Math. Struct. in Comp. Science (2012), *vol.* 22, *pp.* 549–580. © Cambridge University Press 2012
doi:10.1017/S0960129511000685 First published online 2 February 2012

On quasi-interpretations, blind abstractions and implicit complexity[†]

PATRICK BAILLOT[‡], UGO DAL LAGO[§] and JEAN-YVES MOYEN[¶]

[‡]*LIP, UMR 5668 CNRS, ENS-Lyon, UCBL, INRIA, Université de Lyon, France*
Email: patrick.baillot@ens-lyon.fr
[§]*Università di Bologna, Italy*
Email: dallago@cs.unibo.it
[¶]*LIPN, UMR 7030 CNRS, Université Paris 13, France*
Email: jean-yves.moyen@lipn.univ-paris13.fr

Received 23 July 2008; revised 23 September 2011

Quasi-interpretations are a technique for guaranteeing complexity bounds on first-order functional programs: in particular, with termination orderings, they give a sufficient condition for a program to be executable in polynomial time (Marion and Moyen 2000), which we call the P-criterion here. 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 relaxing the termination ordering condition and the other (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

Implicit computational complexity (ICC) explores machine-free characterisations of complexity classes, without referring to explicit resource bounds but instead viewing them as consequences of restrictions on program structures. For the most part, ICC has been developed in the functional programming paradigm by taking advantage of ideas from primitive recursion (Bellantoni and Cook 1992; Leivant 1994), proof-theory and linear logic (Girard 1998), rewriting systems and functional programming (Jones 1999; Bonfante *et al.* 2001a; Bonfante *et al.* 2001b; Marion 2003; Bonfante *et al.* 2011), type systems (Leivant and Marion 1993; Hofmann 1999), and so on.

ICC results usually include both a soundness and a completeness statement: the former saying that all programs of a given language (or those satisfying a criterion) satisfy a certain quantitative property; the latter saying that all *functions* (or *problems*) of the

[†] This work was partially supported by projects CRISS (ACI), NO-CoST (ANR, JC05_43380).

https://doi.org/10.1017/S0960129511000685 Published online by Cambridge University Press

corresponding functional complexity class can be programmed in this language. For instance, in the case of polynomial time complexity, soundness refers to the possibility of evaluating programs in polynomial time, whereas completeness, which has 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, including ramified recursion (Bellantoni and Cook 1992; Leivant 1994), variants of linear logic (Girard 1998) and fragments of functional languages (Jones 1999). Our main aim here is *not* just to define another characterisation of polynomial time computable functions, but rather to begin a study of *existing* characterisations in order to understand their properties better – intensional expressivity *in primis*.

Expressivity

One of the main motivations for ICC comes from programming language theory, because ICC suggests ways to control the 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 that the system corresponds extensionally to a complexity class. This issue has been pointed out by several authors (Marion and 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 *et al.* 2001b; Bonfante *et al.* 2011). However, it is not always easy to measure the improvements offered by a new ICC system, which are usually just illustrated by providing examples, though some experiments (Avanzini and Moser 2008) have been done to compare some rewriting-based criteria on data bases of examples from term rewriting.

We think that in order 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 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. To do this, 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 correspond exactly to hereditarily polytime primitive recursive programs. In the current paper, 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

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

https://doi.org/10.1017/S0960129511000685 Published online by Cambridge University Press

rewriting systems. Used with termination orderings, they allow us to define various criteria to guarantee either space or time complexity bounds (Bonfante *et al.* 2005; Amadio 2005; Bonfante *et al.* 2007; Bonfante *et al.* 2011). QIs offer several advantages: in particular, 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, which we will call the *P-criterion* here, 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, we present a program transformation called *blind abstraction*, which collapses the constructors of a given arity to a single constructor, and modifies the rewriting rules accordingly. In general, this produces non-confluent programs, the efficiency of which can be measured by considering all possible evaluations of the program.

The blind abstraction of a polytime first-order functional program is not itself polytime in general: 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 of non-confluence.

We cannot claim this result to be a breakthrough, but it does represent one of the 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.

Outline of the paper

We begin in Section 2 by describing the syntax and operational semantics of programs, followed by termination orderings and QIs in Section 3. We introduce blind abstractions in Section 4 and give the main property of the P-criterion (with respect to blinding) in Section 5, with applications to safe recursion. In Section 6, we consider a particular class of QIs, the compatible QIs, and study their properties. Finally, in Section 7, we define a generalisation of the previous termination ordering and of QIs (the bounded-values property), which also guarantees polytime soundness.

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

2. Programs as term rewriting systems

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

https://doi.org/10.1017/S0960129511000685 Published online by Cambridge University Press

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

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

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

We should stress that we are actually interested in the study of deterministic programs, but as soon as we consider abstractions of programs, we will obtain non-deterministic rewriting systems, and, in order to analyse them, we will need our definitions and results to handle non-deterministic systems.

2.1. *The syntax and semantics of programs*

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

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

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

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 of an equation also appears in the left-hand side of the same equation. The program has a main function symbol in $\mathcal{F}$, which we shall also call $\mathtt{f}$.

In other words, our programs are constructor (term-rewriting) systems – see the *Examples of TRSs and special rewriting formats* chapter in Klop and 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 use $\mathfrak{S}$ to denote the set of *constructor substitutions*, that is, substitutions σ with range $\mathcal{T}(\mathcal{C})$. Our programs are not necessarily deterministic, in other words, the TRS is not necessarily confluent. Non-confluent programs will be interpreted here by relations.

We shall begin by considering a call-by-value semantics, which is defined 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:

https://doi.org/10.1017/S0960129511000685 Published online by Cambridge University Press

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

Fig. 2. Example of a reduction proof.

— for all $u_i \in \mathscr{T}(\mathscr{C})$, we have $v \in [\![\mathtt{f}]\!](u_1,\cdots,u_n)$ if and only if 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.

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 shows a derivation π.

The 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

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

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

Notice that the set of active terms in a derivation π is exactly the set of terms of the form $\mathtt{f}(v_1,\cdots,v_n)$ appearing in π, where v_i are constructor terms. 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 just on s.

Our aim now is to prove 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 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 $|\pi|\leqslant p(A,S)$.

https://doi.org/10.1017/S0960129511000685 Published online by Cambridge University Press

Proof. First, 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, we have $|s| \leqslant r(|t|)$. That is, the size of s is polynomially bounded by the size of t. This is 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 that is immediately below H. In this way we can associate with any active judgement J in π a set of passive judgements D_J such that any passive judgement is in some D_J. We will now 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 left-hand side of any passive judgements in D_J is a subterm of the left-hand side of H, so 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 left-hand side 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 left-hand side of J. We now note that if q is the right-hand side of any passive judgement in π, then $|q| \leqslant \max\{S, |u|\}$, where u is the right-hand side of the conclusion of π (this follows by induction on π). As a consequence, $|q| \leqslant S$, because the right-hand side of the conclusion of π is part of an active judgement itself. Putting everything together, we get

$$|\pi| \leqslant (A + A \cdot r(S))(r(S) + S),$$

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

We will now consider a call-by-value semantics with memoisation for confluent programs, following Marion and 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, 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 defined 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 together with all the calls required 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 the judgements J in π.

https://doi.org/10.1017/S0960129511000685 Published online by Cambridge University Press

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

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

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

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

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

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

Fig. 4. Example of a reduction proof.

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

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

We also need to classify the rules, terms and judgements in the memoisation semantics.

Definition 4 (Classifying terms, rules and judgements). Constructor and Split are *passive* semantic rules; Update is an *active* rule; and Read is a *semi-active* rule. Active terms and judgements, activated equations, activations and dependencies are defined in a similar way to 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 proofs $\pi : \langle \varnothing, \mathtt{f}(v_1, \cdots, v_n)\rangle \Downarrow \langle C, u\rangle$, if A is the number of distinct active judgements in π and S is the maximum size of an active judgement in π, then $|\pi| \leqslant p(A, S)$.

https://doi.org/10.1017/S0960129511000685 Published online by Cambridge University Press

Proof. We first observe 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-leaf 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 by considering 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 $\pi : \langle \varnothing, t\rangle \Downarrow \langle C, v\rangle$. 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. The proof follows from the fact that 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 problems, so we will not use any of them here and only use memoisation when the program is confluent. For sufficient conditions to check that a program is confluent, see, for instance, Huet's work (Huet 1980).

2.2. *Call trees and call dags*

Following Bonfante *et al.* (2011), we will now present call-trees, which we shall use as a tool throughout the paper. Note that this is not a new kind of semantics for programs but an analysis tool, which we will use to study both the call-by-value 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 $\mathscr{E}$,
(2) there is a substitution σ such that $p_i\sigma = v_i$ for all $1 \leqslant i \leqslant n$,

https://doi.org/10.1017/S0960129511000685 Published online by Cambridge University Press

(3) there is a subterm $g(s_1, \cdots, s_m)$ of t such that $s_i\sigma \downarrow u_i$ for all $1 \leqslant i \leqslant m$.

We write $\overset{*}{\rightsquigarrow}$ for the reflexive transitive closure of $\cup_{e\in\mathscr{E}} \overset{e}{\rightsquigarrow}$.

The above definition of a state is slightly different from, though essentially equivalent to, the one given in 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.

In other words, if t is passive, that is,

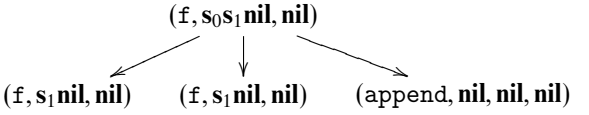
$$\frac{b \in \mathscr{F} \bigcup \mathscr{C} \quad \pi_i : t_i \downarrow v_i}{b(t_1, \cdots, t_n) \downarrow b(v_1, \cdots, v_n)} \text{ (Constructor) or (Split)},$$

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

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

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

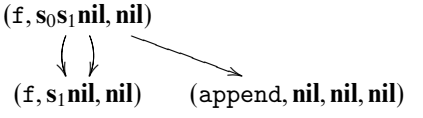
The following is the only element of Θ_π, where π is the reduction proof in Figure 2:



When using the memoisation semantics, some function calls are not recomputed but instead fetched from the cache, so 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 Definition 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 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 $\mathscr{T}$ of Θ_π, all nodes in $\mathscr{T}$ have at most k children.

Proof. For each right-hand side term r of an equation of f, consider the number of maximal subterms of r with a function as head symbol. We can then take k to be the maximum of these integers over the (finite) set of equations of f. □

https://doi.org/10.1017/S0960129511000685 Published online by Cambridge University Press

Lemma 4. Consider the call-by-value semantics. Let π be a reduction proof, $\mathscr{T}$ 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 $\eta \overset{e}{\rightsquigarrow} \mu$. Conversely, if $\eta \overset{e}{\rightsquigarrow} \mu$, then μ is a child of η in $\mathscr{T}$.

Proof. The proof is by 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 (in either the call-by-value or memoisation semantics) takes into account all the symbol occurrences in π: in this way, it can be considered to be a reasonable approximation to the 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 $\mathtt{f}$:

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 $|\pi| \leqslant p_\mathtt{f}(a_\pi, s_\pi)$.

Memoisation semantics : Assume $\mathtt{f}$ is deterministic. There is a polynomial $q_\mathtt{f} : \mathbb{N}^2 \to \mathbb{N}$ such that whenever $\pi : \langle \varnothing, \mathtt{f}(u_1, \ldots, u_n)\rangle \Downarrow \langle C, v\rangle$ 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 $|\pi| \leqslant q_\mathtt{f}(a_\pi, s_\pi)$.

Proof. This is an easy corollary of Propositions 1 and 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 gives a polynomial bound on the execution time.

3. Orderings and quasi-interpretations

This section presents a new way of looking at concepts and results that have already appeared in the literature (Bonfante *et al.* 2011).

3.1. *Termination orderings*

Definition 8 (Precedence). Let $\mathtt{f} = \langle \mathscr{X}, \mathscr{C}, \mathscr{F}, \mathscr{E}\rangle$ be a program. A *precedence* $\preceq_\mathscr{F}$ is a preorder over $\mathscr{F} \bigcup \mathscr{C}$. We use $\approx_\mathscr{F}$ to denote 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, we have $b \preceq_\mathscr{F} \mathtt{g}$, and it is *separating* if for each $\mathbf{c} \in \mathscr{C}, \mathtt{f} \in \mathscr{F}$, $\mathbf{c} \prec_\mathscr{F} \mathtt{f}$

https://doi.org/10.1017/S0960129511000685 Published online by Cambridge University Press

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

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

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

(that is, constructors are the smallest elements of $\prec_{\mathcal{F}}$, and functions are the biggest). It is *fair* if for all constructors **c** and **d** with the same arity, we have $\mathbf{c} \approx_{\mathcal{F}} \mathbf{d}$, and it is *strict* if for all distinct constructors **c** and **d**, we have **c** and **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 (respectively, fair) and separating, then the ordering is the *Product Path Ordering* PPO (respectively, the *Extended Product Path Ordering* EPPO).

Of course, we could consider other ordering extensions. The usual choices are the lexicographic extension, 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 and Lévy 1980) indicating with which extension the parameters must be compared. This leads to the more general Recursive Path Ordering (RPO). However, here we will only use the (extended) Product Path Ordering, and will not describe any of the others.

Given a precedence $\preceq_{\mathcal{F}}$, an equation $l \to r$ is *decreasing* with respect 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 with respect to $\preceq_{\mathcal{F}}$. 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 $\mathtt{f}$ 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 (respectively, 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. The claim follows from the fact that the precedence is compatible with $\mathtt{f}$. □

Proposition 7 (Computing by rank). Let $\mathtt{f}$ 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

https://doi.org/10.1017/S0960129511000685 Published online by Cambridge University Press

descendants of a node with the same precedence:

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

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

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

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

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

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

So B_i is polynomially bounded in $A_{\mathscr{T}}$, as is the size of the call tree (dag).

This concludes the proof. □

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

Proposition 8. Let $\mathtt{f}$ be a program terminating by PPO, $\mathscr{T}$ be a call dag and $\eta = (\mathtt{f}, v_1, \cdots, v_n, v)$ be a node in $\mathscr{T}$. The number of descendants of η in $\mathscr{T}$ 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 $\mathtt{f} \approx_{\mathscr{F}} \mathtt{g}$, then u_i is a subterm of some v_j. There are at most $|v_j|$ subterms of v_j, so there are at most $c \cdot d \cdot \max |v_i|$ 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 Bonfante *et al.* (2011, Lemma 51). Notice that this 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 $b \in \mathscr{F} \bigcup \mathscr{C}$ whose arity is n is a function $(\!|b|\!) : (\mathbb{R})^n \to \mathbb{R}$ such that:

https://doi.org/10.1017/S0960129511000685 Published online by Cambridge University Press

Subterm

$(\!|b|\!)(X_1, \cdots, X_n) \geqslant X_i$ for all $1 \leqslant i \leqslant n$.

Weak monotonicity

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

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

Additivity

$(\!|\mathbf{c}|\!)(X_1, \cdots, X_n) = \sum_{i=1}^{n} X_i + a$ if $\mathbf{c} \in \mathscr{C}$ (where $a \geqslant 1$).

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

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 $l \to r$,

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

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 $|v| \leqslant (\!|v|\!) \leqslant a|v|$.

Proof. By induction, we can show that 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, $(\!|\mathtt{g}(u_1, \cdots, u_m)|\!) \leqslant (\!|\mathtt{f}(v_1, \cdots, v_n)|\!)$.

Proof. The claim follows from the fact that $\mathtt{g}(u_1, \cdots, u_m)$ 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 $\pi : \langle \varnothing, \mathtt{f}(v_1, \cdots, v_n) \rangle \Downarrow \langle D, v \rangle$, 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

$$s = \mathtt{g}(u_1, \cdots, u_m)$$

https://doi.org/10.1017/S0960129511000685 Published online by Cambridge University Press

is bounded by

$$m \cdot \max |u_i| + 1 \leqslant m \cdot \max(\langle u_i \rangle) + 1 \leqslant m \cdot \langle s \rangle + 1.$$

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

If we now 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. In fact, all polytime functions can be obtained in this way.

Theorem 12 (P-criterion, Bonfante *et al.* (2011)). Any deterministic program $\mathtt{f}$ that

(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 with 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, and so on. Note that in this way, even if $\mathtt{f}$ is deterministic, the associated $\overline{\mathtt{f}}$ will not be deterministic in general.

To do this we will first define a target language:

Variables

$\overline{\mathscr{X}} = \mathscr{X}$,

Function symbols

We define the map $\overline{(.)}$ on function symbols by $\overline{\mathtt{f}} = \overline{\mathtt{g}}$ if and only if $\mathtt{f} = \mathtt{g}$. Then $\overline{\mathscr{F}} = \{\overline{\mathtt{f}} \ / \ \mathtt{f} \in \mathscr{F}\}$.

Constructor symbols

We define the map $\overline{(.)}$ on constructor symbols by $\overline{\mathbf{c}} = \overline{\mathbf{d}}$ if and only if $\mathbf{c}$ and $\mathbf{d}$ have the same arity. Then $\overline{\mathscr{C}} = \{\overline{\mathbf{c}} \ / \ \mathbf{c} \in \mathscr{C}\}$.

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

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

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 $\overline{\mathbf{nil}} = \mathbf{0}$.

https://doi.org/10.1017/S0960129511000685 Published online by Cambridge University Press

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

Fig. 6. Blind abstraction of our running example.

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

We have mentioned strings over a binary alphabet, but note that on some other data-types, such as 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 will 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 $p_{\mathtt{f}} : \mathbb{N}^n \to \mathbb{N}$ such that for any sequence $v_1, \cdots, v_n$ and any $\pi : \mathtt{f}(v_1, \cdots, v_n) \downarrow u$, we have $|\pi| \leqslant p_{\mathtt{f}}(v_1, \ldots, v_n)$.

Of course, similar definitions would also make sense for complexity bounds other 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 $\mathtt{f}$ is *blindly polytime* if its blind abstraction $\overline{\mathtt{f}}$ is strongly polytime.

Fact 2. If a program $\mathtt{f}$ is blindly polytime, it is polynomial time in the call-by-value semantics.

Indeed, it is sufficient to observe that any execution (derivation proof) of $\mathtt{f}$ can be mapped by $\mathscr{B}$ to an execution of $\overline{\mathtt{f}}$. But, of course, not all $\overline{\mathtt{f}}$ executions are obtained in this way. Note that for our running example in Figure 6, $\mathtt{f}$ terminates in polynomial time, but this is not the case for $\overline{\mathtt{f}}$. Indeed, if we use the denotation

$$\underline{n} = \underbrace{\mathbf{s} \ldots \mathbf{s}}_{n}\ \mathbf{0},$$

we have that $\overline{\mathtt{f}}(\underline{n})$ can be reduced in an exponential number of steps, with a $\pi : \overline{\mathtt{f}}(\underline{n}) \downarrow 0$.

Note that the property of being blindly polytime is indeed a strong condition since it means, in some sense, that the program will terminate with a polynomial bound for

https://doi.org/10.1017/S0960129511000685 Published online by Cambridge University Press

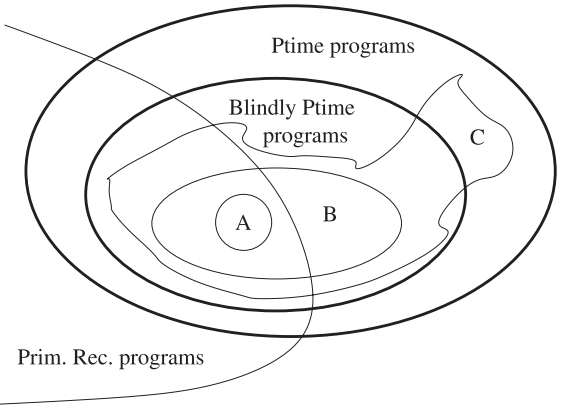


Fig. 7. Subclasses of programs.

reasons that are indifferent to the actual content of the input but only depend on its size. For instance, one can check 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, a 0-bit in a string is replaced by a 1-bit): blindly polytime programs are then programs that remain polynomial time, no matter how many errors occur during the execution.

Our motivation for introducing blind abstraction and the notion of a blindly polytime program is to provide a more abstract method for comparing different ICC systems than just simply providing examples handled by one system but not by the other. Indeed, this kind of abstraction gives a way to subdivide the class of polytime programs into smaller classes.

To illustrate our idea, consider the picture given in 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 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. Figures 8 and 9 list two programs for sorting binary strings. The first uses insertion sort while the second uses bubble sort. In the first case, since we are working on binary strings, 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 a boolean value as a flag to check whether a permutation occurred or not. This is a quite straightforward extension of the language. In `bsort`,

https://doi.org/10.1017/S0960129511000685 Published online by Cambridge University Press

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

Fig. 8. Insertion sort program.

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

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

Fig. 9. Bubble sort program.

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.

Both programs are polytime. The first terminates by PPO and admits a QI, and hence satisfies the P-criterion. However, it does not follow the safe recursion schemata (Bellantoni and 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.

On the other hand, the bubble sort program 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. In fact, the same comparison can be done several times, and the termination of the algorithm is due to the fact that the same

https://doi.org/10.1017/S0960129511000685 Published online by Cambridge University Press

comparison always yields the same result. This is no longer true after blinding the program because the comparison between elements of the string then becomes non-deterministic.

We will now 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*

We want to examine what happens with the ordering through the blinding operation. We have the following lemma.

Lemma 13. Let $\mathtt{f}$ be a program. If $\mathtt{f}$ terminates by PPO, then $\overline{\mathtt{f}}$ terminates by PPO.

Indeed, $\mathscr{F}$ and $\overline{\mathscr{F}}$ are in one–one correspondence, and it is easy to see that if $\prec_{\mathscr{F}}$ is a precedence that gives a PPO ordering for $\mathtt{f}$, then the corresponding $\prec_{\overline{\mathscr{F}}}$ does the same for $\overline{\mathtt{f}}$.

The converse is not true – see, for example, Figure 6 where the first equation terminates by PPO on the blind side but not on the non-blind side. However, we do have the following proposition.

Proposition 14. Let f be a program. The following statements are equivalent:

(i) $\mathtt{f}$ terminates by EPPO.
(ii) $\overline{\mathtt{f}}$ terminates by EPPO.
(iii) $\overline{\mathtt{f}}$ terminates by PPO.

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

4.4. *Blinding and quasi-interpretations*

Assume the program $\mathtt{f}$ admits a quasi-interpretation $(\!|.|\!)$. Then, in general, this does not imply that $\overline{\mathtt{f}}$ admits a quasi-interpretation. Indeed one reason why $(\!|.|\!)$ cannot be simply converted into a quasi-interpretation for $\overline{\mathtt{f}}$ is because, in general, a quasi-interpretation might 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 $\overline{\mathtt{f}}$, there is no natural choice for $(\!|\mathbf{s}|\!)$.

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

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

We now have the following proposition.

Proposition 15. The program $\mathtt{f}$ admits a uniform quasi-interpretation *if and only if* $\overline{\mathtt{f}}$ admits a quasi-interpretation.

https://doi.org/10.1017/S0960129511000685 Published online by Cambridge University Press

5. Linear programs and call-by-value semantics

5.1. *Definitions and main property*

We want to use the blinding transformation to examine the properties of programs satisfying the P-criterion (Theorem 12). In fact, when we do this, we will restrict our attention to particular programs, specifically, *linear* programs. Essentially, this means that only one recursive call is allowed in recursive definitions (equations). This is a fairly natural restriction when we want to control the complexity since it prevents 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). We will now give a formal definition.

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

We will not mention the precedence $\preceq_{\mathscr{F}}$ explicitly when there is no ambiguity.

Proposition 16. Let $\mathtt{f}$ be a (possibly non-deterministic) program that:

(i) terminates by PPO;
(ii) admits a quasi-interpretation; and
(iii) is linear.

Then $\mathtt{f}$ is strongly polytime.

Before giving the proof, we note that compared with Theorem 12 (the P-criterion Theorem from Bonfante *et al.* (2011)), the program here does not need to be deterministic, though 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 when there is non-determinism (see Section 2.1); linearity is a sufficient condition to avoid the use of memoisation and maintain the complexity properties.

Proof. 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 $\mathscr{T}$ be a call tree associated to π and $\eta = (\mathtt{f}, v_1, \cdots, v_n, v)$ be a node in it. Now consider the subset of nodes of $\mathscr{T}$ obtained by keeping η, the sons of η with same precedence as $\mathtt{f}$ 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 the same precedence as $\mathtt{f}$, and the linearity of the program actually implies that this set is a branch; so the number of its 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 $\mathtt{f} \approx_{\mathscr{F}} \mathtt{g}$, then $|u_i| \leqslant |v_i|$, and 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

https://doi.org/10.1017/S0960129511000685 Published online by Cambridge University Press

we can now use Proposition 1 and conclude that $|\pi|$ is polynomially bounded in A, S, and thus also in $|v_1|, \ldots, |v_n|$. Therefore $\mathtt{f}$ is strongly polytime. □

Theorem 17. Let $\mathtt{f}$ be a (possibly non-deterministic) program that:

(i) terminates by PPO;
(ii) admits a *uniform* quasi-interpretation; and
(iii) is linear.

Then $\mathtt{f}$ is blindly polytime.

Proof. Note that:

— $\mathtt{f}$ terminates by PPO, so, by Lemma 13, $\overline{\mathtt{f}}$ also terminates;
— the quasi-interpretation for $\mathtt{f}$ is uniform, so, by Proposition 15, $\overline{\mathtt{f}}$ admits a quasi-interpretation;
— $\mathtt{f}$ is linear, so $\overline{\mathtt{f}}$ is linear too.

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

5.2. *Bellantoni–Cook programs*

To show an example application of Theorem 17 to a simple class of programs, we will consider the class of Bellantoni–Cook safe recursive programs (Bellantoni and Cook 1992). Of course, this is a very particular case, but we think it is interesting since 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 classified as 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

$$\mathbf{s}_i(x), i \in \{0, 1\}$$

Projection

$$\pi_j^{n,m}(x_1, \cdots, x_n; x_{n+1}, \cdots, x_{n+m}) \to x_j$$

Predecessor

$$\mathrm{p}(;\mathbf{0}) \to \mathbf{0}$$
$$p(;\mathbf{s}_i(x)) \to x$$

https://doi.org/10.1017/S0960129511000685 Published online by Cambridge University Press

Conditional

$$\begin{aligned}\mathtt{C}(;\mathbf{0},x,y) &\to x\\ \mathtt{C}(;\mathbf{s}_0(z),x,y) &\to x\\ \mathtt{C}(;\mathbf{s}_1(z),x,y) &\to y,\end{aligned}$$

and closed by:

Safe recursion

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

with $\mathtt{g},\mathtt{h}_i \in$ BC (previously defined)

Safe composition

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

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:

$$\begin{aligned}(\!|\mathbf{0}|\!) &= 1\\ (\!|\mathbf{s}_i|\!)(X) &= X+1\\ (\!|\pi|\!)(X_1,\cdots,X_{n+m}) &= \max(X_1,\cdots,X_{n+m})\\ (\!|\mathtt{p}|\!)(X) &= X\\ (\!|\mathtt{C}|\!)(X,Y,Z) &= \max(X,Y,Z).\end{aligned}$$

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:

— if f is defined by safe recursion, then

$$q_{\mathtt{f}}(A,X_1,\cdots,X_n) = A(q_{\mathtt{h}_0}(A,X_1,\cdots,X_n)+q_{\mathtt{h}_1}(A,X_1,\cdots,X_n))+q_{\mathtt{g}}(X_1,\cdots,X_n);$$

— if f is defined by safe composition, then

$$q_{\mathtt{f}}(X_1,\cdots,X_n) = q_{\mathtt{g}}(q_{\mathtt{h}_1}(X_1,\cdots,X_n),\ldots,q_{\mathtt{h}_p}(X_1,\cdots,X_n))+\sum_i q_{\mathtt{l}_i}(X_1,\cdots,X_n).$$

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

https://doi.org/10.1017/S0960129511000685 Published online by Cambridge University Press

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 we should consider quasi-interpretations with fixed assignments for constructors and 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 $\mathbf{c}$ we have $(\!|\mathbf{c}|\!)_1 = (\!|\mathbf{c}|\!)_2$. A family of assignments for `f` is compatible if its elements are pairwise compatible. We will use similar 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` that take these values on $\mathscr{C}$. We consider the extensional order $\leqslant$ on assignments defined by $(\!|\cdot|\!)_1 \leqslant (\!|\cdot|\!)_2$ if and only if

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

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

$$\begin{aligned} \forall \mathbf{c} \in \mathscr{C}, \quad (\!|\mathbf{c}|\!)_0 &= (\!|\mathbf{c}|\!)_1 = (\!|\mathbf{c}|\!)_2 \\ \forall \mathrm{g} \in \mathscr{F}, \quad (\!|\mathrm{g}|\!)_0 &= (\!|\mathrm{g}|\!)_1 \wedge (\!|\mathrm{g}|\!)_2 \end{aligned}$$

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

Proposition 19. Let `f` be a program and let $(\!|\cdot|\!)_1$ and $(\!|\cdot|\!)_2$ 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 some preliminary Lemmas. We will continue to use the denotation $(\!|\cdot|\!)_0 = (\!|\cdot|\!)_1 \wedge (\!|\cdot|\!)_2$.

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

Proof. To prove monotonicity, we will assume $\vec{x} \leqslant \vec{y}$ for the product ordering. Then, for $i = 1$ or 2

$$(\!|\mathrm{g}|\!)_0(\vec{x}) = (\!|\mathrm{g}|\!)_1(\vec{x}) \wedge (\!|\mathrm{g}|\!)_2(\vec{x}) \leqslant (\!|\mathrm{g}|\!)_i(\vec{x}) \leqslant (\!|\mathrm{g}|\!)_i(\vec{y}),$$

using the monotonicity of $(\!|\mathrm{g}|\!)_i$. Since this is true for $i = 1$ and 2, we have

$$(\!|\mathrm{g}|\!)_0(\vec{x}) \leqslant (\!|\mathrm{g}|\!)_1(\vec{y}) \wedge (\!|\mathrm{g}|\!)_2(\vec{y}) = (\!|\mathrm{g}|\!)_0(\vec{y}).$$

It is also easy to check that $(\!|\mathrm{g}|\!)_0$ satisfies the subterm property. □

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

Proof. The proof is by induction on t, using the definition of $(\!|\mathrm{g}|\!)_0$ and $(\!|\mathbf{c}|\!)_0$, together with the monotonicity property of Lemma 20. □

https://doi.org/10.1017/S0960129511000685 Published online by Cambridge University Press

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

$$(\!|g|\!)_0 \circ ((\!|p_1|\!)_0, \ldots, (\!|p_n|\!)_0) \geqslant (\!|t|\!)_0.$$

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

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

So

$$(\!|g|\!)_i((\!|p_1|\!)_0(\vec{x}), \ldots, (\!|p_n|\!)_0(\vec{x})) \geqslant (\!|t|\!)_0(\vec{x}),$$

since $(\!|p_j|\!)_i = (\!|p_j|\!)_0$. We write

$$\vec{y} = ((\!|p_1|\!)_0(\vec{x}), \ldots, (\!|p_n|\!)_0(\vec{x})).$$

Since $(\!|g|\!)_1(\vec{y}) \geqslant (\!|t|\!)_0(\vec{x})$ and $(\!|g|\!)_2(\vec{y}) \geqslant (\!|t|\!)_0(\vec{x})$, by the definition of $(\!|g|\!)_0$, we get $(\!|g|\!)_0(\vec{y}) \geqslant (\!|t|\!)_0(\vec{x})$, which concludes the proof. □

We can now proceed with the proof of Proposition 19.

Proof of Proposition 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 Subterm conditions, and Lemma 22 ensures it satisfies the condition with respect to the equations of the program. The Additivity condition is satisfied because $(\!|\cdot|\!)_1$ and $(\!|\cdot|\!)_2$ are compatible and satisfy this condition by assumption. Finally, the Polynomial condition 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 f be a program and $\mathcal{Q}$ be a non-empty family of compatible quasi-interpretations for f. Then $\wedge_{(\!|\cdot|\!) \in \mathcal{Q}} (\!|\cdot|\!)$ is a quasi-interpretation for f. Therefore, maximal compatible families of quasi-interpretations for f have an inferior semi-lattice structure for $\leqslant$.

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 in the proof of Proposition 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 to $(\!|\cdot|\!)_1 \wedge (\!|\cdot|\!)_2$ 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

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

https://doi.org/10.1017/S0960129511000685 Published online by Cambridge University Press

Now consider $(\!|\cdot|\!)_1$ and $(\!|\cdot|\!)_2$ given by

$$(\!|\mathtt{g}|\!)_1(x) = x$$
$$(\!|\mathtt{f}|\!)_1(x) = x + 1$$
$$(\!|\mathtt{h}|\!)_1(x) = x$$
$$(\!|\mathtt{g}|\!)_2(x) = x$$
$$(\!|\mathtt{f}|\!)_2(x) = x$$
$$(\!|\mathtt{h}|\!)_2(x) = x + 1.$$

Thus we have,

$$(\!|\mathtt{g}|\!)(x) = x$$
$$(\!|\mathtt{f}|\!)(x) = x + 1$$
$$(\!|\mathtt{h}|\!)(x) = x + 1.$$

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 we consider not only the PPO ordering from the P-criterion, but also an extension that is invariant under the blinding map, *viz.* 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. 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, and then Proposition 7 bounds the total number of nodes in the call-dag.

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

Example 2. We will first show that EPPO is (intensionally) strictly more powerful than PPO. It is quite obvious that any program terminating by PPO also terminates by 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, in other words, 2). The program computes (in a slightly contrived way) x^y using 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 will omit the base cases here). The rules for multiplication and squaring are quite standard, so they are not included here. The equation in parentheses following each rule explains what it means.

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

https://doi.org/10.1017/S0960129511000685 Published online by Cambridge University Press

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 $1 \leqslant i \leqslant n$ 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, so $|p| = |\Omega|$.

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

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

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

Proof. The claim 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 $\mathtt{f} \approx_{\mathscr{F}} \mathtt{g}$. The *production size* of this call is $\max_{1 \leqslant i \leqslant m}\{|q_i|\}$. 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 $\mathtt{f} \approx_{\mathscr{F}} \mathtt{g}_k$ for $1 \leqslant k \leqslant p$, then its production size is

$$K_e = \max_{1 \leqslant k \leqslant p, 1 \leqslant j \leqslant l} |q_j^k|.$$

The production size of a function symbol is the maximum production size of any equation defining a function with the same precedence:

$$K_{\mathtt{h}} = \max_{\mathtt{g} \approx_{\mathscr{F}} \mathtt{h}} \max_{e = \mathtt{g}(\dots) \to t} K_e.$$

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:

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

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

https://doi.org/10.1017/S0960129511000685 Published online by Cambridge University Press

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. The idea is to extend the small pattern matchings so that their length reaches the length of the biggest production. This is illustrated by the following example:

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

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

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

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 in 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 the time complexity. Hence, bounding the time complexity of the normalised program is sufficient to bound the time complexity of the initial program. We will only consider normal programs in the following.

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 right-hand side of $\mathtt{f}$ and label them $\mathtt{g}_1, \ldots, \mathtt{g}_n$. This is simply an enumeration, and 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 the fact 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 $|v_i| \geqslant |u_i|$ for $1 \leqslant i \leqslant n$ and there exists j such that $|v_j| > |u_j|$.

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

https://doi.org/10.1017/S0960129511000685 Published online by Cambridge University Press

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 $n \times (\max_{1 \leqslant i \leqslant n} |v_i|)$ nodes with the same precedence as $\mathtt{f}$.

Lemma 28. Suppose 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 ith components are the same for $1 \leqslant i \leqslant n$. We use $v, v_1, u_1, q_1, v_2, u_2, q_2$ to denote 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

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

and the fact that the program is normal implies that $|\Upsilon_e| \leqslant |\Omega_e|$ (previous lemma). Using a similar denotation for the equation corresponding to the label α, we have

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

In this case, we write $\Omega_\alpha(x) \hookrightarrow \Upsilon_\alpha(x)$ as a short-hand notation, and define $\Omega_\beta, \Upsilon_\beta, \Omega_\gamma, \Upsilon_\gamma$ similarly. 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, $|\Omega_\beta| \geqslant |\Upsilon_\alpha|$. Hence, y_1 is a suffix of x_1, which is 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:

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

Hence $z_1 = z_2$, so $q_1 = q_2$. □

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.

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, and not an induction on it. If $\omega_1 = \alpha_1 \ldots \alpha_n$, 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. □

https://doi.org/10.1017/S0960129511000685 Published online by Cambridge University Press

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 $\mathscr{T}$ 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 $\mathscr{T}$ with the same precedence as $\mathtt{f}$ is bounded by $(I+1)^M$, which is a polynomial in $|\eta|$.

Proof. Any descendant of η with the same precedence can be identified with a word over $\{\mathtt{f}^1, \ldots, \mathtt{f}^M\}$. 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 with 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_i^n be the number of elements from $\mathbb{N}^n$ whose sum is i. This is the number of words of length i over an n-ary alphabet modulo commutativity. It is clear that $D_{i-1}^n \leqslant D_i^n$ (take all the n-tuple whose sum is $i-1$, then add 1 to the first component and you obtain D_{i-1}^n different n-tuples whose sum of components is i). Now, to count D_i^n, we choose a value j for the first component, then find $n-1$ components whose sum is $i-j$. There are D_{i-j}^{n-1} such elements.

$$D_i^n = \sum_{0 \leqslant j \leqslant i} D_{i-j}^{n-1} = \sum_{0 \leqslant j \leqslant i} D_j^{n-1} \leqslant (i+1) \times D_i^{n-1} \leqslant (i+1)^{n-1} \times D_i^1 \leqslant (i+1)^n.$$

This concludes the proof. □

Definition 21 (Bounded values). A program $\mathtt{f} = \langle \mathscr{X}, \mathscr{C}, \mathscr{F}, \mathscr{E} \rangle$ has polynomially bounded values if and only if for every n-ary function symbol $\mathtt{g} \in \mathscr{F}$, 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 $|\eta'| \leqslant p_{\mathtt{g}}(|\eta|)$.

Proposition 31. Any deterministic program $\mathtt{f}$ over unary constructors that

(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 that a program over unary constructors terminating by EPPO and admitting a QI is polytime.

Proof. 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). □

https://doi.org/10.1017/S0960129511000685 Published online by Cambridge University Press

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 in 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 does exactly delineate those EPPO programs over unary constructors that are polytime in the semantics with memoisation.

Proof. The implication (1) $\Rightarrow$ (2) is given by Proposition 31. For the converse, it is sufficient to observe 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. □

In fact, 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 further investigation of this question for future work.

Remark 2. The notion of a *sup-interpretation* (Marion and Péchoux 2009; Marion and Péchoux 2006; Péchoux 2007) has been introduced to generalise quasi-interpretations. However, not all programs admitting a sup-interpretation satisfy the bounded-values condition. Indeed sup-interpretations only relate the interpretation of a term to the interpretation of its normal form (see Marion and Péchoux (2009, Section 4.2, Definition 4.7)), while the notion of a program with polynomially bounded values requires that, in addition, all intermediates states of the call-tree are also bounded. As a counter-example, consider the program defined by

$$\begin{aligned} \mathtt{f}(0, y) &\to 0 \\ \mathtt{f}(\mathbf{s}x, y) &\to \mathtt{f}(x, \mathtt{db}(y)) \\ \mathtt{db}(0) &\to 0 \\ \mathtt{db}(\mathbf{s}(x)) &\to \mathbf{s}(\mathbf{s}(\mathtt{db}(x))). \end{aligned}$$

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

$$\begin{aligned} \theta(\mathtt{f})(X, Y) &= X \\ \theta(\mathtt{db})(X) &= 2X, \end{aligned}$$

and, as expected,

$$\begin{aligned} \theta(\mathbf{s})(X) &= X + 1 \\ \theta(0) &= 0. \end{aligned}$$

However, the final call of f during the execution of $\mathtt{f}(\mathbf{s}^n(0), 0)$ is $\mathtt{f}(0, \mathbf{s}^m(0))$ 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

https://doi.org/10.1017/S0960129511000685 Published online by Cambridge University Press

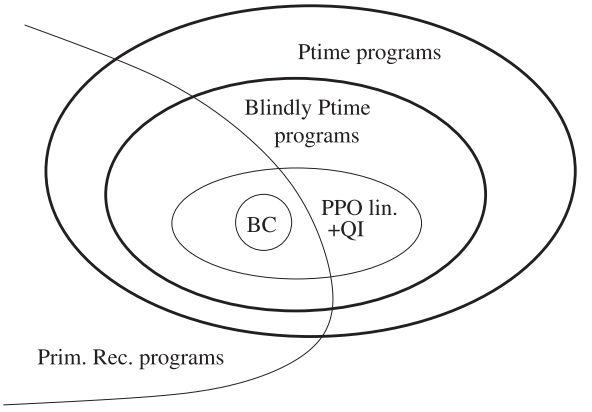


Fig. 10. Subclasses of programs.

recursive calls). As stated in Marion and Péchoux (2009, Theorem 5.3), 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 function symbols. More precisely, these sup-interpretations can be chosen to be polynomials. It is sufficient for that to take for any function symbol g of the program a polynomial p_g as given by Definition 21.

8. Conclusions

We have introduced blind abstractions of first-order functional programs and exploited them to help us gain an understanding of 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*) that describes those deterministic programs admitting product path-orderings that are 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 caught by one and not by the other. The comparison between the Bellantoni–Cook BC class and our 'blind P-criterion' is illustrated in Figure 10.

Further work includes investigations into the conditions characterising the class of programs captured by quasi-interpretation. In particular, the question is still open as to 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 and Péchoux 2009), provided some sort of path-ordering for it exists.

https://doi.org/10.1017/S0960129511000685 Published online by Cambridge University Press

Another direction is to try to find notions similar to blindly polytime that would help comparing and classifying ICC systems. We have seen that being blindly polytime is a condition independent of 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 thus complete the picture in Figure 10.

References

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

Avanzini, M. and Moser, G. (2008) Complexity analysis by rewriting. In: Proceedings of 9th International Symposium on Functional and Logic Programming. *Springer-Verlag Lecture Notes in Computer Science* **4989** 130–146.

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

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

Bonfante, G., Marion, J.-Y. and Moyen, J.-Y. (2001) On lexicographic termination ordering with space bound certifications. In: Ershov Memorial Conference. *Springer-Verlag Lecture Notes in Computer Science* **2244** 482–493.

Bonfante, G., Marion, J.-Y. and Moyen, J.-Y. (2005) Quasi-Interpretations and Small Space Bounds. In: Proceedings of The International Conference on Rewriting Techniques and Applications (RTA 2005). *Springer-Verlag Lecture Notes in Computer Science* **3467**.

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

Bonfante, G., Marion, J.-Y. and Péchoux, R. (2007) Quasi-interpretation synthesis by decomposition. In: Proceedings of the International Colloquium on Theoretical Aspects of Computing (ICTAC 2007). *Springer-Verlag Lecture Notes in Computer Science* **4711** 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)* 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. and Lévy, J.-J. (1980) Attempts for generalising the recursive path orderings. Technical report, University of Illinois, Urbana Champaign. Available at `http://perso.ens-lyon.fr/pierre.lescanne/not_accessible.html#termination`.

Klop, J. W. and de Vrijer, R. (2003) *Term Rewriting Systems*, Cambridge University Press.

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

https://doi.org/10.1017/S0960129511000685 Published online by Cambridge University Press

Leivant, D. and Marion, J.-Y. (1993) Lambda calculus characterizations of poly-time. In: Proceedings of Typed Lambda Calculi and Applications (TLCA '05). *Springer-Verlag Lecture Notes in Computer Science* **664** 274–288.

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

Marion, J.-Y. and 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). *Springer-Verlag Lecture Notes in Artificial Intelligence* **1955** 25–42.

Marion, J.-Y. and Péchoux, R. (2006) Resource analysis by sup-interpretation. In: Proceedings of Symposium on Functional and Logic Programming (FLOPS '06) *Springer-Verlag Lecture Notes in Computer Science* **3945** 163–176.

Marion, J.-Y. and 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 (Ph.D. Thesis), University Nancy 2.

Péchoux, R. (2007) *Analyse de la complexité des programmes par interprétation sémantique*, Ph.D. Thesis, Institut National Polytechnique de Lorraine.

https://doi.org/10.1017/S0960129511000685 Published online by Cambridge University Press