PPO. It is quite obvious that any program terminating by PPO also terminates with EPPO. The following program works overs binary integers (that is, there are two successors) represented with least significant bit first (that is $\mathbf{s}_0\mathbf{s}_1\mathbf{nil}$ corresponds to the integer 10, *i.e.* 2). It computes† x^y with a "fast exponentiation algorithm" in base 4, that is using the recurrence $x^{4y} = ((x^y)^2)^2$, and similar rules for other cases (we omit here the base cases). The rules for multiplication and squaring are quite standard and are not written here. The last column explains the meaning of each rule.

$$\begin{array}{ll}
\mathtt{pow}(x, \mathbf{s}_0\mathbf{s}_0 y) \to \mathtt{sq}(\mathtt{sq}(\mathtt{pow}(x,y))) & (x^{4y} = ((x^y)^2)^2) \\
\mathtt{pow}(x, \mathbf{s}_1\mathbf{s}_0 y) \to \mathtt{mult}(x, \mathtt{sq}(\mathtt{pow}(x, \mathbf{s}_0 y))) & (x^{4y+1} = (x^{2y})^2 \times x) \\
\mathtt{pow}(x, \mathbf{s}_0\mathbf{s}_1 y) \to \mathtt{sq}(\mathtt{mult}(x, \mathtt{pow}(x, \mathbf{s}_0 y)) & (x^{4y+2} = (x^{2y} \times x)^2) \\
\mathtt{pow}(x, \mathbf{s}_1\mathbf{s}_1 y) \to \mathtt{mult}(x, \mathtt{sq}(\mathtt{mult}(x, \mathtt{pow}(x, \mathbf{s}_0 y)))) & (x^{4y+3} = (x^{2y} \times x)^2 \times x)
\end{array}$$

This program does not terminate by PPO since the last rule is not ordered by PPO (because $\mathbf{s}_0 y$ is not comparable with $\mathbf{s}_1\mathbf{s}_1 y$). However, the blind abstraction of this program does terminate by PPO and the program itself terminates by EPPO. Indeed, with a fair precedence, $\mathbf{s}_0$ and $\mathbf{s}_1$ have the same precedence, hence $\mathbf{s}_0 y \prec_{eppo} \mathbf{s}_1\mathbf{s}_1 y$.

Fact 3. Since we are working over words (unary constructors), patterns are either constructor terms (that is, words), or have the form $p = \mathbf{s}^1(\mathbf{s}^2 \dots \mathbf{s}^n(x)\dots)$ where the $\mathbf{s}^i$ for $1 \leq i \leq n$ are constructors. In the second case, we will write $p = \Omega(x)$ with $\Omega = \mathbf{s}^1\mathbf{s}^2 \dots \mathbf{s}^n$.

The length of a pattern is the length of the corresponding word: $|p| = |\Omega|$

Proposition 24. In a program terminating by PPO or EPPO, the only calls at the same precedence that can occur are of the form

$$\mathtt{f}(p_1, \cdots, p_n) \to C[\mathtt{g}_1(q_1^1, \cdots, q_m^1), \dots, \mathtt{g}_p(q_1^p, \cdots, q_l^p)]$$

† in a slightly contrived way.

where $C[\cdot]$ is some context, $\mathtt{f} \approx_{\mathcal{F}} \mathtt{g}_k$ and p_i, q_j^k are patterns. Moreover, each variable appearing in a q_i^k appears in p_i.

Proof. This is a direct consequence of the termination ordering. □

Since we will only consider individual calls, we will put in the context all but one of the $\mathtt{g}_k$: $\mathtt{f}(p_1, \cdots, p_n) \to C[\mathtt{g}(q_1, \cdots, q_m)]$.

Definition 19 (Production size). Let $\mathtt{f}$ be a program terminating by EPPO. Let $\mathtt{f}(p_1, \cdots, p_n) \to C[\mathtt{g}(q_1, \cdots, q_m)]$ be a call in it where $\mathtt{f} \approx_{\mathcal{F}} \mathtt{g}$. The *production size* of this call is $\max_{1 \leq i \leq m}\{|q_i|\}$. The production size of an equation is the greatest production size of any call (at the same precedence) in it. That is if we have an equation $e = \mathtt{f}(p_1, \cdots, p_n) \to C[\mathtt{g}_1(q_1^1, \cdots, q_m^1), \dots, \mathtt{g}_p(q_1^p, \cdots, q_l^p)]$ where $\mathtt{f} \approx_{\mathcal{F}} \mathtt{g}_k$ for $1 \leq k \leq p$, then its production size is $K_e = \max_{1 \leq k \leq p, 1 \leq j \leq l} |q_j^k|$ The production size of a function symbol is the maximum production size of any equation defining a function with the same precedence: $K_{\mathtt{h}} = \max_{\mathtt{g} \approx_{\mathcal{F}} \mathtt{h}} \max_{e = \mathtt{g}(\dots) \to t} K_e$.

Definition 20 (Normality). Let $\mathtt{f}$ be a program. A function symbol $\mathtt{h}$ in it is *normal* if the patterns in the definitions of functions with the same precedence are bigger than its production size:

$$\forall \mathtt{g} \approx_{\mathcal{F}} \mathtt{h}, \forall \mathtt{g}(q_1, \cdots, q_m) \to r \in \mathcal{E}, |q_i| \geq K_{\mathtt{h}}$$

A program $\mathtt{f}$ is *normal* if all the function symbols in it are normal.

If a program is normal, this means that during recursive calls, every constructor produced at a given moment will be consumed by the following pattern matching.

Lemma 25. An EPPO-program can be changed into an equivalent normal program (*normalised*) with at most an exponential growth *in the size of the program.*

The exponential is in the difference between the size of the biggest production and the size of the smallest pattern (with respect to each precedence).

Proof (sketch) of Lemma 25. The idea is to extend the small pattern matching so that their length reaches the length of the biggest production. This is illustrated by the following example:

$$\begin{array}{rcl} \mathtt{f}(\mathbf{s}_1(\mathbf{s}_1(\mathbf{s}_1(x)))) & \to & \mathtt{f}(\mathbf{s}_0(\mathbf{s}_0(x))) \\ \mathtt{f}(\mathbf{s}_0(x)) & \to & \mathtt{f}(x) \end{array}$$

In this case, the biggest production has size 2 but the shortest pattern matching has only size 1. We can normalise the program as follows:

$$\begin{array}{rcl} \mathtt{f}(\mathbf{s}_1(\mathbf{s}_1(\mathbf{s}_1(x)))) & \to & \mathtt{f}(\mathbf{s}_0(\mathbf{s}_0(x))) \\ \mathtt{f}(\mathbf{s}_0(\mathbf{s}_0(x))) & \to & \mathtt{f}(\mathbf{s}_0(x)) \\ \mathtt{f}(\mathbf{s}_0(\mathbf{s}_1(x))) & \to & \mathtt{f}(\mathbf{s}_1(x)) \end{array}$$

Even if the process does extend productions as well as patterns and increases the number

of equations, it does terminate because only the smallest patterns, hence the smallest productions (due to termination ordering) are extended this way. □

Notice that the exponential growth in Lemma 25 is indeed in the initial size of the program and does not depend on the size of any input. Since the size of the call-dag is bounded by the size of the inputs, this does not hamper the polynomial bound on execution time. The normalisation process of Lemma 25 preserves termination by PPO and EPPO and does not decrease time complexity. Hence, bounding the time complexity of the normalised program is sufficient to bound the time complexity of the initial program. In the following, we only consider normal programs.

Let $\mathtt{f}$ be a program and $\mathtt{g}$ be a function, we will enumerate all the occurrences of symbols of same precedence as $\mathtt{g}$ in the rhs of $\mathtt{f}$ and label them $\mathtt{g}_1, \ldots, \mathtt{g}_n$. This is simply an enumeration, not a renaming of the symbols and if a given symbol appears several times (in several equations or in the same one), it will be given several labels (one for each occurrence) with this enumeration. Now, a path in the call-dag staying only at the same precedence as $\mathtt{g}$ is canonically identified by a word over $\{\mathtt{g}_1, \ldots, \mathtt{g}_n\}$. We write $\eta \overset{\omega}{\rightsquigarrow} \mu$, where ω is a word, to denote that the state η is an ancestor of μ and ω is the path between them.

Lemma 26. Let $\eta_1 = (\mathtt{f}, v_1, \cdots, v_n, v)$ and $\eta_2 = (\mathtt{g}, u_1, \cdots, u_m, u)$ be two states such that $\eta_1 \overset{e}{\rightsquigarrow} \eta_2$ and both function symbols have the same precedence. Then $|v_i| \geq |u_i|$ for $1 \leq i \leq n$ and there exists j such that $|v_j| > |u_j|$.

Proof. This is a consequence of the termination proof by EPPO. □

The following Proposition is a consequence of Lemma 26:

Proposition 27. Let $\eta = (\mathtt{f}, v_1, \cdots, v_n, v)$ be a state. Any branch in the call-dag starting from η has at most $n \times (\max_{1 \leq i \leq n} |v_i|)$ nodes with the same precedence as $\mathtt{f}$.

Lemma 28. Suppose that we have labels α, β and γ and nodes in the call dag such that $\eta \overset{\alpha}{\rightsquigarrow} \eta_1 \overset{\beta}{\rightsquigarrow} \mu_1 \overset{\gamma}{\rightsquigarrow} \xi_1$ and $\eta \overset{\beta}{\rightsquigarrow} \eta_2 \overset{\alpha}{\rightsquigarrow} \mu_2 \overset{\gamma}{\rightsquigarrow} \xi_2$. Then $\xi_1 = \xi_2$.

Proof. Let us write $\xi_1 = (\mathtt{f}, s_1, \ldots, s_n, s), \xi_2 = (\mathtt{g}, t_1, \ldots, t_m, t)$. Since labels are unique, we have $\mathtt{f} = \mathtt{g}$, and thus $n = m$. It is sufficient to show that the i-th components are the same for $1 \leq i \leq n$. Let us write as $v, v_1, u_1, q_1, v_2, u_2, q_2$ the ith parameters of $\eta, \eta_1, \mu_1, \xi_1, \eta_2, \mu_2, \xi_2$ respectively. Since we are working on words, an equation e has, with respect to the ith argument, the form:

$$\mathtt{f}(\ldots, \Omega_e(x), \ldots) \rightarrow C[\mathtt{g}(\ldots, \Upsilon_e(x), \ldots)]$$

and the fact that the program is normal implies that $|\Upsilon_e| \leq |\Omega_e|$ (previous lemma). Let us denote in the same way the equation corresponding to the label α:

$$\mathtt{f}(\ldots, \Omega_\alpha(x), \ldots) \rightarrow C[\mathtt{g}(\ldots, \Upsilon_\alpha(x), \ldots)]$$

In this case we write as a short-hand notation: $\Omega_\alpha(x) \hookrightarrow \Upsilon_\alpha(x)$. We define similarly

$\Omega_\beta, \Upsilon_\beta, \Omega_\gamma, \Upsilon_\gamma$. So, in our case, we have:

$$\begin{aligned} v &= \Omega_\alpha(x_1) \hookrightarrow \Upsilon_\alpha(x_1) = v_1 = \Omega_\beta(y_1) \hookrightarrow \Upsilon_\beta(y_1) \\ &= u_1 = \Omega_\gamma(z_1) \hookrightarrow \Upsilon_\gamma(z_1) = q_1 \\ v &= \Omega_\beta(x_2) \hookrightarrow \Upsilon_\beta(x_2) = v_1 = \Omega_\alpha(y_2) \hookrightarrow \Upsilon_\alpha(y_2) \\ &= u_2 = \Omega_\gamma(z_2) \hookrightarrow \Upsilon_\gamma(z_2) = q_2 \end{aligned}$$

Because of normalisation, $|\Omega_\beta| \geq |\Upsilon_\alpha|$. Hence y_1 is a suffix of x_1, itself a suffix of v. Similarly, z_1 and z_2 are suffixes of v. Since they are both suffixes of the same word, it is sufficient to show that they have the same length in order to show that they are identical.

$$\begin{aligned} |x_1| &= |v| - |\Omega_\alpha| \\ |y_1| &= |v| - |\Omega_\beta| = |v| - |\Omega_\alpha| + |\Upsilon_\alpha| - |\Omega_\beta| \\ |z_1| &= |v| - |\Omega_\alpha| + |\Upsilon_\alpha| - |\Omega_\beta| + |\Upsilon_\beta| - |\Omega_\gamma| \\ |z_2| &= |v| - |\Omega_\beta| + |\Upsilon_\beta| - |\Omega_\alpha| + |\Upsilon_\alpha| - |\Omega_\gamma| \end{aligned}$$

So $z_1 = z_2$ and thus $q_1 = q_2$. □

We say that two words have the *same commutative image* if there exists a permutation on the order of letters mapping the first word to the second one.

Proposition 29. Let ω_1, ω_2 be words and α be a label such that: $\eta \overset{\omega_1}{\rightsquigarrow} \mu_1 \overset{\alpha}{\rightsquigarrow} \xi_1$ and $\eta \overset{\omega_2}{\rightsquigarrow} \mu_2 \overset{\alpha}{\rightsquigarrow} \xi_2$. If ω_1 and ω_2 have the same commutative image, then $\xi_1 = \xi_2$.

Proof. This is a generalisation of the previous proof, not an induction on it. If $\omega_1 = \alpha_1 \dots \alpha_n$ then the size in the last node is:

$$|x'| = |q| - \sum |\Omega_{\alpha_i}| + \sum |\Upsilon_{\alpha_i}| - |\Omega_\alpha|$$

which only depends on the commutative image of ω_1. □

So, when using the semantics with memoisation, the number of nodes (at a given precedence) in the call-dag is roughly equal to the number of paths *modulo commutativity*. So any path can be associated with the vector whose components are the number of occurrences of the corresponding label in it.

Proposition 30. Let $\mathcal{T}$ be a call-dag and $\eta = (\mathtt{f}, v_1, \cdots, v_n, v)$ be a node in it. Let $I = n \times (\max |v_j|)$ and M be the number of function symbols with the same precedence as $\mathtt{f}$. The number of descendants of η in $\mathcal{T}$ with the same precedence as $\mathtt{f}$ is bounded by $(I+1)^M$, that is a polynomial in $|\eta|$.

Proof. Any descendant of η with the same precedence can be identified with a word over $\{\mathtt{f}^1, \dots, \mathtt{f}^M\}$. By Proposition 27 we know that these words have length at most I and by Proposition 29 we know that it is sufficient to consider these words modulo commutativity. Modulo commutativity, words can be identified to vectors with as many components as the number of letters in the alphabet and whose sum of components is equal to the length of the word. Let D^n_i be the number of elements from $\mathbb{N}^n$ whose sum is i. This is the number of words of length i over a n-ary alphabet modulo commutativity.

Clearly, $D^n_{i-1} \leq D^n_i$ (take all the n-tuple whose sum is $i-1$, add 1 to the first component, you obtain D^n_{i-1} different n-tuples whose sum of components is i). Now, to count D^n_i, proceed as follows: choose a value j for the first component, then you have to find $n-1$ components whose sum is $i-j$, there are D^{n-1}_{i-j} such elements.

$$D^n_i = \sum_{0 \leq j \leq i} D^{n-1}_{i-j} = \sum_{0 \leq j \leq i} D^{n-1}_j \leq (i+1) \times D^{n-1}_i \leq (i+1)^{n-1} \times D^1_i \leq (i+1)^n$$

This concludes the proof. □

Definition 21 (Bounded Values). A program $\mathtt{f} = \langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E} \rangle$ has polynomially bounded values iff for every n-ary function symbol $\mathtt{g} \in \mathcal{F}$, there is a polynomial $p_{\mathtt{g}} : \mathbb{N} \to \mathbb{N}$ such that for every state η' appearing in a call tree for $\eta = (\mathtt{g}, v_1, \ldots, v_n)$, we have $|\eta'| \leq p_{\mathtt{g}}(|\eta|)$.

Proposition 31. Any deterministic program `f` over unary constructors which (i) terminates by EPPO and (ii) has polynomially bounded values, is executable in polynomial time with the memoisation semantics. Conversely, any function in FP can be represented by a program of this kind.

Since admitting a QI implies having polynomially bounded values, we have: a program over unary constructors terminating by EPPO and admitting a QI is polytime.

Proof of Prop. 31. Proposition 30 bounds the size of the call dag by rank. The Bounded Values property bounds the size of nodes in the call dag. So we can apply Proposition 2 to bound the size of any derivation. The converse is obtained from the P-criterion (Theorem 12). □

Theorem 32. Let `f` be a deterministic program over unary constructors terminating by EPPO. Then the following two conditions are equivalent:

1. `f` has polynomially bounded values;
2. `f` is polytime in the call by value semantics with memoisation.

Note that the interest of this last result is rather theoretical, as we do not provide here a new way to determine if a program admits polynomially bounded values (admitting a QI is a sufficient condition). However it delineates exactly the EPPO programs over unary constructors which are polytime in the semantics with memoisation.

Proof of Theorem 32 The implication $1 \Rightarrow 2$ is given by Prop. 31. For the converse, it is sufficient to see that a state appearing in the call dag also appears in the final cache. Since the size of any term in the cache is bounded by the size of the proof (because we need to perform as many (Constructor) rules as needed to construct the term), it is polynomially bounded because the program is polytime. □

Actually we conjecture that the statements of theorems 31 and 32 hold not only for programs with unary constructors, but also for programs with arbitrary constructors, but we leave this question for future work.

Remark 2. The notion of *sup-interpretation* (Marion & Péchoux 2009, Marion & Péchoux

2006, Péchoux 2007) has been introduced to generalise that of quasi-interpretation. However, not all programs admitting a sup-interpretation satisfy the bounded values condition. Indeed sup-interpretations only relate the interpretation of a term with the interpretation of its normal form (see Def. 4.7, Section 4.2 (Marion & Péchoux 2009)), while the notion of program with polynomially bounded values requires that, in addition, all intermediates states of the call-tree are also bounded. Let us give a counter-example. Consider the program defined by:

$$\begin{array}{rcl} \mathtt{f}(0,y) & \to & 0 \\ \mathtt{f}(\mathtt{s}x,y) & \to & \mathtt{f}(x,\mathtt{db}(y)) \\ \mathtt{db}(0) & \to & 0 \\ \mathtt{db}(\mathtt{s}(x)) & \to & \mathtt{s}(\mathtt{s}(\mathtt{db}(x))) \end{array}$$

This program can be given a sup-interpretation for all its symbols:

$$\theta(\mathtt{f})(X,Y) = X; \qquad \theta(\mathtt{db})(X) = 2X;$$

and, as expected,

$$\theta(\mathtt{s})(X) = X+1; \qquad \theta(0) = 0.$$

However, the last call of $\mathtt{f}$ during the execution of $\mathtt{f}(\mathtt{s}^n(0),0)$ is $\mathtt{f}(0,\mathtt{s}^m(0))$ with $m = 2^n$, so it is exponentially large and the program does not have polynomially bounded values.

To handle this, sup-interpretations come together with the notions of fraternities (to detect recursive calls) and weight (which, having the subterm property of QI tame the recursive calls). As stated in Theorem 5.3 of (Marion & Péchoux 2009), programs with the *quasi-friendly* property (that is, sup-interpretation plus other conditions) also have the bounded values property.

On the other hand, any program satisfying the bounded values properties admits a sup-interpretation, that is to say a sup-interpretation can be attributed to all its functions symbols. More precisely these sup-interpretations can be chosen to be polynomials. It is sufficient for that to take for any function symbol $\mathtt{g}$ of the program a polynomial $p_{\mathtt{g}}$ as given by Definition 21.

8. Conclusions

We have introduced blind abstractions of first-order functional programs and exploited them in understanding the intensional expressive power of quasi–interpretations. In particular, we have shown that linear programs that admit product path-ordering and uniform quasi-interpretation are necessarily blindly polytime. This has suggested an extension of product path-ordering for which we have proved that the P-criterion still holds. Finally we have delineated a property (*polynomially bounded values*) which describes, among deterministic programs admitting product path-orderings, those executable in polynomial time.

We have introduced the notion of *blindly polytime* programs and shown that this notion can be used to compare ICC systems in a more general way than by exhibiting examples

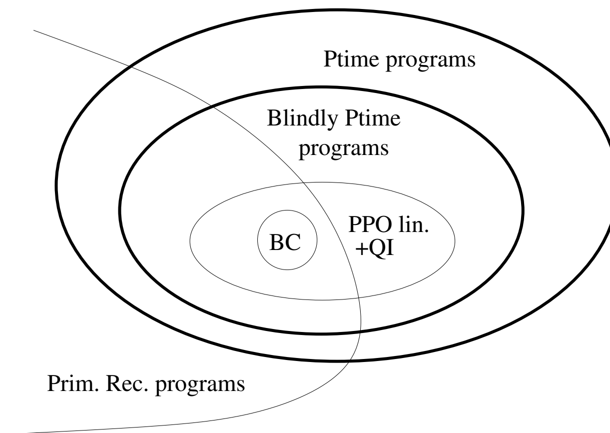


Fig. 8. Subclasses of programs

caught by one and not by the other. The comparison between the Bellantoni-Cook BC class and our "blind P-criterion" is illustrated on Figure 8.

Further work includes investigations on conditions characterising the class of programs captured by quasi-interpretation. In particular, the question is still open whether being blindly polytime is a necessary *and sufficient* condition for a program to have a (polynomially bounded) sup-interpretation or to be quasi-friendly (Marion & Péchoux 2009), provided some sort of path-ordering for it exists.

Another direction is to try to find notions similar to blindly polytime which would help comparing and classifying ICC systems. We have seen that being blindly polytime is a condition independent from the primitive recursion scheme. In order to better understand the relative power of ICC systems, it would be useful to have other similar classes of programs and complete the picture of Figure 8.

References

Amadio, R. (2005), 'Synthesis of max-plus quasi-interpretations', *Fundamenta Informaticae* **65**, 29–60.

Avanzini, M. & Moser, G. (2008), Complexity analysis by rewriting, *in* 'Proceedings of 9th International Symposium on Functional and Logic Programming', Vol. 4989 of *LNCS*, Springer, pp. 130–146.

Bellantoni, S. J. & Cook, S. A. (1992), 'A new recursion-theoretic characterization of the polytime functions', *Computational Complexity* **2**, 97–110.

Bonfante, G., Cichon, A., Marion, J.-Y. & Touzet, H. (2001), 'Algorithms with polynomial interpretation termination proof', *Journal of Functional Programming* **11**(1), 33–53.

Bonfante, G., Marion, J.-Y. & Moyen, J.-Y. (2001), On lexicographic termination ordering with space bound certifications, *in* 'Ershov Memorial Conference', Vol. 2244 of *LNCS*, Springer, pp. 482–493.

Bonfante, G., Marion, J.-Y. & Moyen, J.-Y. (2005), Quasi-Interpretations and Small Space Bounds, *in* 'Proceedings of The International Conference on Rewriting Techniques and Applications (RTA) 2005', Vol. 3467 of *LNCS*, Springer.

Bonfante, G., Marion, J.-Y. & Moyen, J.-Y. (2011), 'Quasi-interpretations a way to control resources', *Theoretical Computer Science* **412**(25), 2776 – 2796.

Bonfante, G., Marion, J.-Y. & Péchoux, R. (2007), Quasi-interpretation synthesis by decomposition, *in* 'Proceedings of the International Colloquium on Theoretical Aspects of Computing (ICTAC) 2007', Vol. 4711 of *LNCS*, Springer, pp. 410–424.

Dal Lago, U. (2007), Note on the intentional expressive power of bounded calculi. Manuscript available at `http://www.cs.unibo.it/~dallago/iepbc.pdf`.

Dershowitz, N. (1982), 'Orderings for term-rewriting systems', *Theoretical Computer Science* **17**(3), 279–301.

Girard, J.-Y. (1998), 'Light linear logic', *Information and Computation* **143**, 175–204.

Hofmann, M. (1999), Linear types and Non-Size Increasing polynomial time computation, *in* 'Proceedings of the Fourteenth IEEE Symposium on Logic in Computer Science (LICS) 99', pp. 464–473.

Hofmann, M. (2003), 'Linear types and non-size-increasing polynomial time computation', *Information and Computation* **183**(1), 57–85.

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

Jones, N. D. (1999), 'LOGSPACE and PTIME characterized by programming languages', *Theoretical Computer Science* **228**, 151–174.

Kamin, S. & Lévy, J.-J. (1980), Attempts for generalising the recursive path orderings., Technical report, Univerity of Illinois, Urbana Champaign. unpublished note. Accessible on `http://www.ens-lyon.fr/pierre.lescanne/not_accessible.html`.

Klop, J. W. & de Vrijer, R. (2003), *Term Rewriting Systems*, Cambridge University Press, chapter Examples of TRSs and special rewriting formats.

Leivant, D. (1994), Predicative recurrence and computational complexity I: word recurrence and poly-time, *in* 'Feasible Mathematics II', Birkhauser, pp. 320–343.

Leivant, D. & Marion, J.-Y. (1993), Lambda calculus characterizations of poly-time, *in* 'Proceedings of Typed Lambda Calculi and Applications (TLCA) 05', Vol. 664 of *LNCS*, Springer, pp. 274–288.

Marion, J.-Y. (2003), 'Analysing the implicit complexity of programs', *Information and Computation* **183**(1), 2–18.

Marion, J.-Y. & Moyen, J.-Y. (2000), Efficient First Order Functional Program Interpreter with Time Bound Certifications, *in* 'Proceedings of Logic for Programming Artificial Intelligence and Reasoning (LPAR) 00)', Vol. 1955 of *LNAI*, Springer, pp. 25–42.

Marion, J.-Y. & Péchoux, R. (2006), Resource analysis by sup-interpretation, *in* 'Proceedings of Symposium on Functional and Logic Programming (FLOPS) 06', Vol. 3945 of *LNCS*, Springer, pp. 163–176.

Marion, J.-Y. & Péchoux, R. (2009), 'Sup-interpretations, a semantic method for static analysis of program resources', *ACM Transactions on Computational Logic (TOCL)* **10**(4).

Moyen, J.-Y. (2003), Analyse de la complexité et transformation de programmes, Thèse d'université (PhD thesis), University Nancy 2.

Péchoux, R. (2007), Analyse de la complexité des programmes par interprétation sémantique, PhD thesis, Institut National Polytechnique de Lorraine.