HAL open science

# Observation of implicit complexity by non confluence

Guillaume Bonfante

## To cite this version:

Guillaume Bonfante. Observation of implicit complexity by non confluence. International Workshop on Developments in Implicit Computational complExity - DICE 2010, Mar 2010, Paphos, Cyprus. inria-00535539

**HAL Id: inria-00535539**

**https://inria.hal.science/inria-00535539v1**

Submitted on 11 Nov 2010

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Observation of implicit complexity by non confluence

Guillaume Bonfante
Nancy University
guillaume(dot)bonfante(at)loria(dot)fr

We propose to consider non confluence with respect to implicit complexity. We come back to some well known classes of first-order functional program, for which we have a characterization of their intentional properties, namely the class of cons-free programs, the class of programs with an interpretation, and the class of programs with a quasi-interpretation together with a termination proof by the product path ordering. They all correspond to PTIME. We prove that adding non confluence to the rules leads to respectively PTIME, NPTIME and PSPACE. Our thesis is that the separation of the classes is actually a witness of the intentional properties of the initial classes of programs.

In implicit complexity theory, one of the issues is to characterize large classes of programs, not extensionally but intentionally. That is, for a given class of functions, to delineate the largest set of programs computing this class. One of the issues with this problem is that it is hard to compare (classes of) programs. Indeed, strict syntactical equality is definitely too restrictive, but larger (the interesting ones) relations are undecidable. So, comparing theories (defining their own class of programs) is rather complicated. Usually, one gives a remarkable example, illustrating the power of the theory.

We propose here an other way to compare sets of programs. The idea is to add a new feature–in the present settings, non determinism– to two programming languages. Intuitively, if a function can be computed with this new feature in a language $L_1$ but non in the language $L_2$, we say that $L_1$ is more powerful than $L_2$. Let us formalize a little bit our intuition.

Suppose for the discussion that programs are written as rewriting systems, that is, programming languages are classes of rewriting systems. Let us say furthermore that a program $p$ is simulated by $q$ whenever each step of rewriting in $t \overset{\ell\to r}{\to} u \in p$ can be simulated by a rewriting step $t' \overset{\ell'\to r'}{\to} u' \in q$. Equivalence of languages $L_1$ and $L_2$ states that any program in $L_1$ is simulated by a program in $L_2$ and vice versa.

Given a programming language $L$, its non deterministic extension $L.n$ is the programming language obtained by adding to $L$ an oracle **choose**$(r_1, r_2)$ which, given two rules $r_1$ and $r_2$ which can be defined in $L$, applies the "right" rule depending on the context. So, given some $p \in L$, both $p \cup \{r_1\}$ and $p \cup \{r_2\}$ are supposed to be in $L$, but not necessarily $p \cup \{r_1, r_2\}$.

Suppose now, that we are given two programming languages $L_1$ and $L_2$ such that functions computed in $L_1.n$ are strictly included in those in $L_2.n$. Then, $L_1$ cannot simulate $L_2$. Ad absurdum, suppose that $L_1$ can simulate $L_2$, take $f$ computed by $p_2 \in L_2.n$ but not in $L_1.n$. Then, each rule $\ell \to r$ in $p_2$ is simulated by a rule $\ell' \to r'$ in some program $p_1 \in L_1$. But then, the derivation $t_1 \to t_2 \to \cdots \to t_n$ in $L_2.n$ can be simulated by some derivation $t'_1 \to t'_2 \to \cdots \to t'_n$ in $L_1.n$. This leads to the contradiction.

It is clear that the notion of equivalence we took for the discussion is very strong. However, we believe that the argument would hold in a larger context.

In this paper, we observe three programming languages,

- programs with a polynomially bounded constructor preserving interpretation (F.cons) which extend cons-free programs,

[0]Work partially supported by project ANR-08-BLANC-0211-01 (COMPLICE)

- programs with (polynomial) strict interpretation (F.SI) and
- programs with a quasi-interpretation together with a proof of termination by PPO, written F.QI.PPO.

These three languages characterize PTIME. The first one is a new result, the two latter ones are respectively proved in [2] and [3].

$$\begin{array}{ccccc} \text{F.cons} & & \text{F.SI} & & \text{F.QI.}\mathit{PPO} \\ \downarrow & & \downarrow & & \downarrow \\ \text{PTIME} & = & \text{PTIME} & = & \text{PTIME} \end{array}$$

Their non deterministic observation characterize PTIME, NPTIME and PSPACE. The second characterization has been proven in [2].

$$\begin{array}{ccccc} \text{F.cons.}n & & \text{F.SI.}n & & \text{F.QI.}\mathit{PPO}.n \\ \downarrow & & \downarrow & & \downarrow \\ \text{PTIME} & \stackrel{?}{\neq} & \text{NPTIME} & \stackrel{?}{\neq} & \text{PSPACE} \end{array}$$

The issue of confluence of Term Rewriting Systems has been largely studied, see for instance [19]. It benefits from some nice properties, for instance it is modular and algorithms are given to automatically compute the confluence up to termination.

It is clear that there is also an intrinsic motivation for a study of non confluent programs. It would not be reasonable to cover all the researches dealing with this issue. But, let us make three remarks. First, since non-confluence can give us some freedom to write programs, it is of interest to observe what new functions this extra feature allows us to compute. From our result about non-confluent programs in F.cons.$n$, one may extract a compilation procedure to compute them "deterministically". Second, Kristiansen and Mender in [15, 16] have proposed a scale –a la Grzegorczyk, using non determinism, to characterize LINSPACE. Finally, one should keep in mind characterizations in the logical framework. Let us mention for instance the characterization of PSPACE given in [7]. It is an extension of a characterization of PTIME, and thus, we think that their construction is a good candidate for observation as presented above.

# 1 Preliminaries

We suppose that the reader has familiarity with first-order rewriting. We briefly recall the context of the theory, essentially to fix the notations. Dershowitz and Jouannaud's survey [6] of rewriting is a good entry point for beginners.

Let $\mathscr{X}$ denote a (countable) set of *variables*. Given a *signature* $\Sigma$, the set of *terms* over $\Sigma$ and $\mathscr{X}$ is denoted by $\mathbf{T}(\Sigma, \mathscr{X})$ and the set of *ground terms*, that is terms without variables, by $\mathbf{T}(\Sigma)$.

The size $|t|$ of a term $t$ is defined as the number of symbols in $t$. For example the size of the term $f(a,x)$ is 3.

A *context* is a term $C$ with a particular variable $\Diamond$. If $t$ is a term, $C[t]$ denotes the term $C$ where the variable $\Diamond$ has been replaced by $t$.

## 1.1 Syntax of programs

Let $\mathscr{C}$ be a (finite) signature of *constructor* symbols and $\mathscr{F}$ a (finite) signature of *function symbols*. Thus, we are given an algebra of constructor terms $\mathbf{T}(\mathscr{C}, \mathscr{X})$. A rule is a pair $(\ell, r)$, next written $\ell \to r$, where:

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

**Definition 1.** A *program* is a quadruplet $\mathtt{f} = \langle \mathscr{X}, \mathscr{C}, \mathscr{F}, \mathscr{E} \rangle$ such that $\mathscr{E}$ is a finite set of rules. We distinguish among $\mathscr{F}$ a main function symbol whose name is given by the program name `f`. `F` denotes the set of programs.

The set of rules induces a rewriting relation $\rightarrow$. The relation $\xrightarrow{+}$ is the transitive closure of $\rightarrow$, and $\xrightarrow{*}$ is the reflexive and transitive closure of $\rightarrow$. Finally, we say that a term $t$ is a *normal form* if there is no term $u$ such that $t \rightarrow u$. Given two terms $t$ and $u$, $t \xrightarrow{!} u$ denotes the fact that $t \xrightarrow{*} u$ and $u$ is a normal form.

All along, when it is not explicitly mentioned, we suppose programs to be *confluent*, that is, the rewriting relation is confluent.

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

**Definition 2** (Call-tree)**.** Suppose we are given a program $\langle \mathscr{X}, \mathscr{C}, \mathscr{F}, \mathscr{E} \rangle$. Let $\rightsquigarrow$ be the relation

$$(f, t_1, \ldots, t_n) \rightsquigarrow (g, u_1, \ldots, u_m) \Leftrightarrow f(t_1, \ldots, t_n) \rightarrow C[g(v_1, \ldots, v_m)] \xrightarrow{*} C[g(u_1, \ldots, u_m)]$$

where $f$ and $g$ are function symbols, $C$ is a context and $t_1, \ldots, t_n, u_1, \ldots, u_m$ are constructor terms. Given a function symbol $f$ and constructor terms $t_1, \ldots, t_n$, the relation $\rightsquigarrow$ defines a tree whose root is $(f, t_1, \ldots, t_n)$ and $\eta'$ is a daughter of $\eta$ iff $\eta \rightsquigarrow \eta'$. The relation $\rightsquigarrow^+$ is the transitive closure of $\rightsquigarrow$.

## 1.2 Interpretations of programs

Given a signature $\Sigma$, a $\Sigma$-algebra on a domain $A$ is a mapping $(\!|-|\!)$ which associates to every $n$-ary symbol $f \in \Sigma$ an $n$-ary function $(\!|f|\!) : A^n \rightarrow A$. Such a $\Sigma$-algebra can be extended to terms by:

- $(\!|x|\!) = 1_A$, that is the identity on $A$, for $x \in \mathscr{X}$,
- $(\!|f(t_1, \ldots, t_m)|\!) = \mathrm{comp}((\!|f|\!), (\!|t_1|\!), \ldots, (\!|t_m|\!))$ where comp is the composition of functions.

Given a term $t$ with $n$ variables, $(\!|t|\!)$ is a function $A^n \rightarrow A$.

**Definition 3.** Given an ordered set $(A, <)$ and a program $\langle \mathscr{X}, \mathscr{C}, \mathscr{F}, \mathscr{E} \rangle$, let us consider a $(\mathscr{C} \cup \mathscr{F})$-algebra $(\!|-|\!)$ on $A$. It is said to:

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

**Definition 4.** Given an ordered set $(A,<)$ and a program $\langle \mathscr{X},\mathscr{C},\mathscr{F},\mathscr{E}\rangle$, a $(\mathscr{C}\cup\mathscr{F})$-algebra on $A$ is said to be a strict interpretation whenever it verifies (1), (3), (4). It is a quasi-interpretation whenever it verifies (2), (3), (5). It is a monotone interpretation whenever it verifies (2) and (5).

Clearly, a strict interpretation is a quasi-interpretation which itself is a monotone interpretation.When we want to speak arbitrarily of one of those concepts, we use the generic word "interpretation". We also use this terminology to speak about the function $(\!|f|\!)$ given a symbol $f$.

Finally, by default, $A$ is chosen to be the set of real non negative numbers with its usual ordering. Moreover, we restrict the interpretations over the real numbers to be *Max-Poly functions*, that is functions obtained by finite compositions of the constant functions, maximum, addition and multiplication. **Max-Poly** denotes the set of these functions.

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

$$\begin{array}{rcl}
\varepsilon=\varepsilon & \to & \mathbf{tt}\\
\mathbf{i}(x)=\mathbf{i}(y) & \to & x=y \text{ with } \mathbf{i}\in\{0,1\}\\
\mathbf{i}(x)=\mathbf{j}(y) & \to & \mathbf{ff} \text{ with } \mathbf{i}\neq\mathbf{j}\in\{0,1\}\\
\mathtt{or}(\mathbf{tt},y) & \to & \mathbf{tt}\\
\mathtt{or}(\mathbf{ff},y) & \to & y\\
\mathtt{and}(\mathbf{tt},y) & \to & y\\
\mathtt{and}(\mathbf{ff},y) & \to & \mathbf{ff}\\
\mathtt{if}\ \mathbf{tt}\ \mathtt{then}\ y\ \mathtt{else}\ z & \to & y\\
\mathtt{if}\ \mathbf{ff}\ \mathtt{then}\ y\ \mathtt{else}\ z & \to & z\\
\mathtt{in}(a,\mathbf{nil}) & \to & \mathbf{ff}\\
\mathtt{in}(a,\mathbf{cons}(b,l)) & \to & \mathtt{if}\ a=b\ \mathtt{then}\ \mathbf{tt}\ \mathtt{else}\ \mathtt{in}(a,l)
\end{array}$$

Such a program has the strict interpretation[1] given by:

$$\begin{array}{rcl}
(\!|\varepsilon|\!)=(\!|\mathbf{tt}|\!)=(\!|\mathbf{ff}|\!)=(\!|\mathbf{nil}|\!) & = & 1\\
(\!|\mathbf{i}|\!)(x) & = & x+1 \text{ with } \mathbf{i}\in\{0,1\}\\
(\!|\mathbf{cons}|\!)(x,y) & = & x+y+1\\
(\!|=|\!)(x,y)=(\!|\mathtt{or}|\!)(x,y)=(\!|\mathtt{and}|\!)(x,y) & = & x+y\\
(\!|\mathtt{if\ then\ else}|\!)(x,y,z) & = & x+y+z\\
(\!|\mathtt{in}|\!)(x,y) & = & (x+2)\times y
\end{array}$$

**Definition 5.** The interpretation of a symbol $f$ is said to be additive if it has the shape $\sum_i x_i + c$. A program with an interpretation is said to be additive when its *constructors* are additive.

## 1.3 Termination by Product Path Ordering

Let us recall that the Product Path Ordering is a particular form of the Recursive Path Orderings, a class of simplification orderings (and so well-founded). Pioneers of this subject include Plaisted [17], Dershowitz [5], Kamin and Lévy [12].

[1] To simplify the verification of inequalities, interpretations are taken in $[1,\infty[$.

$$\frac{s = t_i \text{ or } s \prec_{ppo} t_i}{s \prec_{ppo} \mathtt{f}(\ldots, t_i, \ldots)} \; \mathtt{f} \in \mathscr{F} \bigcup \mathscr{C}$$

$$\frac{\forall i \; s_i \prec_{ppo} \mathtt{f}(t_1, \cdots, t_n)}{\mathbf{c}(s_1, \cdots, s_m) \prec_{ppo} \mathtt{f}(t_1, \cdots, t_n)} \; \mathtt{f} \in \mathscr{F}, \mathbf{c} \in \mathscr{C}$$

$$\frac{\forall i \; s_i \prec_{ppo} \mathtt{f}(t_1, \cdots, t_n) \qquad \mathtt{g} \prec_{\mathscr{F}} \mathtt{f}}{g(s_1, \cdots, s_m) \prec_{ppo} \mathtt{f}(t_1, \cdots, t_n)} \; \mathtt{f}, \mathtt{g} \in \mathscr{F}$$

$$\frac{(s_1, \cdots, s_n) \prec^p_{ppo} (t_1, \cdots, t_n) \qquad \mathtt{f} \approx_{\mathscr{F}} \mathtt{g} \qquad \forall i \; s_i \prec_{ppo} \mathtt{f}(t_1, \cdots, t_n)}{\mathtt{g}(s_1, \cdots, s_n) \prec_{ppo} \mathtt{f}(t_1, \cdots, t_n)} \; \mathtt{f}, \mathtt{g} \in \mathscr{F}$$

Figure 1: Definition of $\prec_{ppo}$

Finally, let us mention that Krishnamoorthy and Narendran in [14] have proved that deciding whether a program terminates by Recursive Path Orderings is a NP-complete problem.

Let $\preceq_\Sigma$ be a preorder on a signature $\Sigma$, called *quasi-precedence* or simply *precedence*. We write $\prec_\Sigma$ for the induced strict precedence and $\simeq_\Sigma$ for the induced equivalence relation on $\Sigma$. Usually, the context makes clear what $\Sigma$ is, and thus, we drop the subscript $\Sigma$.

**Definition 6.** Given an ordering $\preceq$ over terms $\mathbf{T}(\Sigma)$, the product extension of $\preceq$ over sequences, written $\preceq^p$, is defined as $(s_1, \ldots, s_k) \prec^p (t_1, \ldots, t_k)$ iff

- for all $i \leq k : s_i \preceq t_i$ and,
- there is some $j \leq k$ such that $s_i \prec t_i$.

where $\prec$ is the strict part of $\preceq$.

**Definition 7.** Given a program $\langle \mathscr{X}, \mathscr{C}, \mathscr{F}, \mathscr{E} \rangle$ and a precedence $\preceq_{\mathscr{F}}$ over function symbols, the Product Path Ordering $\prec_{ppo}$ is defined as the least ordering verifying rules given in Figure 1.

## 1.4 Characterizations in the confluent case

**Theorem 1** (Bonfante, Cichon, Marion and Touzet [2])**.** *Functions computed by programs with additive strict interpretation are exactly* PTIME *functions.*

It is Theorem 4 in [2], first item.

**Theorem 2** (Bonfante, Marion and Moyen [3])**.** *Functions computed by programs with*

- *an additive quasi-interpretation and*
- *a termination proof by PPO*

*are exactly* PTIME *functions.*

The programs of this latter theorem are mentioned as $\mathrm{RPO}^{\mathrm{QI}}_{Pro}$-programs in [3], Theorem 48.

## 2 Constructor preserving interpretations

It is well known that the interpretations above can be used to bound both the length of the computations and the size of terms during the computations (see for instance [9, 18]). Here we show that interpretations also cope with syntactic constraints. More precisely, programs with polynomial constructor preserving interpretations generalize cons-free programs [11].

Let us consider a signature $\mathscr{C}$, the signature of constructors in the sequel. $S(\mathscr{C})$ denotes the set of *finite non-empty sets* of terms in $\mathbf{T}(\mathscr{C})$. Let $\trianglelefteq$ denotes the sub-term relation on terms. On $S(\mathscr{C})$, we define $\trianglelefteq^*$ as follows: $m \trianglelefteq^* m'$ iff $\forall t \in m : \exists t' \in m' : t \trianglelefteq t'$. As an ordering on sets, for interpretations, we will use the inclusion relation. One may observe that $m \subseteq m' \Longrightarrow m \trianglelefteq^* m'$.

**Definition 8.** Let us consider a monotone interpretation $(\!|-|\!)$ of a program $\langle \mathscr{X}, \mathscr{C}, \mathscr{F}, \mathscr{E} \rangle$ over $(S(\mathscr{C}), \subseteq)$. We say that it preserves constructors if

1. for any constructor symbol $\mathbf{c} \in \mathscr{C}$, $(\!|\mathbf{c}|\!)(m_1, \ldots, m_k) = \{\mathbf{c}(t_1, \ldots, t_k) \mid t_i \in m_i, i = 1, \ldots, k\}$.
2. given a rule $f(p_1, \cdots, p_n) \to r$ and a ground substitution $\sigma$, for all $u \trianglelefteq r$,

$$(\!|\sigma(u)|\!) \trianglelefteq^* (\!|\sigma(f(p_1, \cdots, p_n))|\!) \cup_{i=1}^{n} (\!|\sigma(p_i)|\!).$$

By extension, we say that a program is constructor preserving if it admits a constructor preserving monotone interpretation.

The fact that a program preserves constructors fixes the definition of the interpretation over constructors. Moreover, (1) below gives a simple characterization of the interpretations of constructor terms.

**Proposition 1.** *For any constructor preserving interpretation $(\!|-|\!)$, the following holds:*

1. *for any ground constructor term $t$, $(\!|t|\!) = \{t\}$,*
2. *given a ground substitution $\sigma$, for any constructor terms $u \trianglelefteq v$, $(\!|\sigma(u)|\!) \trianglelefteq^* (\!|\sigma(v)|\!)$.*

*Proof.* (1) is proved by induction on the structure of terms. (2) is by induction on the structure of $v$. Suppose $v$ is a variable, if $u \trianglelefteq v$, then $u = v$ and the property holds trivially. Suppose $v = \mathbf{c}(v_1, \ldots, v_k)$. The case $u = v$ is as above. Otherwise, $u \trianglelefteq v_j$ for some $j \leq k$. In that case, for all $t \in (\!|\sigma(u)|\!)$, by induction, there is a $w_j \in (\!|\sigma(v_j)|\!)$ such that $t \trianglelefteq w_j$. Let us choose some $w_i \in (\!|\sigma(v_i)|\!)$ for all $i \neq j$. Then, $t \trianglelefteq \mathbf{c}(w_1, \ldots, w_k) \in (\!|\sigma(\mathbf{c}(v_1, \ldots, v_k))|\!)$. □

**Proposition 2.** *Given a program $\langle \mathscr{X}, \mathscr{C}, \mathscr{F}, \mathscr{E} \rangle$ and a constructor preserving monotone interpretation $(\!|-|\!)$, for all constructor terms $t_1, \ldots, t_n$ and all symbols $f$ of arity $n$, if $[\![f]\!](t_1, \ldots, t_n) = t$, then $t \in (\!|f(t_1, \ldots, t_n)|\!)$.*

*Proof.* For a monotone interpretation, if $u \to v$, then $(\!|v|\!) \subseteq (\!|u|\!)$. Suppose that $f(t_1, \ldots, t_n) \xrightarrow{!} t$ with $t$ a constructor term, then, $(\!|t|\!) \subseteq (\!|f(t_1, \ldots, t_n)|\!)$. But, due to Proposition 1-(1), $(\!|t|\!) = \{t\}$. The conclusion follows. □

**Definition 9.** In the present context, an interpretation is said to be polynomially bounded if for any symbol $f$, for any sets $m_1, \ldots, m_n$, the set $(\!|f|\!)(m_1, \ldots, m_n)$ has a size polynomially bounded w.r.t. to the size of the $m_i$'s. The size of a set $m$ is defined to be $|m| = \sum_{t \in m} |t|$.

**Proposition 3.** *Given a constructor preserving program, then, for all constructors $\mathbf{c}$, the size of the set $(\!|\mathbf{c}|\!)(m_1, \ldots, m_n)$ is polynomially bounded w.r.t. the size of the $m_i$'s.*

*Proof.* Let us write $M = \sum_{i=1}^{n} |m_i|$. Then,

$$\begin{array}{rcll} |(\!|\mathbf{c}|\!)(m_1,\ldots,m_n)| & \leq & \displaystyle\sum_{i\leq n, t_i\in m_i} |\mathbf{c}(t_1,\ldots,t_n)| & \\ & \leq & \displaystyle\sum_{i\leq n, t_i\in m_i} n\times M+1 & \text{since } |t_i|\leq |m_i|\leq M \\ & \leq & M^n\times(n\times M+1) & \text{by a rough enumeration of} \\ & & & \text{the indices of the sum.} \end{array}$$

□

**Example 2.** Let us come back to Example 1, it has a polynomially bounded constructor preserving monotone interpretation. Apart from the generic interpretation on constructors, we define:

$$\begin{array}{rcl} (\!|=|\!)(m,m') = (\!|\texttt{in}|\!)(m,m') & = & \{\mathbf{tt},\mathbf{ff}\} \\ (\!|\texttt{and}|\!)(m,m') = (\!|\texttt{or}|\!)(m,m') & = & m'\cup\{\mathbf{tt},\mathbf{ff}\} \\ (\!|\texttt{if then else}|\!)(m_b,m_y,m_z) & = & m_y\cup m_z \end{array}$$

It is clear that this interpretation is polynomially bounded.

Actually, the notion of constructor preserving programs generalizes the notion of constructor-free programs as introduced by Jones (see for instance [11]). He has shown how constructor-free programs characterize PTIME and LOGSPACE. We recall that a program is constructor-free whenever, for any rule $f(p_1,\cdots,p_n)\to r$, for any subterm $t\trianglelefteq r$,

- if $t$ is a constructor term, then $t\trianglelefteq f(p_1,\cdots,p_n)$,
- otherwise, the root of $t$ is not a constructor symbol.

**Proposition 4.** *Any* constructor-free *program* $\langle\mathscr{X},\mathscr{C},\mathscr{F},\mathscr{E}\rangle$ *has a polynomially bounded constructor preserving monotone interpretation* $(\!|-|\!)$.

*Proof.* We use the generic definition for constructor symbols. For functions, let

$$(\!|f|\!)(m_1,\ldots,m_n) = \{u \mid \exists i\leq n, t\in m_i : u\trianglelefteq t\}.$$

We have to prove a) that it is a monotone interpretation over $S(\mathscr{C})$, b) that it preserves constructors and c) that it is polynomially bounded.

**Proof of c).** Due to Proposition 1, it is sufficient to verify the size condition on function symbols. Given some sets $m_1,\ldots,m_n$, we define $m = \{u \mid \exists i\leq n, t\in m_i : u\trianglelefteq t\}$ and $K = |\cup_{i=1}^{n} m_i|$. From the definition of the size of a set, for all $j\leq n$, $|m_j|\leq K\leq\sum_{i=1}^{n}|m_i|$. Let $S_{m_1,\ldots,m_n} = \{t \mid \exists i\leq n : t\in m_i\}$. Then,

$$\#S_{m_1,\ldots,m_n} \leq \sum_{i=1}^{n} \#m_i \leq n\times K \tag{1}$$

where $\#m$ denotes the cardinality of a set $m$ (recall that for any set $m$ : $\#m\leq|m|!$). For each term $t$, let $D_t = \{u \mid u\trianglelefteq t\}$. Since $m = \cup_{t\in S_{m_1,\ldots,m_n}} D_t$, $|m|\leq\sum_{t\in S_{m_1,\ldots,m_n}}|D_t|$. It is clear that for all $t$, $\#D_t = |t|$. Moreover, each $u\trianglelefteq t$ has a size smaller than $t$. Consequently, $|D_t|\leq|t|^2$. Since for all terms $t\in S_{m_1,\ldots,m_n}$, $|t|\leq K$, we have $|m|\leq\sum_{t\in S_{m_1,\ldots,m_n}} K^2$. Combining this latter equation with Equation 1, we can state that $|m|\leq n\times K^3\leq n\times(\sum_{i=1}^{n}|m_i|)^3$.

**Proof of b).** Let us come back to the Definition 8. The first item comes from our generic choice for the interpretations of constructor symbols. Concerning the second item, let us consider a rule $f(p_1,\cdots,p_n)\to r$, a ground substitution $\sigma$ and a subterm $u\trianglelefteq r$. We prove actually a stronger fact than condition 2, namely: $(\!|\sigma(u)|\!)\subseteq(\!|\sigma(f(p_1,\cdots,p_n))|\!)$. By induction on $u$.

If $u$ is a variable or more generally a constructor term, since the program is constructor-free, $u\trianglelefteq p_j$ for some $j$. Take $t\in(\!|\sigma(u)|\!)$. Due to Proposition 1-(2), there is $t'\in(\!|\sigma(p_j)|\!)$ such that $t\trianglelefteq t'$. Then, recalling the definition of $(\!|f|\!)$, $t$ belongs to $(\!|f|\!)((\!|\sigma(p_1)|\!),\ldots,(\!|\sigma(p_n)|\!))=(\!|\sigma(f(p_1,\cdots,p_n))|\!)$.

Otherwise, $u=g(v_1,\ldots,v_k)$ and, since the program is constructor-free, $g$ is a function symbol. Take $t\in(\!|\sigma(u)|\!)=(\!|g|\!)((\!|\sigma(v_1)|\!),\ldots,(\!|\sigma(v_k)|\!))$. Recall that $g$ is a function symbol. Then, by definition of $(\!|g|\!)$, there is a $j\leq k$ and a term $t'\in(\!|\sigma(v_j)|\!)$ such that $t\trianglelefteq t'$. By induction, $t'\in(\!|\sigma(f(p_1,\cdots,p_n))|\!)$. But then, by definition of $(\!|f|\!)$, $t\in(\!|f(p_1,\cdots,p_n)|\!)$.

**Proof of a),** Item (2) of Definition 3 is a direct consequence of the definition of the interpretation. Let us justify now (5). As seen above, for all rules $f(p_1,\cdots,p_n)\to r$, ground substitutions $\sigma$ and subterms $u\trianglelefteq r$, $(\!|\sigma(u)|\!)\subseteq(\!|\sigma(f(p_1,\cdots,p_n))|\!)$. In particular, the result holds for $r$. □

Do those kind of interpretations really go beyond constructor-freeness? Here is an example of a program which is not constructor-free, but with a constructor preserving interpretation.

**Example 3.** Using the tally numbers $\mathbf{0},\mathbf{s}$, and lists, the function `f` builds the list of the first $n-1$ integers given the argument $n$.

$$\begin{aligned}\mathtt{f}(\mathbf{0}) &= \mathbf{nil}\\ \mathtt{f}(\mathbf{s}(n)) &= \mathbf{cons}(n,\mathtt{f}(n))\end{aligned}$$

Such a program has a constructor preserving interpretation. Let

$$\begin{aligned}(\!|f|\!)(m) &= \{\mathbf{cons}(n_1,\mathbf{cons}(n_2,\cdots(\mathbf{cons}(n_k,\mathbf{nil}))\cdots))\mid\\ &\qquad \exists n_1,\ldots,n_k\in m:\forall j\leq k-1:n_j=\mathbf{s}(n_{j+1})\}\\ &\quad\cup\{\mathbf{nil}\}.\end{aligned}$$

It is clear that this program is not constructor-free. But, there is a stronger difference: the function computed by this program cannot be computed by *any* constructor-free program. Indeed, recall that the output of functions computed by constructor-free programs are subterms of the inputs. Since this is not the case of `f`, the conclusion follows.

Let us make one last observation about the example. Actually, the interpretation is polynomially bounded. Indeed, a list of the shape

$$\mathbf{cons}(n_1,\mathbf{cons}(n_2,\cdots(\mathbf{cons}(n_k,\mathbf{nil}))\cdots))$$

is fixed by the choice of $n_1$ and $k$. Since $k\leq|n_1|\leq|m|$, there are at most $|m|^2$ such lists, each of which has a polynomial size.

**Theorem 3.** *Predicates computed by programs with a polynomially bounded constructor preserving interpretation are exactly* PTIME *predicates.*

*Proof.* From Jones's result and Proposition 4, it is clear that PTIME predicates can be computed by constructor preserving programs.

In the other direction, suppose we want to evaluate $f(t_1,\dots,t_n)$ where $t_1,\dots,t_n$ are some constructor terms. First, due to Propostion 2, one observes that the set of constructor terms $(\!|f(t_1,\dots,t_n)|\!) = (\!|f|\!)((\!|t_1|\!),\dots,(\!|t_n|\!))$ contains the normal form of $f(t_1,\dots,t_n)$. Moreover, this set has a polynomial size w.r.t. the size of $(\!|t_i|\!)$'s. Due to Proposition 1, $(\!|t_i|\!) = \{t_i\}$ and consequently $|(\!|t_i|\!)| = |t_i|$. That is $(\!|f(t_1,\dots,t_n)|\!)$ has a polynomial size w.r.t. the size of inputs.

As this is done by Jones, we use a call-by-value semantics with cache, that is:

- we restrict substitutions to ground constructor substitutions,
- each time a term $g(u_1,\dots,u_m)$ is evaluated, it is put in a map $(g,u_1,\dots,u_m) \mapsto [\![g]\!](u_1,\dots,u_m)$. This map is called the cache.

The key point to prove that computations can be done in polynomial time is to show that the cache has a polynomial size w.r.t. the size of the inputs. We begin to establish that for all constructor term $u_i$ such that $f(t_1,\dots,t_n) \xrightarrow{+} C[g(u_1,\dots,u_m)]$:

- there is a term $t \in (\!|f(t_1,\dots,t_n)|\!) \cup \{t_1,\dots,t_n\}$ such that $u_i \trianglelefteq t$,
- $(\!|g(u_1,\dots,u_m)|\!) \trianglelefteq^* (\!|f(t_1,\dots,t_n)|\!) \cup \{t_1,\dots,t_n\}$.

One will have noticed that $\{t_1,\dots,t_n\} = \cup_{i=1}^n (\!|t_i|\!)$, so that $(\!|f(t_1,\dots,t_n)|\!) \cup \{t_1,\dots,t_n\} = (\!|f(t_1,\dots,t_n)|\!) \cup_{i=1}^n (\!|t_i|\!)$. Second remark, terms like $f(t_1,\dots,t_n)$, $g(u_1,\dots,u_m)$ as above correspond to nodes in the call tree with $(f,t_1,\dots,t_n) \rightsquigarrow^+ (g,u_1,\dots,u_m)$. So, we work by induction on $\rightsquigarrow^+$.

**Base case.** Suppose that $(f,t_1,\dots,t_n) \rightsquigarrow (g,u_1,\dots,u_m)$. In other words, there is a context $C$ such that:

$$f(t_1,\dots,t_n) \to C[g(v_1,\dots,v_m)] \xrightarrow{*} C[g(u_1,\dots,u_m)].$$

By Lemma 1 below, $u_i \trianglelefteq t$ for some term $t \in (\!|f(t_1,\dots,t_n)|\!) \cup \{t_1,\dots,t_n\}$ as required.

For the second item, notice that $g(v_1,\dots,v_m) \trianglelefteq C[g(v_1,\dots,v_m)]$. Then,

$$\begin{array}{lll} (\!|g(u_1,\dots u_m)|\!) & \subseteq (\!|g(v_1,\dots,v_m)|\!) & \text{since } g(v_1,\dots,v_m) \xrightarrow{*} g(u_1,\dots,u_m) \\ & \trianglelefteq^* (\!|f(t_1,\dots,t_n)|\!) \cup \{t_1,\dots,t_n\} & \text{by Definition 8, second item} \end{array}$$

**Induction step.** Otherwise, $(f,t_1,\dots,t_n) \rightsquigarrow^+ (g,u_1,\dots,u_m) \rightsquigarrow (h,w_1,\dots,w_k)$. By induction, we have $(\!|g(u_1,\dots,u_m)|\!) \trianglelefteq^* (\!|f(t_1,\dots,t_n)|\!) \cup \{t_1,\dots,t_n\}$ and for all $i \leq m$, $\{u_i\} = (\!|u_i|\!) \trianglelefteq^* (\!|f(t_1,\dots,t_n)|\!) \cup \{t_1,\dots,t_n\}$. Consequently,

$$(\!|g(u_1,\dots,u_m)|\!) \cup_{i=1}^m (\!|u_i|\!) \trianglelefteq^* (\!|f(t_1,\dots,t_n)|\!) \cup \{t_1,\dots,t_n\}. \tag{2}$$

By Lemma 1, for all $w_i$, there is a term $v \in (\!|g(u_1,\dots,u_m)|\!) \cup_{i=1}^m (\!|u_i|\!)$. By Equation 2, there is a term $t \in (\!|f(t_1,\dots,t_n)|\!) \cup \{t_1,\dots,t_n\}$ such that $w_i \trianglelefteq t$.

For the second item,

$$h(w_1,\dots,w_k) \trianglelefteq^* (\!|g(u_1,\dots,u_m)|\!) \cup_{i=1}^m (\!|u_i|\!) \trianglelefteq^* (\!|f(t_1,\dots,t_n)|\!) \cup \{t_1,\dots,t_n\}.$$

where the first relation is due to Definition 8-(2).

After this preliminary work, we are ready to bound the size of the cache. As a consequence of what precedes, the arguments of all the calls $g(u_1,\dots,u_m)$ in the call tree are contained in the set

$$S = \{u \mid \exists t \in (\!|f(t_1,\dots,t_n)|\!) \cup \{t_1,\dots,t_n\} : u \trianglelefteq t\}.$$

Since $(\!| f(t_1,\dots,t_n) |\!) \cup \{t_1,\dots,t_n\}$ has a polynomial size, $S$ itself has cardinality bounded by a polynomial, say $P(|t_1|,\dots,|t_n|)$. As a consequence, since the $u_i$'s are in $S$, the cache has at most $|\mathscr{F}| \times P(|t_1|,\dots,|t_n|)^D$ entries, where $D$ is a bound on the arity of symbols. Since each elements $u_i$ and each normal form of $g(u_1,\dots,u_m)$ are subterms of $(\!| f(t_1,\dots,t_n) |\!) \cup \{t_1,\dots,t_n\}$, they have a polynomial size. Then, the cache itself has a polynomial size. □

**Lemma 1.** *Let $\langle \mathscr{X}, \mathscr{C}, \mathscr{F}, \mathscr{E} \rangle$ be a program with a polynomially bounded constructor preserving interpretation $(\!| - |\!)$. Suppose given a rewriting step $f(t_1,\dots,t_n) \to w$ with $t_1,\dots,t_n$ some constructor terms and $v \trianglelefteq w$. If $v \xrightarrow{!} u$ with $u$ a constructor term, then there is a term $t \in (\!| f(t_1,\dots,t_n) |\!) \cup_{i=1}^n (\!| t_i |\!)$ such that $u \trianglelefteq t$.*

*Proof.* Let $f(p_1,\cdots,p_n) \to r$ and $\sigma$ be such that $f(t_1,\dots,t_n) = \sigma(f(p_1,\cdots,p_n)) \to \sigma(r) = w$. There are two cases: if $v \trianglelefteq \sigma(x)$ for some variable $x \in p_j$. Since the $t_i$ are constructor terms, $v$ is necessarily a constructor term, and consequently a normal form. So, $v = u$. As a matter of fact, $v \trianglelefteq \sigma(p_j) = t_j$. We conclude taking $t = t_i$.

Otherwise, $v = \sigma(v')$ for some $v' \trianglelefteq r$. Since $v \xrightarrow{!} u$, we have $(\!| u |\!) \subseteq (\!| v |\!)$. By Proposition 1-(1), $u \in (\!| v |\!)$. Due to Definition 8, second item, there is $t \in (\!| f(t_1,\dots,t_n) |\!) \cup_{i=1}^n (\!| t_i |\!)$ such that $u \trianglelefteq t$. □

# 3 Observation by non confluence

## 3.1 Semantics

We first have to define what we mean when we say that a function is computed by a non-confluent rewriting system. Computations lead to several normal forms, depending on the reductions applied. At first sight, we shall regard a non-confluent rewrite system as a non-deterministic algorithm.

In this section, given a program, we do not suppose its underlying rewriting system to be confluent. By extension, we say that such programs are not confluent (even if they may be so).

**Example 4.** A 3-SAT formula is given by a set of clauses, written $\vee(x_1,x_2,x_3)$ where the $x_i$ have either the shape $\neg(n_i)$ or $\mathtt{e}(n_i)$.[2] The $n_i$'s which are the identifiers of the variables are written in binary, with unary constructors $0,1$ and the constant $\varepsilon$. To simplify the program, we suppose all identifiers to have the same length. $\mathtt{tt},\mathtt{ff}$ represent the boolean values *true* and *false*. $\vee$ serves for the disjunction. Since we focus on 3-SAT formulae, we take it to be a ternary function. For instance the formula $(x_1 \vee x_2 \vee \overline{x_3}) \wedge (x_1 \vee \overline{x_2} \vee \overline{x_1})$ is represented as:

$$\begin{aligned}&\mathbf{cons}(\vee(\mathtt{e}(0(1(\varepsilon))),\mathtt{e}(1(0(\varepsilon))),\neg(1(1(\varepsilon)))),\\ &\quad\mathbf{cons}(\vee(\mathtt{e}(0(1(\varepsilon))),\neg(1(0(\varepsilon))),\neg(0(1(\varepsilon)))),\mathbf{nil})).\end{aligned}$$

Recalling rules given in Example 1, the following program computes the satisfiability of a formula. Let us suppose that $\ell$ denotes the list of variables with the valuation "true", we have:

$$\begin{aligned}\mathtt{ver}(\mathbf{nil},\ell) &\to \mathbf{tt}\\ \mathtt{ver}(\mathbf{cons}(\vee(x_1,x_2,x_3),\psi),\ell) &\to \mathtt{and}(\mathtt{or}(\mathtt{or}(\mathtt{eval}(x_1,\ell),\mathtt{eval}(x_2,\ell)),\mathtt{eval}(x_3,\ell)),\mathtt{ver}(\psi,\ell))\\ \mathtt{eval}(\neg(n),\ell) &\to \mathtt{if}\ \mathtt{in}(n,\ell)\ \mathtt{then}\ \mathbf{ff}\ \mathtt{else}\ \mathbf{tt}\\ \mathtt{eval}(\mathtt{e}(n),\ell) &\to \mathtt{if}\ \mathtt{in}(n,\ell)\ \mathtt{then}\ \mathbf{tt}\ \mathtt{else}\ \mathbf{ff}\end{aligned}$$

[2] $\mathtt{e}$ corresponds to a positive occurrence of a variable. It is introduced for a question of uniformity.

It is sufficient to compute the set of "true" variable. This is done by the rules:

$$\begin{array}{rcl}
\mathtt{hyp}(\mathbf{nil}) & = & \mathbf{nil} \\
\mathtt{hyp}(\mathbf{cons}(\vee(\mathbf{a}(x_1),\mathbf{b}(x_2),\mathbf{c}(x_3)),\ell)) & \to & \mathtt{hyp}(\ell) \\
\mathtt{hyp}(\mathbf{cons}(\vee(\mathbf{a}(x_1),\mathbf{b}(x_2),\mathbf{c}(x_3)),\ell)) & \to & \mathbf{cons}(x_i,\mathtt{hyp}(\ell)) \\
\mathtt{hyp}(\mathbf{cons}(\vee(\mathbf{a}(x_1),\mathbf{b}(x_2),\mathbf{c}(x_3)),\ell)) & \to & \mathbf{cons}(x_i,\mathbf{cons}(x_j,\mathtt{hyp}(\ell))) \\
\mathtt{hyp}(\mathbf{cons}(\vee(\mathbf{a}(x_1),\mathbf{b}(x_2),\mathbf{c}(x_3)),\ell)) & \to & \mathbf{cons}(x_i,\mathbf{cons}(x_j,\mathbf{cons}(x_k,\mathtt{hyp}(\ell)))) \\
\mathtt{f}(\psi) & \to & \mathtt{ver}(\psi,\mathtt{hyp}(\psi))
\end{array}$$

with $\mathbf{a},\mathbf{b},\mathbf{c} \in \{\neg,\mathtt{e}\}$ and $i \neq j \neq k \in \{1,2,3\}$. The main function is $\mathtt{f}$.

The rules involving $\mathtt{hyp}$ are not confluent, and correspond exactly to the non-deterministic choice. (By Newman's Lemma, the systems considered are not weakly confluent since they are terminating.)

Such a program has an interpretation, given by:

$$\begin{array}{rcl}
(\!|\neg|\!)(x) = (\!|\mathtt{e}|\!)(x) & = & x+1 \\
(\!|\vee|\!)(x_1,x_2,x_3) & = & x_1+x_2+x_3+10 \\
(\!|\mathtt{eval}|\!)(x,y) & = & (x+1)\times y+3 \\
(\!|\mathtt{ver}|\!)(x,y) & = & (x+1)\times(y+1) \\
(\!|\mathtt{hyp}|\!)(x) & = & x+1 \\
(\!|\mathtt{f}|\!)(x) & = & (\!|\mathtt{ver}(x,\mathtt{hyp}(x))|\!)+1
\end{array}$$

Our notion of computation by a non-confluent system appears in Krentel's work [13], in a different context. It seems appropriate and robust, as argued by Grädel and Gurevich [8].

We suppose given a linear order $\prec$ on symbols, this order can be extended to terms using the lexicographic ordering. We use the same notation $\prec$ for this order. Then, we say that a (partial) function $\varphi : \mathbf{T}(\mathscr{C})^m \to \mathbf{T}(\mathscr{C})$ is computed by a program $\langle \mathscr{X},\mathscr{C},\mathscr{F},\mathscr{R}\rangle$ if for all $t_1,\ldots,t_m \in \mathbf{T}(\mathscr{C})$:

$$\varphi(t_1,\ldots,t_n) \text{ is defined} \Leftrightarrow \varphi(t_1,\ldots,t_n) = \max_{\prec}\{v \mid f(t_1,\ldots,t_n) \xrightarrow{!} v\}$$

In some case, we get the expected result: non-confluence corresponds exactly to non-determinism. Confluent programs with an interpretation compute PTIME, and the non-confluent ones compute NPTIME.

**Theorem 4** (Bonfante, Cichon, Marion and Touzet [2])**.** *Functions computed by non confluent programs with an additive polynomial interpretation are exactly* NPTIME *functions;*

### 3.2 Non confluent programs with a polynomial quasi-interpretation

The following result is more surprising.

**Theorem 5.** *Functions computed by non confluent programs that admit a quasi-interpretation and a PPO proof of termination are exactly* PSPACE *functions.*

The proof of the theorem essentially relies on the following example:

**Example 5.** [Quantified Boolean Formula] Let us compute the problem of the Quantified Boolean Formula. The principle of the algorithm is in two steps, the first one is top-down, the second one is bottom-up. In the first part, we span the computation to the leaves where we make an hypothesis on the value

of (some of) the variables. In the second part, coming back at the top, we compute the truth value of the formula and verify that the hypotheses chosen in the different branches are compatible between them.

As above, we suppose that variables are represented by binary strings on constructors $0, 1, \varepsilon$. To them, we add the constructors $\mathbf{cons}, \mathbf{nil}$ to build lists, $\blacktriangledown, \blacklozenge, \blacktriangle$ to decorate variables. Given a variable $n$, the decorations give the truth value of the variables. $\blacklozenge(n)$ corresponds to an unchosen value, that is true or false, $\blacktriangle(n)$ corresponds to false, and $\blacktriangledown(n)$ to true. The booleans are $\mathbf{tt}, \mathbf{ff}$ and $\perp$ serves for trash. $\mathbf{T}$ and $\mathbf{F}$ are two (unary) constructors representing booleans within computations. To help the reader, we give an informal type to the key functions: $\Psi$ corresponds to formulae, $\Lambda$ to lists (of decorated variables) and $B$ to truth values. Truth values are terms of the shape $\mathbf{T}(\Lambda)$ or $\mathbf{F}(\Lambda)$. Finally, $V$ is the type of variables and $\mathbb{B} = \{\mathbf{tt}, \mathbf{ff}\}$. The following rules correspond to the first step of the computation. $\mathtt{f} : \Psi \to B$, $\mathtt{ver} : \Psi \times \Lambda \to B$, $\mathtt{not} : B \to B$, $\mathtt{or} : B \times B \to B$, $\mathtt{vhyp} : B \times V \times \mathbb{B} \to B$, $\mathtt{put} : \Lambda \times V \times \mathbb{B} \to \Lambda$ and $\mathtt{hypList} : \Psi \to \Lambda$:

$$
\begin{array}{rcl}
\mathtt{f}(\varphi) & \to & \mathtt{ver}(\varphi, \mathtt{hypList}(\varphi)) \\
\mathtt{ver}(\mathbf{Var}(x), h) & \to & \mathbf{T}(\mathtt{put}(h, x, \mathbf{tt})) \\
\mathtt{ver}(\mathbf{Var}(x), h) & \to & \mathbf{F}(\mathtt{put}(h, x, \mathbf{ff})) \\
\mathtt{ver}(\mathbf{Or}(\varphi_1, \varphi_2), h) & \to & \mathtt{or}(\mathtt{ver}(\varphi_1, h), \mathtt{ver}(\varphi_2, h)) \\
\mathtt{ver}(\mathbf{Not}(\varphi), h) & \to & \mathtt{not}(\mathtt{ver}(\varphi, h)) \\
\mathtt{ver}(\mathbf{Exists}(x, \varphi), h) & \to & \mathtt{or}(\mathtt{vhyp}(\mathtt{ver}(\varphi, h), x, \mathbf{tt}), \mathtt{vhyp}(\mathtt{ver}(\varphi, h), x, \mathbf{ff}))
\end{array}
$$

where $h$ is a valuation of the variables. The rules $\mathtt{ver}(\mathbf{Var}(x), h) \to \mathbf{T}(\mathtt{put}(h, x, \mathbf{tt}))$ and $\mathtt{ver}(\mathbf{Var}(x), h) \to \mathbf{F}(\mathtt{put}(h, x, \mathbf{ff}))$ are the unique rules responsible of the non confluence of the program. This is the step where the value of variables is actually chosen. Concerning the valuations, they are written as lists $\mathbf{cons}(\mathrm{tv}(n), \mathbf{cons}(\cdots))$ where the truth value tv of a variable is in $\{\blacklozenge, \blacktriangledown, \blacktriangle\}$.

Suppose that $x$ is a variable ocuring in $\mathbf{Or}(\varphi_1, \varphi_2)$. One key feature is that in a computation of $\mathtt{ver}(\mathbf{Or}(\varphi_1, \varphi_2), h) \to \mathtt{or}(\mathtt{ver}(\varphi_1, h), \mathtt{ver}(\varphi_2, h))$, the choice of the truth value of the variable $x$ can be different in the two sub-computation $\mathtt{ver}(\varphi_1, h)$ and $\mathtt{ver}(\varphi_2, h)$. Then, the role of the bottom-up part of the computation is to verify that these choices are actually compatible.

The two functions $\mathtt{put}$ and $\mathtt{hypList}$ are computed by:

$$
\begin{array}{rcl}
\mathtt{hypList}(\mathbf{Var}(x)) & \to & \mathbf{nil} \\
\mathtt{hypList}(\mathbf{Or}(\varphi_1, \varphi_2)) & \to & \mathtt{append}(\mathtt{hypList}(\varphi_1), \mathtt{hypList}(\varphi_2)) \\
\mathtt{hypList}(\mathbf{Not}(\varphi)) & \to & \mathtt{hypList}(\varphi) \\
\mathtt{hypList}(\mathbf{Exists}(x, \varphi)) & \to & \mathbf{cons}(\blacklozenge(x), \mathtt{hypList}(\varphi)) \\
\mathtt{put}(\mathbf{cons}(\blacklozenge(n), l), m, \mathbf{tt}) & \to & \mathtt{if}\ \ n = m\ \mathtt{then}\ \ \mathbf{cons}(\blacktriangledown(n), l) \\
 & & \mathtt{else}\ \ \mathbf{cons}(\blacklozenge(n), \mathtt{put}(l, m, \mathbf{tt})) \\
\mathtt{put}(\mathbf{cons}(\blacklozenge(n), l), m, \mathbf{ff}) & \to & \mathtt{if}\ \ n = m\ \mathtt{then}\ \ \mathbf{cons}(\blacktriangle(n), l) \\
 & & \mathtt{else}\ \ \mathbf{cons}(\blacklozenge(n), \mathtt{put}(l, m, \mathbf{ff}))
\end{array}
$$

Then, the computation returns back. The top-down part of the computation returned a "tree" whose interior nodes are labeled with "`or`" and "`not`". At the leaves, we have $\mathbf{T}(l)$ or $\mathbf{F}(l)$ where $l$ stores the truth value of variables. The logical rules are:

$$\begin{array}{lcl} \mathtt{not}(\mathbf{F}(x)) & \to & \mathbf{T}(x) \\ \mathtt{not}(\mathbf{T}(x)) & \to & \mathbf{F}(x) \end{array} \qquad \begin{array}{lcl} \mathtt{or}(\mathbf{T}(x),\mathbf{T}(y)) & \to & \mathbf{T}(\mathtt{match}(x,y)) \\ \mathtt{or}(\mathbf{F}(x),\mathbf{T}(y)) & \to & \mathbf{T}(\mathtt{match}(x,y)) \\ \mathtt{or}(\mathbf{T}(x),\mathbf{F}(y)) & \to & \mathbf{T}(\mathtt{match}(x,y)) \\ \mathtt{or}(\mathbf{F}(x),\mathbf{F}(y)) & \to & \mathbf{F}(\mathtt{match}(x,y)) \end{array}$$

where $x$ and $y$ correspond to the list of hypothesis. The function `match` takes two lists and verify that they made compatible hypotheses on the truth value of variables. $\blacklozenge(n)$ is compatible with both $\blacktriangle(n)$ and $\blacktriangledown(n)$. But $\blacktriangle(n)$ and $\blacktriangledown(n)$ are not compatible.

$$\begin{array}{rcl} \mathtt{match}(\mathbf{nil},\mathbf{nil}) & \to & \mathbf{nil} \\ \mathtt{match}(\mathbf{cons}(x,l),\mathbf{cons}(x,l')) & \to & \mathbf{cons}(x,\mathtt{match}(l,l')) \\ \mathtt{match}(\mathbf{cons}(\blacktriangledown(x),l),\mathbf{cons}(\blacklozenge(x),l')) & \to & \mathbf{cons}(\blacktriangledown(x),\mathtt{match}(l,l')) \\ \mathtt{match}(\mathbf{cons}(\blacktriangle(x),l),\mathbf{cons}(\blacklozenge(x),l')) & \to & \mathbf{cons}(\blacktriangle(x),\mathtt{match}(l,l')) \\ \mathtt{match}(\mathbf{cons}(\blacklozenge(x),l),\mathbf{cons}(\blacktriangledown(x),l')) & \to & \mathbf{cons}(\blacktriangledown(x),\mathtt{match}(l,l')) \\ \mathtt{match}(\mathbf{cons}(\blacklozenge(x),l),\mathbf{cons}(\blacktriangle(x),l')) & \to & \mathbf{cons}(\blacktriangle(x),\mathtt{match}(l,l')) \\ \mathtt{match}(\mathbf{cons}(\blacktriangledown(x),l),\mathbf{cons}(\blacktriangle(x),l')) & \to & \bot \\ \mathtt{match}(\mathbf{cons}(\blacktriangle(x),l),\mathbf{cons}(\blacktriangledown(x),l')) & \to & \bot \end{array}$$

The matching process runs only for lists of equal length and variables must be presented in the same order. This hypothesis is fulfilled for our program. The last verification corresponds to the **Exists** constructor. For the left branch of the `or`, the variable $x$ is supposed to be true, for the second branch it is supposed to be false. This verification is performed by the `vhyp` function.

$$\begin{array}{rcl} \mathtt{vhyp}(\mathbf{T}(h),x,y) & \to & \mathbf{T}(\mathtt{vhyp}(h,x,y)) \\ \mathtt{vhyp}(\mathbf{F}(h),x,y) & \to & \mathbf{F}(\mathtt{vhyp}(h,x,y)) \\ \mathtt{vhyp}(\mathbf{cons}(\blacktriangledown(y),l),x,\mathbf{tt}) & \to & \mathtt{if}\ \ x=y\ \mathtt{then}\ \ l \\ & & \mathtt{else}\ \ \mathbf{cons}(\blacktriangledown(y),\mathtt{vhyp}(l,x,\mathbf{tt})) \\ \mathtt{vhyp}(\mathbf{cons}(\blacktriangle(y),l),x,\mathbf{tt}) & \to & \mathtt{if}\ \ x=y\ \mathtt{then}\ \ \bot \\ & & \mathtt{else}\ \ \mathbf{cons}(\blacktriangle(y),\mathtt{vhyp}(l,x,\mathbf{tt})) \\ \mathtt{vhyp}(\mathbf{cons}(\blacktriangle(y),l),x,\mathbf{ff}) & \to & \mathtt{if}\ \ x=y\ \mathtt{then}\ \ l \\ & & \mathtt{else}\ \ \mathbf{cons}(\blacktriangle(y),\mathtt{vhyp}(l,x,\mathbf{ff})) \\ \mathtt{vhyp}(\mathbf{cons}(\blacktriangledown(y),l),x,\mathbf{ff}) & \to & \mathtt{if}\ \ x=y\ \mathtt{then}\ \ \bot \\ & & \mathtt{else}\ \ \mathbf{cons}(\blacktriangledown(y),\mathtt{vhyp}(l,x,\mathbf{ff})) \end{array}$$

The rules for `if then else` , for $=$ and for `append` are omitted.

It is then routine to verify that this program is ordered by PPO. The order $\mathtt{f} \succ \mathtt{ver} \succ \mathtt{vhyp} \succ \mathtt{put} \succ \mathtt{or} \succ \mathtt{not} \succ \mathtt{hypList} \succ \mathtt{match} \succ \mathtt{if} \ \succ \mathtt{append} \succ =$ is compatible with the rules.

To end the Example, we provide a quasi-intepretation. $(\!|\mathtt{ver}|\!)(X,H) = X+H$, $(\!|\mathtt{f}|\!)(\Phi) = 2\Phi$, and for all other function symbols we take $(\!|f|\!)(X_1,\ldots,X_n) = \max(X_1,\ldots,X_n)$. For constructors, we take for all of them $(\!|\mathbf{c}|\!)(X_1,\ldots,X_n) = \sum_{i=1}^{n} X_i + 1$.

*of Theorem 5.* Let us begin with the following proposition.

**Proposition 5.** *F.QI.PPO is closed by composition. That is if $f, g_1, \dots g_k$ are computable with programs in F.QI.PPO, then, the function*

$$\lambda x_1, \dots, x_n . f(g_1(x_1, \dots, x_n), \dots, g_k(x_1, \dots, x_n))$$

*is computable by a program in F.QI.PPO.*

*Proof.* One adds a new rule $h(x_1, \dots, x_n) \to f(g_1(x_1, \dots, x_n), \dots, g_k(x_1, \dots, x_n))$ with precedence $h \succ f$ and $h \succ g_i$ for all $i \leq k$. The rule is compatible with the interpretation:

$$(\!|h|\!)(x_1, \dots, x_n) = (\!|f(g_1(x_1, \dots, x_n), \dots, g_k(x_1, \dots, x_n))|\!).$$

□

The Example 5 shows that a PSPACE-complete problem can be solved in the considered class of programs. By composition of QBF with the reduction, since polynomial time functions can be computed in F.QI.*PPO.n*, any PSPACE predicate can be computed in F.QI.*PPO.n*. Let us recall now that computing bit *i*th of the output of a PSPACE function is itself computable in PSPACE. Since, building the list of the first integers below some polynomials can be computed in polynomial time, by composition, the conclusion follows. □

### 3.3 Non confluent constructor preserving programs

**Theorem 6.** *Predicates computed by non confluent programs with a polynomially bounded constructor preserving interpretation are exactly the* PTIME*-computable predicates.*

This result is close to the one of Cook in [4] (Theorem 2). He gives a characterization of PTIME by means of auxiliary pushdown automata working in logspace, that is a Turing Machine working in logspace plus an extra (unbounded) stack. It is also the case that the result holds whether or not the auxiliary pushdown automata is deterministic.

The proof follows the line of [1], we propose thus just a sketch of the proof. The key observation is that arguments of recursive calls are sub-terms of the initial interpretation, a property that holds for confluent programs. As a consequence, following a call-by-value semantics, any arguments in sub-computations are some sub-terms of the initial interpretations. From that, it is possible to use memoization, see [10]. The original point is that we have to manage non-determinism.

The proof of Proposition 2 holds for non confluent computations. So, normal forms of a term $t$ are in $(\!|t|\!)$. As we have seen in the proof of Theorem 3, this set has a polynomial size wrt the size of the inputs. In the non deterministic case, the cache is still a map, with the same keys, but the values of the map enumerate the list of normal forms. With the preceding observation, we can state that the map has still a polynomial size.

**Acknowledgement.** I'd like to thank the anonymous referees for their precious help. Their sharp reading has been largely valuable to rewrite some part of the draft.

## References


[1] Guillaume Bonfante (2006): *Some programming languages for LOGSPACE and PTIME*. In: *11th International Conference on Algebraic Methodology and Software Technology - AMAST'06*, Kuresaare/Estonie.

[2] Guillaume Bonfante, Adam Cichon, Jean-Yves Marion & Hélène Touzet (2001): *Algorithms with polynomial interpretation termination proof*. *J. Funct. Program.* 11(1), pp. 33–53.

[3] Guillaume Bonfante, Jean-Yves Marion & Jean-Yves Moyen (2009): *Quasi-interpretations: a way to control resources*. *Theoretical Computer Science* To appear.

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

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

[6] Nachum Dershowitz & Jean-Pierre Jouannaud (1990): *Handbook of Theoretical Computer Science vol.B*, chapter Rewrite systems, pp. 243–320.

[7] Marco Gaboardi, Jean-Yves Marion & Simona Ronchi Della Rocca (2008): *A logical account of pspace*. *SIGPLAN Not.* 43(1), pp. 121–131.

[8] Erich Grädel & Yuri Gurevich (1995): *Tailoring recursion for complexity*. *Journal of symbolic logic* 60(3), pp. 952–69.

[9] Dieter Hofbauer & Clemens Lautemann (1988): *Termination proofs and the length of derivations*. *Lecture Notes in Computer Science* 355, pp. 167–177.

[10] Neil Jones (1997): *Computability and complexity, from a programming perspective*. MIT Press.

[11] Neil Jones (1999): *LOGSPACE and PTIME characterized by programming languages*. *Theroretical Computer Science* 228, pp. 151–174.

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

[13] Mark W. Krentel (1988): *The complexity of optimization problems*. *Journal of computer and system sciences* 36, pp. 490–519.

[14] Mukkai S. Krishnamoorthy & Paliath Narendran (1985): *On recursive path ordering*. *Theoretical Computer Science* 40(2-3), pp. 323–328.

[15] Lars Kristiansen (2006): *Complexity-Theoretic Hierarchies*. In: *CiE*, *Lecture Notes in Computer Science* 3988, Springer, pp. 279–288.

[16] Lars Kristiansen & Bedeho Mender (2009): *The Semantics and Complexity of Successor-free non deterministic Gödel T and PCF*. In: *Computability in Europe, CIE '09*, Heidelberg, Germany.

[17] D. Plaisted (1978): *A recursively defined ordering for proving termination of term rewriting systems*. Technical Report R-78-943, Department of Computer Science, University of Illinois.

[18] Olha Shkaravska, Marko van Eekelen & Ron van Kesteren (2009): *Polynomial Size Analysis of First-Order Shapely Functions*. *CoRR* abs/0902.2073. Available at `http://arxiv.org/abs/0902.2073`.

[19] TeReSe (2003): *Term Rewriting Systems*, *Cambridge Tracks in Theoretical Computer Science* 55. Cambridge University Press.