Some programming languages for LOGSPACE and PTIME

Guillaume Bonfante

Some programming languages for LOGSPACE and PTIME

Guillaume Bonfante

Loria, Calligramme project, B.P. 239, 54506 Vandœuvre-lès-Nancy Cedex, France,
and École Nationale Supérieure des Mines de Nancy, INPL, France.
Guillaume.Bonfante@loria.fr

Abstract. We propose two characterizations of complexity classes by means of programming languages. The first concerns LOGSPACE while the second leads to PTIME. This latter characterization shows that adding a choice command to a PTIME language (the language `WHILE` of Jones [1]) may not necessarily provide NPTIME computations. The result is close to Cook in [2] who used "auxiliary push-down automata". LOGSPACE is obtained through a decidable mechanism of tiering. It is based on an analysis of deforestation due to Wadler in [3]. We get also a characterization of NLOGSPACE.

We propose a contribution to the program of Jones [1]: "We maintain that Computability and Complexity theory, and Programming Language and Semantics [...] have much to offer each other, in both directions." ; we give characterizations of complexity classes by means (of restrictions) of programming languages.

The present contribution belongs to a largely wider program (see [4–6]) where we have shown the interest of that kind of characterization. Let us recall it briefly. From a practical point of view, a static analysis allows an evaluation of the bounds on the resources before computations are effectively performed. It can be used by an operating system to manage processes. In particular, it avoids the monitoring of memory usage. Maybe more interestingly, it can be used by the compiler to deal with memory management, and so, to optimize the complexity of programs. Such analyses are the theoretical core of projects like Amadio's CRISS project[1] whose objective is to control resources—time and space—for synchronous systems.

We propose a characterization of LOGSPACE. It is obtained with respect to a kind of tiering discipline. This fruitful approach has been initially considered by Bellantoni and Cook in [7] and Leivant and Marion [8] who characterized PTIME. Leivant and Marion showed that such a stratification could be used for other complexity classes, see [9, 10]. Following Bellantoni-Cook, Neergaard [11] has shown what restrictions of the language B leads to LOGSPACE. The current proposition differs from the preceding ones in the following way. The role of tiering is not to control recursion but rather to restrict the width of the call graph. It is essentially based on the work of Wadler [3] and Jones [1].

[1] `http://www.pps.jussieu.fr/~amadio/Criss/criss.html`

Among space characterizations, we mention the work of Hofmann [12, 13] who, in particular, showed how to compile functional programs to `malloc()`-free C. Applications of these techniques are to be found in the Embounded Project[2] whose aim is to quantify and certify resources for real-time embedded systems. An other approach, based on linear logic, was carried on by Baillot and Terui, see [14]. In the vein of Leivant, Oitavem proposed an interesting characterization of small complexity classes in [15, 16].

The second characterization we propose deals with non determinism. We show, and the result is surprising, that adding some choice command in the language `WHILE` of Jones does not change the class of computed functions. Naively, one would have expected to characterize NPTIME. Indeed, if one considers — as does Jones — the class of functions computed in polynomial time in `WHILE`, one gets PTIME. Adding the choice command, one gets NPTIME. Since `WHILE`-cons-free programs characterize PTIME, adding the choice command "should" have resulted in NPTIME. It is not the case, and we show that such a system characterize PTIME. The result is all the more surprising that for the corresponding space characterization, that is of LOGSPACE, adding the choice command leads to the corresponding non-deterministic complexity class NLOGSPACE. We mention here the work of Cook [2] whose characterization of PTIME by means of auxiliary pushdown automata is in essence close to us. The main difference lies in the fact that we have an implicit call stack (for recursion) where Cook has an explicit one.

The upper bound on the complexity of functions computed in LOGSPACE is obtained through a mechanism of compilation. The syntactical restrictions make the method sound. Two points. First, we do not explicitly compute time or space bounds. We know that they exist as a consequence of Jone's analysis of life without `cons`. Second, the proposition enters the field of implicit complexity as one has to compile the program in order to stay within LOGSPACE. Computing in the original framework leads to polynomial time computations.

The structure of the paper is as follows. In Section 1, we define the two programming languages we will consider, namely the language `WHILE` of Jones and a functional language. We define also some syntactical restrictions on them, in particular, we present our tiering discipline. In Section 2, we show the LOGSPACE characterization. At that point, we show how tiering can be used for the evaluation of programs. Section 3 deals with non-determinism, choice command and the corresponding complexity classes. We give a polynomial time procedure for computing non-deterministic `WHILE`-cons-free programs.

1 Programming Languages

We introduce two programming languages, `WHILE` programs, and `FOFP` programs. We suppose from now on, that we are given a signature *Cns*, that is, some symbols with their arity. These symbols are called the constructor symbols.

[2] `http://www.embounded.org/`

The signature defines a term algebra $\mathcal{T}(Cns)$ on which computations will be performed. We suppose that among these symbols one is `nil` of arity 0. The expression `nil` serves as "false". Any other value is "true".

For each constructor symbol $\mathbf{c}$, we define a set of destructor functions $\mathbf{d}_{\mathbf{c},k}$ which map $\mathbf{c}(t_1, \cdots, t_n) \mapsto t_k$. Finally, we suppose, for any constructor symbol $\mathbf{c}$, we are given a pattern matching function $\mathbf{p}_\mathbf{c}$ that tells whether a term has the form $\mathbf{c}(t_1, \cdots, t_n)$ or not.

1.1 The `WHILE` language

Let us begin with the syntax, which is due to Jones [1], except that we authorize more than one input. This is only for convenience.

Definition 1. *A* `WHILE` *program is given by the following grammar:*

$\mathtt{P} : \mathtt{Program}$	$::= \mathtt{read\ X1}, ..., \mathtt{Xn}; \mathtt{C}; \mathtt{write\ Y}$
$\mathtt{C} : \mathtt{Command}$	$::= \mathtt{Z} := \mathtt{E}$
	$\mid \mathtt{C}_1; \mathtt{C}_2$
	$\mid \mathtt{if\ E\ then\ C}_1 \mathtt{\ else\ C}_2$
	$\mid \mathtt{while\ E\ do\ C\ done}$
$\mathtt{E} : \mathtt{Expression}$	$::= \mathtt{Z}$
	$\mid \mathtt{D}$
	$\mid \mathbf{c}(\mathtt{E}_1, \mathtt{E}_2, \ldots, \mathtt{E}_k)$
	$\mid \mathbf{d}_{\mathbf{c},k}\ \mathtt{E}$
	$\mid \mathbf{p}_\mathbf{c}\ \mathtt{E}$
$\mathtt{X}, \mathtt{Y}, \mathtt{Z} : \mathtt{Variable}$	$::= \mathtt{X0} \mid \mathtt{X1} \mid \ldots$
$\mathtt{D} : \mathtt{Data} - \mathtt{value}$	$::= \mathcal{T}(Cns)$

We note $Var(\mathtt{p})$ *the variables appearing in a program* `p`.

The semantics are given by Jones. We recall it informally. A store for a program `p` is a function $\sigma^\mathtt{p} : Var(\mathtt{p}) \to \mathcal{T}(Cns)$. The initial store given the input data $\mathtt{d}_1, \mathtt{d}_2, \ldots, \mathtt{d}_n$ is the store $\sigma_0^\mathtt{p}(\mathtt{d}_1, \ldots, \mathtt{d}_n) = [\mathtt{X}_1 \mapsto \mathtt{d}_1, \ldots, \mathtt{X}_n \mapsto \mathtt{d}_n, \mathtt{Z} \mapsto \mathtt{nil}, \ldots, \mathtt{Y} \mapsto \mathtt{nil}]$. Commands have the intuitive meaning. For instance, in an expression $\mathtt{if\ E\ then\ C}_1 \mathtt{\ else\ C}_2$, one tests if $\mathtt{E} = \mathtt{nil}$, in which case one executes $\mathtt{C}_1$. Otherwise, one executes $\mathtt{C}_2$. Assignments modify the store.

Definition 2. *Given a program* `p`*, its execution induces a partial function* $[\![\mathtt{p}]\!] : \mathcal{T}(Cns)^\mathtt{n} \to \mathcal{T}(Cns)$ *which maps* $\mathtt{d}_1, \mathtt{d}_2, \ldots, \mathtt{d}_n$ *to* $\sigma(\mathtt{Y})$ *if the program terminates and where* σ *is the last store of the computation, otherwise it is undefined.*

Definition 3. *A program is called cons-free, if it does not use an expression of the form* $\mathbf{c}(\mathtt{E}_1, \ldots, \mathtt{E}_k)$*. We note* $\mathtt{WHILE}^{cons\text{-}free}$ *the set of cons-free programs.*

Theorem 1 (Jones [1]). *The set of decision problems computed by cons-free programs is exactly* LOGSPACE.

Definition 4. *The recursive extension of* `WHILE` *is described as follows. To* `WHILE`*, we add the instruction* `call` *that calls some sub-procedure. A program is given by*

```
globalvariable U_1, ..., U_u;
procedure P1; localvariable P11, ..., P1v;
C1;
procedure P2; localvariable P21, ..., P2w;
C2;
.......
read U1; call P1; write U1
```

Variables appearing in `Ci` *belong to the local variables of the procedure or to the global variables. The semantics are briefly as follows. Each time one calls a new procedure, one stacks some fresh local variables. Then, one executes the instructions until one reaches the end of the procedure (modifying the fresh local variables and the global ones). At this point, just forget the local variables. We note* $\texttt{WHILE}^{rec-cons-free}$ *the set of such programs.*

Theorem 2 (Jones [1]). *The set of decision problems computed by* $\texttt{WHILE}^{rec-cons-free}$ *programs is exactly the set of* PTIME *decision problems.*

1.2 FOFP

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

$$
\begin{array}{lll}
\texttt{Programs} \ni \mathbf{p} & ::= & d_1, \cdots, d_m \\
\texttt{Definitions} \ni d & ::= & \mathtt{f}(x_1, \cdots, x_n) = e^{\mathtt{f}} \\
\texttt{Expression} \ni e & ::= & x \mid \mathbf{op}(e_1, \cdots, e_n) \mid \mathtt{f}(e_1, \cdots, e_n) \\
 & & \mid \mathbf{c}(e_1, \cdots, e_n) \\
 & & \mid \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \\
 & & \mid \textbf{let } x = e_1 \textbf{ in } e_2 \\
 & & \mid \textbf{case } x_1, \cdots, x_n \textbf{ of } \overline{p_1} \rightarrow e_1 \ldots \overline{p_\ell} \rightarrow e_\ell \\
\texttt{Patterns} \ni p & ::= & x \mid \mathbf{c}(p_1, \cdots, p_n)
\end{array}
$$

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

Throughout the proofs which follows, we make no distinction between operators and function symbols. We have introduced operators only for convenience when writing the examples.

The set of variables Var is disjoint from Σ. In a definition, $e^{\mathtt{f}}$ is called the body of `f`. A variable of $e^{\mathtt{f}}$ is either a variable in the parameter list $x_1, \cdots, x_n$ of the definition of `f` or a variable which occurs in a pattern of a case definition.

In a case expression, patterns are supposed to be non overlapping. We will come back to this Hypothesis in the Section on non determinism.

Given a function symbol $\mathtt{f}$, we say that an expression e is $\mathtt{f}$-free if there is no occurrences of $\mathtt{f}$ in e. We call functional an expression of the form $\mathtt{g}(e_1, \cdots, e_n)$.

Lastly, it is convenient, because it avoids tedious details, to restrict our attention to programs without nested **case** , **let** , **if** expressions within functional expressions. This is not a severe restriction as one can easily transform programs to avoid this nesting. For instance, one transforms $\mathtt{f}(\ldots, \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3, \ldots)$ into $\textbf{if } e_1 \textbf{ then } \mathtt{f}(\ldots, e_2, \ldots) \textbf{ else } \mathtt{f}(\ldots, e_3, \ldots)$.

Definition 5. *Rules for evaluation are given by Fig. 1. A function $f : \mathcal{T}(Cns)^k \to \mathcal{T}(Cns)$ is computed by a program $\boldsymbol{p}$ if there is a function $\boldsymbol{f} \in \boldsymbol{p}$ such that $\forall \overline{t} \in \mathcal{T}(Cns)^k : \boldsymbol{f}(\overline{t}) \downarrow f(\overline{t})$.*

From now on, we suppose that the programs that we consider are terminating. Any method for proving their termination can be considered, for instance Recursive Path Orderings, Dependency Pairs, and so on.

$$\frac{t \in \mathcal{T}(Cns)}{t \downarrow t} \qquad \frac{e_1 \downarrow v_1 \ldots e_n \downarrow v_n \quad \mathtt{f}(x_1, \cdots, x_n) = e^{\mathtt{f}} \quad e^{\mathtt{f}}[x_i \leftarrow v_i] \downarrow v}{\mathtt{f}(e_1, \cdots, e_n) \downarrow v}$$

$$\frac{e_1 \downarrow \mathtt{tt} \quad e_2 \downarrow v}{\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \downarrow v} \qquad \frac{e_1 \downarrow \mathtt{ff} \quad e_3 \downarrow v}{\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \downarrow v}$$

$$\frac{e_1 \downarrow u \quad e_2[x \leftarrow u] \downarrow v}{\textbf{let } x = e_1 \textbf{ in } \ e_2 \downarrow v} \qquad \frac{e_k \downarrow u_k \quad \exists \sigma,\ i \ :\ \overline{p_i}\sigma = \overline{u} \quad e_i\sigma \downarrow v}{\textbf{case } t_1, \cdots, t_n \textbf{ of } \overline{p_1} \to e^1 \ldots \overline{p_\ell} \to e^\ell \downarrow v}$$

Fig. 1. Call by value semantics of a program **p**

Definition 6. *We say that a program $\boldsymbol{p}$ is cons-free if the definitions do not use the rule $\mathbf{c}(e_1, \cdots, e_n)$ of the grammar. In other words, there are only constructors in patterns. The set of such cons-free programs is noted $\mathtt{FOFP}^{cons\text{-}free}$.*

Definition 7. *A definition $\boldsymbol{f}(x_1, \cdots, x_n) = e^f$ induces a relation on function symbols. Say that $\boldsymbol{f}$ calls $\boldsymbol{g}$ if $\boldsymbol{g}$ appears in the body of $\boldsymbol{f}$. We note this relation $\to$. The reflexive-transitive closure of this relation induces a pre-order on function symbols, noted $\overset{*}{\to}$. The corresponding equivalence relation $\simeq$ is defined by $\boldsymbol{f} \simeq \boldsymbol{g} \Leftrightarrow (\boldsymbol{f} \overset{*}{\to} \boldsymbol{g} \wedge \boldsymbol{g} \overset{*}{\to} \boldsymbol{f})$. The corresponding strict partial order is noted $\prec$. We have $\boldsymbol{g} \prec \boldsymbol{f} \Leftrightarrow (\boldsymbol{f} \overset{*}{\to} \boldsymbol{g} \wedge \neg(\boldsymbol{f} \overset{*}{\to} \boldsymbol{g}))$.*

Definition 8. *We say that an expression e is tail-recursive w.r.t. a function symbol $\boldsymbol{f}$ if*

1. $e = x$,
2. $e = \mathtt{g}(e_1, \cdots, e_n)$ *where for all* $\mathtt{h} \in e$, $\mathtt{h} \prec \mathtt{f}$,
3. $e = \mathtt{f}(x_1, \cdots, x_n)$,
4. $e =$ **if** e_1 **then** e_2 **else** e_3 *and* e_1 *is* $\mathtt{f}$*-free and both* e_2 *and* e_3 *are tail recursive wrt* $\mathtt{f}$,
5. $e =$ **case** $\overline{x}$ **of** $\overline{p_1} \to e_1 \ldots \overline{p_\ell} \to e_\ell$ *and for all* $i \leq \ell$ *the expression* e^k *is tail recursive.*

A definition $\mathtt{f}(x_1, \cdots, x_n) = e^{\mathtt{f}}$ *is tail recursive if* $e^{\mathtt{f}}$ *is tail recursive wrt* $\mathtt{f}$. *A program is tail recursive if any definition is tail recursive. The set of such tail recursive programs is noted* $\mathtt{FOFP}^{tr}$. *We note* $\mathtt{FOFP}^{tr\text{-}cons\text{-}free}$ *the set of programs that are both tail recursive and cons-free.*

The following is due to Jones [17].

Theorem 3.

1. *The set of decision problems computed by* $\mathtt{FOFP}^{cons-free}$ *programs is exactly the set of* PTIME *decision problems.*
2. *The set of decision problems computed by* $\mathtt{FOFP}^{tr-cons-free}$ *programs is exactly the set of* LOGSPACE *decision problems.*

In the following, we reinforce Definition 8 to allow nesting of functions. We propose a finer discipline on programs that stays within LOGSPACE.

Example 1.

$$\begin{aligned} x_1 < x_2 = \mathbf{case}\ x_1, x_2\ \mathbf{of} \\ x_1', \mathbf{0} &\to \mathbf{ff} \\ \mathbf{0}, \mathbf{s}(x_2') &\to \mathbf{tt} \\ \mathbf{s}(x_1'), \mathbf{s}(x_2') &\to x_1' < x_2' \end{aligned} \qquad \begin{aligned} x_1 - x_2 = \mathbf{case}\ x_1, x_2\ \mathbf{of} \\ x_1', \mathbf{0} &\to x_1 \\ \mathbf{0}, x_2' &\to \mathbf{0} \\ \mathbf{s}(x_1'), \mathbf{s}(x_2') &\to x_1' - x_2' \end{aligned}$$

$$\begin{aligned} \mathtt{pgcd}(x_1, x_2) = \mathbf{case}\ x_2\ \mathbf{of} \\ \mathbf{0} &\to x_1 \\ \mathbf{s}(x_2') &\to \mathbf{if}\ x_1 < x_2 \\ &\quad \mathbf{then}\ \ \mathtt{pgcd}(x_2, x_1) \\ &\quad \mathbf{else}\ \mathtt{pgcd}(x_1 - x_2, x_2) \end{aligned}$$

Here the first two definitions are tail-recursive. This is not the case of the third expression. Note that it cannot be directly handled by Wadler's approach, see [3], as there is some composition of function symbols. Note also that there is more than one occurrence of `pgcd` in the right hand side of the second rule. In the following, we show how to compute `pgcd` in LOGSPACE.

Definition 9. *An expression is strongly tail-recursive if it follows Definition 8 except that clauses (2) and (3) are replaced by clauses*

2'. $e = \mathtt{g}(e_1, \cdots, e_n)$ *and for all the* $i \leq n$, e_i *is* $\mathtt{f}$*-free. Here,* $\mathtt{g}$ *may be equal to* $\mathtt{f}$;

6'. $e = \textbf{let } x = e_1 \textbf{ in } \ e_2$ *where* e_1 *is* f*-free and* e_2 *is strongly tail-recursive.*

This extends to the definition of function symbols and to programs. We note the set of such programs $\texttt{FOFP}^{s-tr}$*.*

One may observe that the above definition of the `pgcd` respects the strong-tail recursiveness condition. The next Section shows that programs in $\texttt{FOFP}^{s-tr-cons-free}$ can be computed within Logspace.

Theorem 4. *The set of decision problems computed by* $\texttt{FOFP}^{s-tr-cons-free}$ *programs is exactly the set of* Logspace *decision problems.*

Finally, we propose a notion that goes beyond strong tail-recursion.

Definition 10 (Linear programs). *Given a function symbol* f*, the level of an expression is given by the inductive rules:*

- $lvl_f(x) = 0$,
- $lvl_f(g(e_1, \cdots, e_n)) = 1 + \sum_{k \leq n} lvl_f(e_k)$ *where* $g \simeq f$,
- $lvl_f(g(e_1, \cdots, e_n)) = \sum_{k \leq n} lvl_f(e_k)$ *where* $g \prec f$,
- $lvl_f(\textbf{let } x = e_1 \textbf{ in } e_2) = lvl_f(e_1) + lvl_f(e_2)$,
- $lvl_f(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3) = lvl_f(e_1) + \max(lvl_f(e_2), lvl_f(e_3))$,
- $lvl_f(\textbf{case } \overline{x} \textbf{ of } \ p_1 \to e_1, \ldots, p_k \to e_k) = \max(lvl_f(e_1), \ldots, lvl(e_k))$.

We say that a definition $f(\overline{x}) = e^f$ *is linear if* $lvl_f(e^f) = 1$*. A program is linear if any definition has level* 1*. The set of such programs is noted* $\texttt{FOFP}^{lin}$*.*

Theorem 5. *Decision problems decided by linear cons-free programs are exactly* Logspace *decision problems.*

Example 2. The following program is not strongly-tail-recursive but linear.

$$
\begin{aligned}
\texttt{pred}(x) &= \textbf{case } \ x \textbf{ of} \\
&\quad\ \ \mathbf{0} \to \mathbf{0} \\
&\quad\ \ \mathbf{s}(x') \to x' \\
\texttt{incr}(x, y) &= y - \texttt{pred}(y - x) \\
\texttt{log}(x) &= \textbf{case } \ x \textbf{ of} \\
&\quad\ \ \mathbf{0} \to \mathbf{0} \\
&\quad\ \ \mathbf{s}(x') \to \texttt{incr}(\texttt{log}(\texttt{half}(x)), x)
\end{aligned}
\qquad\qquad
\begin{aligned}
\texttt{half}(x) &= \textbf{case } \ x \textbf{ of} \\
&\quad\ \ \mathbf{0} \to \mathbf{0} \\
&\quad\ \ \mathbf{s}(\mathbf{0}) \to \mathbf{0} \\
&\quad\ \ \mathbf{s}(\mathbf{s}(x')) \to \texttt{incr}(\texttt{half}(x), x)
\end{aligned}
$$

2 Compiling FOFP programs

The proofs of the Theorems 4, 5 involve the same argument. We compile programs in $\texttt{FOFP}^{s-tr-cons-free}$ and in $\texttt{FOFP}^{lin-cons-free}$ to `WHILE`-cons-free. As a consequence, function computable in these two languages can be computed within Logspace due to Theorem 3. The converse part, that is to show that all Logspace decidable problems can be computed by strongly-tail-recursive programs or linear-cons-free programs is a direct consequence of the fact that $\texttt{FOFP}^{tr-cons-free} \subseteq \texttt{FOFP}^{s-tr-cons-free} \subseteq \texttt{FOFP}^{lin-cons-free}$.

We first begin with a few observations.

Proposition 1. *Given a program in* $\texttt{FOFP}^{cons-free}$*, for any constructor terms* $\bar{t}$*, and any expression* e*, suppose that* $e[x_i \leftarrow t_i] \downarrow v$*, then,* v *is a subterm of one of the* t_i*.*

Corollary 1. *For a* $\texttt{FOFP}^{cons-free}$ *program, the height of the evaluation tree (cf. Fig. 1) is bounded by a polynomial in the input.*

Proof. Given a term t, the number of subterms of t is linear in the size of t. Suppose we are given a constant d. Consider the set $C_{\bar{t}} = \{s_1, \cdots, s_n \mid n \leq d \wedge (\forall i \leq n : \exists j \leq n : s_i \trianglelefteq t_j)\}$. Then, this set has polynomial size in the size of $\bar{t}$. Now, take d to be the maximal arity of a symbol, the polynomial bound together the property of termination of programs and the preceding proposition gives the result.

Proposition 2. *Suppose that* $\mathtt{f}_1, \ldots, \mathtt{f}_k$ *are* FOFP *programs which are* Logspace *computable. Then any function* $\mathtt{f}(x_1, \ldots, x_n) = e^{\mathtt{f}}$ *with* $e^{\mathtt{f}}$ *a composition of the functions* $\mathtt{f}_1, \ldots, \mathtt{f}_k$ *is computed by a* WHILE*-cons free program, and so, is* Logspace*. Given an expression* e*, we note the corresponding code* $\mathtt{C_e}$*.*

Proof. The proof is by induction on the expression $e^{\mathtt{f}}$. Suppose that we are given for each function $\mathtt{f}_i$ a WHILE-cons-free program $\mathtt{read\ Xfi1, Xfi2, \ldots, Xfik; C_{fi}; write\ Y}$; that computes it. We suppose w.l.o.g that these programs do not change de values of the input variables. Suppose that $e^{\mathtt{f}} = x_k$, it is computed by:

```
read X1, X2, ..., Xn;
Y := Xk;
write Xk;
```

Suppose now that $e^{\mathtt{f}} = \mathtt{g}(\overline{e})$. Then, by induction we can suppose that e_i is computed by $\mathtt{C_{e_i}}$. In that case, the following WHILE-cons-free program computes f.

```
read X1, ..., Xn;
Ce1;
X1g := Y;
Ce2;
X2g := Y;
.
.
.
Cek;
Xkg := Y;
Cg;
write Y;
```

Remark 1. It is well known that if f and g are Logspace, so is $f \circ g$. Since the output of functions are subterms of the inputs, we have a much easier proof of the composition. Furthermore, we use the construction throughout the paper. This is why we give an explicit construction.

2.1 Strongly-tail-recursive programs

We prove now Theorem 4. First of all, let us eliminate the **let** construction.

Proposition 3. *Consider the program transformation that maps* **let** $x = e_1$ **in** e_2 *to* $e_2[x \leftarrow e_1]$. *It preserves the semantics of the program. Furthermore, if a program is strongly-tail recursive, so is its transform.*

As a consequence, we may consider w.l.o.g only programs without the **let** construction.

For each definition, we build a WHILE-cons-free program that computes it. We proceed by induction on the ordering $\prec$. For the sake of the proof, to avoid a tedious case analysis, we suppose that for all symbols $\mathtt{g} \in e^{\mathtt{f}}$, either $\mathtt{g} = \mathtt{f}$ or $\mathtt{g} \prec \mathtt{f}$. In other words, we avoid mutual definitions.

For minimal elements, observe that their definitions are tail-recursive. So, one applies Theorem 3 to get a WHILE-cons-free programs that computes them.

Now, we suppose that we are given an expression $e^{\mathtt{f}}$ that computes $\mathtt{f}$. We suppose by induction that we have a program computing any function $\mathtt{g} \neq \mathtt{f}$ involved in the definition of $\mathtt{f}$. As a consequence, applying Proposition 2, one gets for each composition of such functions some program that computes it.

We now perform an induction on the structure of the definition of $\mathtt{f}$. In the following compiling procedure, we suppose (by induction) that we are given for any sub-expression e some program $\mathtt{read\ X_{e1}, \ldots, X_{en}; C_e; writeY}$; where the $\mathtt{X}_{e_i}$ are the variables of e_i .

By compositions of pattern expression $\mathbf{p_c}$ as well as destructors $\mathbf{d_c}$, we build for each pattern p a code $P_p \in \mathtt{WHILE}^{cons-free}$ that returns $\mathtt{tt}$ if the inputs verify the pattern. We suppose given the code for the unification of variables in patterns. So, after $\overline{X'_p} :=_p \overline{X}$, the variables in the patterns are supposed to have their value after pattern matching.

Given an expression e, Table 1 gives rules to build the code $\mathtt{D_e}$.

$\mathtt{D}_{\mathbf{case}\ \overline{x}\ \mathbf{of}\ p_1 \to e_1, \ldots, p_k \to e_k}$	$\mathtt{D}_{\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3}$	$\mathtt{D}_{f(e_1,\cdots,e_n)}$	$\mathtt{D}_{g(\overline{e})}$
P_{p_1};		$\mathtt{C_{e_1}}$;	
if Y		$\mathtt{X_1} := \mathtt{Y}$;	
then $\overline{\mathtt{X'_{p_1}}} :=_{p_1} \overline{\mathtt{X}}; \mathtt{D_{e_1}}$;	$\mathtt{C_{e_1}}$;	$\mathtt{C_{e_2}}$;	
else $\mathtt{P_{p_2}}$;	if Y	$\mathtt{X_2} := \mathtt{Y}$;	$\mathtt{C_{g(\overline{e})}}$;
if Y	then $\mathtt{D_{e_2}}$;	$\vdots$	$\mathtt{R} := \mathtt{ff}$;
then $\overline{\mathtt{X'_{p_2}}} :=_{p_2} \overline{\mathtt{X}}; \mathtt{D_{e_2}}$;	else $\mathtt{D_{e_3}}$;	$\mathtt{C_{e_k}}$;	
$\vdots$		$\mathtt{X_k} := \mathtt{Y}$;	
$\overline{X'_{p_k}} :=_{p_3} \overline{X}; \mathtt{D_{e_k}}$;			

Table 1. Compilation rules

What do these programs do? If the expression is $\mathtt{f}$-free, that is if we reached the end of the recursion, $\mathtt{Y}$ is assigned the value of the computation. In that case,

the flag variable R is turned to tt. Otherwise, it computes the arguments of the next call of f, and continues the process.

So, the function f is computed by the following program:

```
read X_1, ··· , X_n;
R := tt;
while R do
De^f;
write Y;
```

2.2 Linear programs

The rationale behind the compilation of linear programs is the following. In the first part of the computation, we just compute the arguments of the intermediate calls of f and forget the context in which they appear. At the end of the process, one knows two crucial points.

First, one knows the exact number of nested calls of f, moreover, due to Corollary 1, this number is polynomial in the size of the input. As a consequence, it is representable in log-space. Second, one knows the value of the function f on its terminating call.

In the second part, you just redo what has been said above except that at each step you compute one less nesting of calls of f and reuse the last result of the loop to compute the value of f in its full context.

Contrarily to what happened for strong-tail-recursion, we cannot get rid off the **let** construction. Indeed, the transform does not preserve linearity. A counter-example is the definition $\mathtt{f}(x) = \mathbf{case}\ x\ \mathbf{of}\ (\mathbf{0} \to \mathbf{0}, \mathbf{s}(x') \to \mathbf{let}\ y = \mathtt{f}(x')\ \mathbf{in}\ \mathtt{g}(y, y))$ where g is an already defined binary function. So, don't forget we have to cope with **let** .

Proof. As above, we proceed by induction on the $\prec$ order. For minimal elements, the definition is tail-recursive. For those, we have already seen that we have a procedure. Suppose now that we are trying to compute function f whose definition is $\mathtt{f}(x_1, \cdots, x_n) = e^{\mathtt{f}}$. As above, we suppose that there is no symbol equivalent to f in $e^{\mathtt{f}}$ (except for f of course!). We suppose that we have built for any function $\mathtt{g} \prec \mathtt{f}$ some WHILE-cons-free code that computes it. Moreover, using Proposition 2, we suppose that we are able to compute any expression composed of such symbols.

Now, suppose we are given a functional term $\mathtt{h}(\overline{e})$ of level 1. It contains one occurrence of f. It can be seen as $C[\mathtt{f}(\overline{e'})]$ where the context C can be seen as an expression over the variables of $\mathtt{h}(\overline{e})$ plus an extra variable F that corresponds to the call of f. We can suppose that for any of these expressions, we have some code that computes them. We use a similar notation to that of the proof above.

To compute the value of the last call, we build a code analogous to what we have done for strong-tail recursion. The rules are somewhat different.

– For the **case** construction, we use the construction of Table 1.

– The **if** case splits into two sub-cases. When $\texttt{lvl}_\texttt{f}(e_1) = 0$, we use the rule of Table 1. The other case, is shown below.
– The **let** construction also splits into two parts that are considered below.
– The last case correspond to the functional expressions. When the functional expression has level 0, we use the rule of Table 1. The other case is presented below.

$\texttt{D}_{\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3}$ when $\texttt{lvl}_{\texttt{f}_1}(e_1) = 1$	$\texttt{D}_{\textbf{let } \texttt{v}=e_1 \textbf{ in } e_2}$ when $\texttt{lvl}_\texttt{f}(e_1) = 0$	$\texttt{D}_{\textbf{let } \texttt{v}=e_1 \textbf{ in } e_2}$ when $\texttt{lvl}_\texttt{f}(e_1) = 1$	$\texttt{D}_{\texttt{C}[\texttt{f}(\overline{e'})]}$
$\texttt{D}_{e_1}$	$\texttt{C}_{e_1}$; $\texttt{V} := \texttt{Y}; \texttt{D}_{e_2}$	$\texttt{D}_{e_1}$	$\texttt{C}_{e'_1}$; $\texttt{X}_1 := \texttt{Y}$; $\texttt{C}_{e'_2}$; $\texttt{X}_2 := \texttt{Y}$; $\vdots$ $\texttt{C}_{e'_k}$; $\texttt{X}_k := \texttt{Y}$;

Given an expression e, we define now (by induction) a code $\texttt{E}_e$ that computes the value of `f` given that `F` contains the value of the sub-call of `f`.

$\texttt{E}_{\textbf{case } \overline{\texttt{x}} \textbf{ of } \texttt{p}_1 \to e_1, \ldots, \texttt{p}_k \to e_k}$	$\texttt{E}_{\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3}$ when $\texttt{lvl}_\texttt{f}(e_1) = 0$	$\texttt{E}_{\textbf{let } \texttt{v}=e_1 \textbf{ in } e_2}$ when $\texttt{lvl}_\texttt{f}(e_1) = 0$
P_{p_1}; `if Y` `then` $\overline{X'_{p_1}} :=_{p_1} \overline{X}; \texttt{E}_{e_1}$; `else` $\texttt{P}_{p_2}$; `if Y` `then` $\overline{X'_{p_2}} :=_{p_2} \overline{X}; \texttt{E}_{e_2}$; $\vdots$ $\overline{X'_{p_k}} :=_{p_k} \overline{X}; \texttt{E}_{e_k}$;	$\texttt{C}_{e_1}$; `if Y` `then` $\texttt{E}_{e_2}$; `else` $\texttt{E}_{e_3}$;	$\texttt{C}_{e_1}$; $\texttt{V} := \texttt{Y}$; $\texttt{E}_{e_2}$;

$\texttt{E}_{\texttt{C}[\texttt{f}(\overline{e'})]}$	$\texttt{E}_{\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3}$ when $\texttt{lvl}_\texttt{f}(e_1) = 1$	$\texttt{E}_{\textbf{let } \texttt{v}=e_1 \textbf{ in } e_2}$ when $\texttt{lvl}_\texttt{f}(e_1) = 1$
$\texttt{C}_\texttt{C}$;	$\texttt{E}_{e_1}$; `if Y` `then` $\texttt{C}_{e_2}$; `else` $\texttt{C}_{e_3}$;	$\texttt{E}_{e_1}$; $\texttt{V} := \texttt{Y}$; $\texttt{C}_{e_2}$;

We can now compute f by the following code:

```
read X_{1,0}, ..., X_{n,0};
X̄_i := X̄_{i,0}; //a copy of the inputs
R := tt;
while R do
  D_{e^f}; incr N;
done; N := pred N;
while N ≠ 0 do
X̄_i := X̄_{i,0};
  M = N;
  while M ≠ 0 do
    D_{e^f}; M := pred M;
  done;
  N := pred N;
  E_{e^f}; F := Y;
done;
Y := F;
writeY;
```

Some last words about this code. Our management of the counters N and M is licit, even the incrementing, since we have a polynomial bound due to Proposition 1 on the two counters. We refer to Jones [17] who extensively discusses how to carry this out.

3 Non-determinism

This part of the paper introduces some "non-determinism" to the languages. To WHILE, we add a new command choose. We propose non-confluence as a functional correspondence of this instruction.

Definition 11. *Following Jones, to* WHILE*we add the expression* choose C_1 C_2 *whose operational semantics is to evaluate either* C_1 *or* C_2*. A program induces now a relation between inputs and outputs. We say that a decision problem f is computed by a program* f *if for all inputs* $\bar{t}$*, the value of* $f(\bar{t})$ *is true iff one execution of* f *on* $\bar{t}$ *reaches* tt*. We note* WHILE^n *the set of programs with this extra instruction.* $\text{WHILE}^{n\text{-}ptime}$ *denotes the set of (non deterministic) programs working in polynomial time, etc.*

Theorem 6 (Jones [17]).

1. $\text{WHILE}^{n\text{-}ptime} = \text{NP}_{\text{TIME}}$;
2. $\text{WHILE}^{n-log-space} = \text{NL}_{\text{OGSPACE}}$.

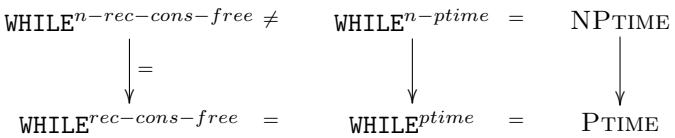
Definition 12. *We consider here some* FOFP *programs without the confluence property, that is, patterns may overlap each other. A normal form is one possible result of the computation. Following Grädel and Gurevich [18], the value of any*

term is the maximal normal form of the term (for a given order on terms). Notice that this includes the usual definition for decision problem by choosing ***true*** $>$ ***false****. We add the superscript n to denote the fact that we include non-deterministic programs.*

Theorem 7.

1. $\texttt{WHILE}^{n-cons-free} = \texttt{FOFP}^{n-lin-cons-free} = \text{NLogspace}$;
2. $\texttt{WHILE}^{n-rec-cons-free} = \texttt{FOFP}^{n\text{-}cons\text{-}free} = \text{Ptime}$.

This latter fact is surprising as it breaks a similarity (similarity that holds for logspace):



This result is analogous to that of Cook [2] Th.2 p7. 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.

3.1 Bound on $\texttt{FOFP}^{\text{n-cons-free}}$

We propose a proof for FOFP-programs. The case of WHILE-programs is similar. The key observation is that Proposition 1 remains true in the context of non-confluent programs. As a consequence, following a call-by-value semantics, any arguments in subcomputations are some subterms of the initial inputs. From that, it is possible to use memoization, see [17]. The original point is that we have to manage non-determinism.

So, the crucial point is that the arguments of functions are subterms of the input and moreover, that the cardinality of this set is polynomial as was shown in Proposition 1. A second point is that normal forms are also subterms of the input. It means that, for each defined symbol, the induced relation can be stored in polynomial space. This leads to a procedure where we remember the normal forms of each (already computed) function on arguments and reuse it when necessary.

Suppose we are given a program $\texttt{f}$ which is n-cons-free. Given input $t_1, \cdots, t_n$, let us note $I = \{t \trianglelefteq t_i \mid i \leq n\}$. We have $\sharp I \leq O(|t_1, \cdots, t_n|)$.

We consider a 3D table. The first dimension corresponds to $\mathcal{F}$, the second to I^A (where A is the maximal arity of a symbol), that is the arguments of functions. The third to I, the possible values of the relation. The entries of the table are boolean, and $\mathrm{T}[\mathsf{g}][t][v]$ is (intended to be) true iff $\mathsf{g}(t) \xrightarrow{+} v$. This table has a polynomial size w.r.t. the inputs.

Consider the following algorithm (at the beginning, the entries of table `T[g][t][v]` are false):

```
var r : Term;
for i := 1 to |F| * O(|t1,...,tn|^A) * O(|t1,...,tn|) do
for g in F do
for t in I^A do
for v in I do
        r1, ..., rn := find(g,t);
        for l := 1 to n do
        T[g][t][v] := T[g][t][v] || compute(ri,t,v);
        end
end;
end;
end;
end;
```

`find(g,t)` is charged to give the list of all rules that can be applied on t. There are finitely many of these. This can be done in linear time.

`compute(ri,t,v)` is charged to see if the rule `ri` given by `find` may lead to value v. That is, to see if the subcalls (with the corresponding inputs) in `r` have been already computed, choose for all of them the already computed values and finally turns the table cell to true if one of these choices leads to the value `v`. This process is easily proved polynomial by a simple induction on the construction of the rule `ri`. So, the instructions inside the loop take polynomial time.

For each loop on `i`, one will fulfil some of the $T[\mathtt{g}][t][v]$. As a consequence, the bound on the exterior loop is enough to get the result. So, the fixpoint is reached within a polynomial in the number of entries in the table. This algorithm works in polynomial time, hence we obtain the following corollary:

Corollary 2. $\mathtt{FOFP}^{n-cons-free} \subseteq$ PTIME.

Concerning the counterpart of the proof. In the case of PTIME, one may note that $\mathtt{FOFP}^{cons-free} \subseteq \mathtt{FOFP}^{n-cons-free}$. As a consequence, w.r.t. Theorem 3, it is PTIME complete.

3.2 $\mathtt{FOFP}^{\text{n-lin-cons-free}}$

First, proving that $\mathtt{WHILE}^{nlogspace} \simeq \mathtt{WHILE}^{n-cons-free}$ can be achieved following Jones's proof that $\mathtt{WHILE}^{logspace} \simeq \mathtt{WHILE}^{cons-free}$. Here, non-determinism plays no special role.

For the case of non-deterministic linear programs, one may note that the rules for the case analysis can be transformed to take into account the fact that more than one pattern applies. At this point, use the choice operator to decide which pattern to take.

As a consequence, the analysis of Section 2 can be used here. So, by the remark at the beginning of the subsection, the global process can be performed within NLOGSPACE.

Acknowledgment This work has been largely inspired by "life without `cons`" of Jones.

References

1. Jones, N.D.: LOGSPACE and PTIME characterized by programming languages. Theoretical Computer Science **228** (1999) 151–174
2. Cook, S.: Characterizations of pushdown machines in terms of time-bounded computers. Journal of the ACM **18(1)** (1971) 4–18
3. Wadler, P.: Deforestation: Transforming programs to eliminate trees. In: ESOP '88. European Symposium on Programming, Nancy, France, 1988 (Lecture Notes in Computer Science, vol. 300), Berlin: Springer-Verlag (1988) 344–358
4. Marion, J.Y., Moyen, J.Y.: Efficient first order functional program interpreter with time bound certifications. In: LPAR 2000. Volume 1955 of Lecture Notes in Computer Science., Springer (2000) 25–42
5. Bonfante, G., Marion, J.Y., Moyen, J.Y.: On lexicographic termination ordering with space bound certifications. In: PSI 2001, Ershov Memorial Conference. Volume 2244 of Lecture Notes in Computer Science., Springer (2001)
6. Bonfante, G., Marion, J.Y., Moyen, J.Y.: Quasi-Interpretations and Small Space Bounds. In Giesl, J., ed.: Rewrite Techniques and Applications. Volume 3467 of Lecture Notes in Computer Science., Springer (2005) 150–164
7. Bellantoni, S., Cook, S.: A new recursion-theoretic characterization of the polytime functions. Computational Complexity **2** (1992) 97–110
8. Leivant, D., Marion, J.Y.: Lambda calculus characterizations of poly-time. Fundamenta Informaticae **19** (1993) 167,184
9. Leivant, D., Marion, J.Y.: Predicative functional recurrence and poly-space. In Bidoit, M., Dauchet, M., eds.: TAPSOFT'97, Theory and Practice of Software Development. Volume 1214 of Lecture Notes in Computer Science., Springer (1997) 369–380
10. Leivant, D., Marion, J.Y.: A characterization of alternating log time by ramified recurrence. Theoretical Computer Science **236** (2000) 192–208
11. Neergaard, P.: A functional language for logarithmic space. In Springer-Verlag, L., ed.: In Proc. 2nd Asian Symp. on Prog. Lang. and Systems (APLAS 2004). (2004)
12. Hofmann, M.: Linear types and non-size-increasing polynomial time computation. In: Proceedings of the Fourteenth IEEE Symposium on Logic in Computer Science (LICS'99). (1999) 464–473
13. Hofmann, M.: The strength of non size-increasing computation. In Notices, A.S., ed.: POPL'02. Volume 37. (2002) 260 – 269
14. Baillot, P., Terui, K.: Light types for polynomial time computation in lambda-calculus. In Press, I.C.S., ed.: Proceedings. (2004)
15. Oitavem, I.: Characterizing nc with tier 0 pointers. Archive for Mathematical Logic **41** (2002) 35–47
16. Oitavem, I.: A term rewriting characterization of the functions computable in polynomial space. Archive for Mathematical Logic **41** (2002) 35–47
17. Jones, N.D.: Computability and complexity, from a programming perspective. MIT press (1997)
18. Grädel, E., Gurevich, Y.: Tailoring recursion for complexity. Journal of Symbolic Logic **60** (1995) 952–969