Type-based complexity analysis for fork processes

Emmanuel Hainry, Jean-Yves Marion, Romain Péchoux

Type-based complexity analysis for concurrent programs

Emmanuel Hainry, Jean-Yves Marion, and Romain Péchoux

Université de Lorraine and LORIA
{emmanuel.hainry,jean-yves.marion,romain.pechoux}@loria.fr

Abstract. We introduce a type system for concurrent programs described as a parallel imperative language using while-loops and fork/wait instructions, in which processes do not share a global memory, in order to analyze computational complexity. The type system provides an analysis of the data-flow based both on a data ramification principle related to tiering discipline and on secure typed languages. The main result states that well-typed processes characterize exactly the set of functions computable in polynomial space under termination, confluence and lock-freedom assumptions. More precisely, each process computes in polynomial time so that the evaluation of a process may be performed in polynomial time on a parallel model of computation. Type inference of the presented analysis is decidable in linear time provided that basic operator semantics is known.

Keywords: Implicit Computational Complexity, Tiering, Secure Information Flow, Concurrent Programming, PSpace

1 Introduction

We propose a type system for an imperative language with while loops and forks calls, which provides an upper bound on the complexity of a process by controlling its data flow. Threads are created dynamically and they do not share a global memory. Each `fork` call creates a new child process with a distinct identifier (id) and duplicates the execution context including the program counter. The parent process keeps the ids of its children (but not the converse). Communications between children and their parent are performed through the use of a `wait` instruction that allows a returning child to pass a value to its parent process. The domain of computation is, in essence, the set of strings, on which various admissible operators can be defined. Thus, we are able to use arrays of strings with respect to the typing limitation as in Example 1.

We demonstrate that each subprocess generated by a well-typed process (under some restrictions) runs in polynomial time (Proposition 2) and that the number of process offsprings is bounded by a polynomial (Proposition 3). As a result, the amount of interactions between two processes is also bounded by a polynomial. Thanks to Savitch's Theorem [21], we show that a program runs in

polynomial space on a sequential machine (Theorem 2), and conversely (Theorem 3). This result is expressed in the central Theorem 1. As far as we know, it is the first characterization of FPSpace based on a typed system for an imperative language. The type inference procedure is computable in linear time in the program size (Proposition 1). As a result, applications about the complexity measure of process calculi may be considered. We refer to the conclusion for a discussion about practical issues.

That said, one of our main motivation is to understand the relationship between data flow control and computational complexity. In [17], a type system was introduced for a sequential imperative programming language, characterizing the set of polynomial time computable functions. The idea behind these typing systems is to control the information flow in an execution by enforcing a data tiering discipline. The idea is that data are ramified into tiers in such a way that data at a given tier may only produce information at the same tier or at a lower tier. Thus, the type system prevents the alteration of upper tier data by lower tier data, following the principle of an integrity policy [3]. From this observation, there is only a small step to do in order to devise a type system based on works on type-based information flow analysis. We refer to the survey [20] for further explanations. The type system assigns tiers, which are similar to security levels to variables. As in [23, 22], the type system prevents information to flow from lower tier variables to higher tier variables by direct or indirect assignations. From a complexity point of view, the main novelty here is that we have to control the information that flows between processes so that the termination of a father process does not depend on the return values of its children. For this, we suggest a three-tier lattice, $\{-\mathbf{1}, \mathbf{0}, \mathbf{1}\}$, where tier $\mathbf{1}$ variables control the while loops, tier $\mathbf{0}$ variables are working data, and tier $-\mathbf{1}$ variables are output values. Consequently, we prevent an unsafe declassification due to the termination of a process through the wait/return mechanism. Lastly, it is worth noticing that we establish a non-interference property (Proposition 2), which expresses the relationship between information flow and complexity.

There are several works, which are directly related to our results. On function algebra, a characterization of Pspace has been provided by ramified recursion with parameter substitutions [14] and by ramified recurrence over functions [15]. For lambda calculus, [7] introduced a typed lambda calculus based on Soft Linear Logic [11], which identifies exactly Pspace. On functional languages, several characterizations of Pspace have been provided in [8] consisting in restrictions on data manipulation and on recursion schemes. The completeness proofs of the aforementioned works simulate alternating polynomial time Turing machines [5]. Another approach is the one of Pola [6] which relates complexity and category theory. On imperative languages, an approach consists in analyzing the growth of the data flow by means of a matrix calculus propagating constraints between variables [19, 9, 18]. Thus, Niggl and Wunderlich gave a characterization of Pspace [19]. On the complexity of message passing languages based on implicit computational complexity, Amadio and Dabrowski show how to obtain an upper bound on instants for synchronous languages by using quasi-interpretation

based tools [1, 2]. More recently, Dal Lago *et al* proposed to polynomially bound process interactions by type systems based on Light Linear Logic [13, 12]. Notice that the fragment of the process calculi is too weak to establish a completeness result. Lastly, the recent work of Madet [16] proposes a multithreaded programs with side effects also based on light linear logic which are polynomial.

2 Imperative language with forks

2.1 Syntax of processes

Expressions, instructions, commands and processes are defined by the following grammar, where $\mathcal{V}$ is the set of variables and $\mathbb{O}$ is the set of operators. The size of an expression is $|X| = 1$ and $|op(E_1, \ldots, E_n)| = \sum_{i=1}^{n} |E_i| + 1$. We note $\mathcal{V}(K)$, $K \in \text{Exp} \cup \text{Proc}$ the set of variables occurring in K.

$$
\begin{array}{lll}
E, E_1, \ldots, E_n \in \text{Exp} & ::= X \mid op(E_1, \ldots, E_n) & X \in \mathcal{V}, op \in \mathbb{O} \\
I \in \text{Inst} & ::= \texttt{fork}() \mid \texttt{wait}(E) & \\
C, C' \in \text{Cmd} & ::= X{:=}E \mid C;\ C' \mid \texttt{while}\ E\ \texttt{do}\ C \mid & \\
 & \quad \texttt{skip} \mid X{:=}I \mid \texttt{if}\ E\ \texttt{then}\ C\ \texttt{else}\ C' & \\
\text{P} \in \text{Proc} & ::= \texttt{return}\ X \mid C;\ \text{P} &
\end{array}
$$

2.2 Informal semantics

The semantics is similar to the one of `C` programs and `Unix` processes. Each process has an id (a pid). When a fork command is executed, a new child process is created, that will run concurrently with its parent process with its own duplicated memory. The parent process knows the child id, whereas a child has no knowledge of its parent process id. A process P is evaluated inside a *configuration*, a triple $(\text{P}, \mu)_\rho$ consisting of a process, a *store* μ mapping each variable to a value of the computational domain and a set of ids ρ. In a configuration, the process can be viewed as the code that remains to be executed, and ρ stores the children process id's. At the creation of a child process, the store μ and the program counter are duplicated, and ρ is $\emptyset$. The main process id is always 1. When a new child is created by a `fork` its id is automatically set to a new integer and it is recorded in the set ρ of the father's configuration. All the configurations corresponding to the main process and its subprocesses are stored inside an *environment* $\mathscr{E}$, a partial function mapping the process id to a configuration. Processes do not share a global memory. Consequently, the only way for a process to communicate with its children is through the use of a $\texttt{wait}(E)$ instruction, which provides a one-way communication. An example is given in the appendix.

2.3 Semantics of expressions, configurations and environments.

Domain. Let $\mathbb{W}$ be the set of words over a finite alphabet Σ including two symbols `tt` and `ff` that denote truth values true and false. Let ε be the empty

word. The length of a word d is denoted $|d|$. As usual, we set $|\varepsilon| = 0$. Define $\trianglelefteq$ as the sub-word relation over $\mathbb{W}$, by $v \trianglelefteq w$, iff there are u and u' of $\mathbb{W}$ s.t. $w = u.v.u'$, where . is the concatenation. We write $\underline{n}$ to mean the binary word encoding the natural number n.

Store. A *store* μ is a total function from process variables in $\mathcal{V}$ to words in $\mathbb{W}$. Let $\mu\{X_1 \leftarrow d_1, \ldots, X_n \leftarrow d_n\}$, with X_i pairwise distinct, denote the store μ where the value stored by X_i is updated to d_i, for each $i \in \{1, \ldots, n\}$. The size of a store μ, denoted $|\mu|$ is defined by $|\mu| = \sum_{X \in \mathcal{V}} |\mu(X)|$.

Configuration. Given a store μ and a process P, the triplet $c = (\mathrm{P}, \mu)_\rho$, where ρ is an element of $\mathcal{P}(\mathbb{N})$, is called a *configuration*. Let $\perp$ be a special erased configuration. The size of a configuration is defined by $|\perp| = 0$ and $|(\mathrm{P}, \mu)_\rho| = |\mu| + \sharp\rho$, where $\sharp\rho$ is the cardinal of ρ.

Expressions and configurations. Each operator of arity n is interpreted by a total function $[\![op]\!] : \mathbb{W}^n \mapsto \mathbb{W}$. The expression evaluation relation $\xrightarrow{\mathsf{e}}$ and the sequential command evaluation relation are $\xrightarrow{\mathsf{c}}$ are described in Figure 1.

$$(X, \mu) \xrightarrow{\mathsf{e}} \mu(X) \qquad \text{(Variable)}$$

$$(op(E_1, \ldots, E_n), \mu) \xrightarrow{\mathsf{e}} [\![op]\!](d_1, \ldots, d_n), \quad \text{if } \forall i, (E_i, \mu) \xrightarrow{\mathsf{e}} d_i \qquad \text{(Operator)}$$

$$(\texttt{skip}; \mathrm{P}, \mu)_\rho \xrightarrow{\mathsf{c}} (\mathrm{P}, \mu)_\rho \qquad \text{(Skip)}$$

$$(X{:=}E; \mathrm{P}, \mu)_\rho \xrightarrow{\mathsf{c}} (\mathrm{P}, \mu\{X \leftarrow d\})_\rho \quad \text{if } (E, \mu) \xrightarrow{\mathsf{e}} d \qquad \text{(Assign)}$$

$$(\texttt{if } E \texttt{ then } C_{\texttt{tt}} \texttt{ else } C_{\texttt{ff}}; \mathrm{P}, \mu)_\rho \xrightarrow{\mathsf{c}} (C_{\texttt{tt}}; \mathrm{P}, \mu)_\rho \quad \text{if } (E, \mu) \xrightarrow{\mathsf{e}} \texttt{tt} \qquad (\mathrm{If}_{\texttt{tt}})$$

$$(\texttt{if } E \texttt{ then } C_{\texttt{tt}} \texttt{ else } C_{\texttt{ff}}; \mathrm{P}, \mu)_\rho \xrightarrow{\mathsf{c}} (C_{\texttt{ff}}; \mathrm{P}, \mu)_\rho \quad \text{if } (E, \mu) \xrightarrow{\mathsf{e}} \texttt{ff} \qquad (\mathrm{If}_{\texttt{ff}})$$

$$(\texttt{while } E \texttt{ do } C\ ; \mathrm{P}, \mu)_\rho \xrightarrow{\mathsf{c}} (\mathrm{P}, \mu)_\rho \quad \text{if } (E, \mu) \xrightarrow{\mathsf{e}} \texttt{ff} \qquad (\mathrm{While}_{\texttt{ff}})$$

$$(\texttt{while } E \texttt{ do } C\ ; \mathrm{P}, \mu)_\rho \xrightarrow{\mathsf{c}} (C; \texttt{while } E \texttt{ do } C\ ; \mathrm{P}, \mu)_\rho \quad \text{if } (E, \mu) \xrightarrow{\mathsf{e}} \texttt{tt} \qquad (\mathrm{While}_{\texttt{tt}})$$

Fig. 1. Small step semantics of expressions and configurations

Environments. An *environment* $\mathscr{E}$ is a partial function from $\mathbb{N}$ to configurations. The domain of $\mathscr{E}$ is denoted $dom(\mathscr{E})$ and we denote $\sharp\mathscr{E}$ its cardinal when it is finite. We abbreviate $\mathscr{E}(n)$ by $\mathscr{E}_n$. The size of a finite environment $\|\mathscr{E}\|$ is defined by $\|\mathscr{E}\| = \sum_{i \in dom(\mathscr{E})} |\mathscr{E}_i|$. The notation $\mathscr{E}[i := c]$ is the environment $\mathscr{E}'$ defined by $\mathscr{E}'(j) = \mathscr{E}(j)$ for all $j \neq i \in dom(\mathscr{E})$ and $\mathscr{E}'(i) = c$. As usual $\mathscr{E}[i_1 := c_1, \ldots, i_k := c_k]$ is a shortcut for $\mathscr{E}[i_1 := c_1] \ldots [i_k := c_k]$. The *initial* environment is noted $\mathscr{E}_{init}[\mathrm{P}, \mu]$ and consists in the main process with no child. That is $\mathscr{E}_{init}[\mathrm{P}, \mu](1) = (\mathrm{P}, \mu)_\emptyset$ and $dom(\mathscr{E}_{init}[\mathrm{P}, \mu]) = \{1\}$. An environment $\mathscr{E}$ is *terminal* if the root process satisfies $\mathscr{E}_1 = (\texttt{return } X, \mu)_\rho$.

Semantics. The transition $\rightarrow$ for process evaluation is provided in Figure 2. The (Fork) rule creates a new configuration, a new process, say of id n, with a new store, and adds it to the environment. The parent process records its child id $\underline{n}$ into the variable X on which the fork instruction has been called and the child id set of the parent is updated to $\rho \cup \{n\}$. Note also that X is assigned to 0 in the child configuration. The (Wait) commands $Z{:=}\texttt{wait}(E)$ evaluates the expression E to some binary numeral $\underline{n}$. If the process $\underline{n}$ is a terminating configuration $\mathscr{E}_n$ with $n \in \rho$, then the output value $\mu'(Y)$ is transmitted and stored in the variable Z. Finally, the returning process n is killed by erasing it through the following operation $\mathscr{E}[n := \bot]$. Note that the side condition $n \in \rho$ prevents locks.

$$\mathscr{E}[i := c] \rightarrow \mathscr{E}[i := c'] \quad \text{if } c \xrightarrow{\mathsf{c}} c' \tag{Conf}$$

$$\mathscr{E}[i := (X{:=}\texttt{fork}(); \mathrm{P}, \mu)_\rho] \rightarrow \mathscr{E}[i := (\mathrm{P}, \mu\{X \leftarrow \underline{n}\})_{\rho\cup\{n\}}, n := (\mathrm{P}, \mu\{X \leftarrow \underline{0}\})_\emptyset] \text{ with } n = \sharp\mathscr{E} + 1 \tag{Fork}$$

$$\mathscr{E}[i := (X{:=}\texttt{wait}(E); \mathrm{P}, \mu)_\rho] \rightarrow \mathscr{E}[i := (\mathrm{P}, \mu\{X \leftarrow \mu'(Y)\})_\rho, n := \bot] \text{ if } (E, \mu) \xrightarrow{\mathsf{e}} \underline{n},\ n \in \rho \text{ and } \mathscr{E}_n = (\texttt{return } Y, \mu') \tag{Wait}$$

Fig. 2. Semantics of Environments

2.4 Strong normalization, lock-freedom and confluence

Throughout the paper, given a relation $\mapsto$, let $\mapsto^*$ be the reflexive and transitive closure of $\mapsto$ and let $\mapsto^k$ denote the k-fold self composition of $\mapsto$. A process P is *strongly normalizing* if there is no infinite reduction starting from the initial environment $\mathscr{E}_{init}[\mathrm{P}, \mu]$ through the relation $\rightarrow$, for any store μ. Given an initial environment $\mathscr{E}_{init}[\mathrm{P}, \mu]$, for some strongly normalizing process P and some store μ, if $\mathscr{E}_{init}[\mathrm{P}, \mu] \rightarrow^* \mathscr{E}'$, for some environment $\mathscr{E}'$ such that there is no environment $\mathscr{E}''$, $\mathscr{E}' \rightarrow \mathscr{E}''$, then either $\mathscr{E}'$ is terminal, i.e. $\mathscr{E}'_1 = (\texttt{return } X, \mu')_\rho$ (the *main process is returning*) or $\mathscr{E}'_1 = (X{:=}\texttt{wait}(E);\ C', \mu')_\rho$(we say that the environment $\mathscr{E}'$ is locked). A process $\mathrm{P} = C$; $\texttt{return } X$ is *lock-free* if for any initial environment $\mathscr{E}_{init}[\mathrm{P}, \mu]$, there is no locked environment $\mathscr{E}'$ such that $\mathscr{E}_{init}[\mathrm{P}, \mu] \xrightarrow{*} \mathscr{E}'$. A process P is *confluent* if for each initial environment $\mathscr{E}_{init}[\mathrm{P}, \mu]$ and any reductions $\mathscr{E}_{init}[\mathrm{P}, \mu] \rightarrow^* \mathscr{E}'$ and $\mathscr{E}_{init}[\mathrm{P}, \mu] \rightarrow^* \mathscr{E}''$ there exists an environment $\mathscr{E}^3$ such that $\mathscr{E}' \rightarrow^* \mathscr{E}^3$ and $\mathscr{E}'' \rightarrow^* \mathscr{E}^3$. A strongly normalizing, lock free and confluent process P computes a total function $f : \mathbb{W}^n \rightarrow \mathbb{W}$ defined:

$$\forall d_1, \ldots, d_n \in \mathbb{W},\ f(d_1, \ldots, d_n) = w$$

if $\mathscr{E}_{init}[\mathrm{P}, \mu[X_i \leftarrow d_i]] \rightarrow^* \mathscr{E}$, for some terminal environment $\mathscr{E}$ with $\mathscr{E}_1 = (\texttt{return } X, \mu')_\rho$ and $\mu'(X) = w$.

3 Type system

3.1 Tiers and typing environments.

Tiers are elements of the lattice $(\{-\mathbf{1}, \mathbf{0}, \mathbf{1}\}, \vee, \wedge)$ where $\wedge$ and $\vee$ are the greatest lower bound operator and the least upper bound operator, respectively. The induced order, denoted $\preceq$, is such that $-\mathbf{1} \preceq \mathbf{0} \preceq \mathbf{1}$. In what follows, let $\alpha, \beta, \ldots$ denote tiers in $\{-\mathbf{1}, \mathbf{0}, \mathbf{1}\}$. Tiers will be used to type both expressions and commands. Operator types τ are defined by $\tau ::= \alpha \mid \alpha \longrightarrow \tau$. As usual, we will use right associativity for $\longrightarrow$. A *variable typing environment* Γ maps each variable in $\mathcal{V}$ to a tier. An *operator typing environment* Δ maps each operator *op* to a set of operator types $\Delta(op)$, where the operator types corresponding to an operator of arity n are of the shape $\tau = \alpha_1 \longrightarrow \ldots \longrightarrow \alpha_n \longrightarrow \alpha$.

Intuitively, each tier variable has a specific role to play:

- tier $\mathbf{1}$ variables will be used as guards of while loops. They should not be allowed to take more than a polynomial number of distinct values;
- tier $\mathbf{0}$ variables may increase and cannot be used as while loop guards;
- tier $-\mathbf{1}$ variables will store values returned by child processes and cannot increase. Intuitively, they play the role of an output tape.

3.2 Well-typed processes.

Typing rules. Figure 3 provides the typing rules for expressions, commands and processes. Typing rules consist in judgments of the shape $\Gamma, \Delta \vdash K : \alpha$, $K \in \mathrm{Exp} \cup \mathrm{Cmd}$, meaning that K has type α under variable and operator typing environments Γ and Δ, respectively.

There are some important points to explain in this type system. First, the typing discipline precludes values from flowing from tier α to tier β, whenever $\alpha \preceq \beta$. Consequently, the guards of while loops are enforced to be of tier $\mathbf{1}$ in rule (CW). Moreover, in a (CB) rule, we enforce the tier of the guard to be equal to the tier of both branches. Also note that the subtyping rule (CSub) is restricted to commands in order not to break this preclusion. On the opposite, information may flow from tier $\mathbf{1}$ to tier $\mathbf{0}$ then to tier $-\mathbf{1}$. This point is underlined by the side condition of the (CA) rule. The (F) rule enforces the tier of the variable storing the process id to be of tier $\mathbf{0}$ since the value stored will increase dynamically during the process execution. Finally, the tier of the variable storing the result returned by a child process (rule (W)) has to be of tier $-\mathbf{1}$, which means that no information may flow from a variable of a child process to tier $\mathbf{0}$ and tier $\mathbf{1}$ variables of its parent process.

Notations. In practice, we write $C : \alpha$ to say that C is of type α, and E^{α} to says that E is of type α under the considered environments.

Example 1. The following example illustrates how we can program a reduce operation used in parallel prefix sum [10]. The problem consists in computing the greatest element among n integers in an array-like structure $(A[0], \ldots, A[n-1])$. For this, we run n processes which returns $\max_i(A[i])$, when $n = 2^k - 1$ for some k.

$$\frac{\Gamma(X) = \alpha}{\Gamma, \Delta \vdash X : \alpha}\ (EV) \qquad \frac{\Gamma, \Delta \vdash E_i : \alpha_i \qquad \alpha_1 \longrightarrow \ldots \longrightarrow \alpha_n \longrightarrow \alpha \in \Delta(op)}{\Gamma, \Delta \vdash op(E_1, \ldots, E_n) : \alpha}\ (EO)$$

$$\frac{\Gamma, \Delta \vdash X : \mathbf{0}}{\Gamma, \Delta \vdash X{:=}\texttt{fork}() : \mathbf{0}}\ (F) \qquad \frac{\Gamma, \Delta \vdash E : \mathbf{0} \quad \Gamma, \Delta \vdash X : -\mathbf{1}}{\Gamma, \Delta \vdash X{:=}\texttt{wait}(E) : -\mathbf{1}}\ (W)$$

$$\frac{\Gamma, \Delta \vdash X : \alpha \qquad \Gamma, \Delta \vdash E : \alpha' \qquad E \in \mathrm{Exp}}{\Gamma, \Delta \vdash X{:=}E : \alpha}\ \alpha \preceq \alpha' \quad (CA)$$

$$\frac{\Gamma, \Delta \vdash C : \alpha \qquad \Gamma, \Delta \vdash C' : \alpha'}{\Gamma, \Delta \vdash C;\ C' : \alpha \vee \alpha'}\ (CC) \qquad \frac{\Gamma, \Delta \vdash E : \mathbf{1} \qquad \Gamma, \Delta \vdash C : \alpha}{\Gamma, \Delta \vdash \texttt{while}(E)\texttt{do}\{C\} : \mathbf{1}}\ (CW)$$

$$\frac{\Gamma, \Delta \vdash E : \alpha \qquad \Gamma, \Delta \vdash C : \alpha \qquad \Gamma, \Delta \vdash C' : \alpha}{\Gamma, \Delta \vdash \texttt{if}\ E\ \texttt{then}\ C\ \texttt{else}\ C' : \alpha}\ (CB) \qquad \frac{}{\Gamma, \Delta \vdash \texttt{skip} : \alpha}\ (CS)$$

$$\frac{\Gamma, \Delta \vdash C : \alpha}{\Gamma, \Delta \vdash C : \beta}\ \alpha \preceq \beta \quad (CSub) \qquad \frac{\Gamma, \Delta \vdash X : \beta}{\Gamma, \Delta \vdash \texttt{return}\ X : \beta}\ (R)$$

Fig. 3. Type system for expressions, instructions, commands and processes

Throughout, we assume that integers and arrays are implemented by strings. We skip details in order to focus on the control flow structure of programs. We simply use the notation $A[r]$ to express the cell r of the array A. We find convenient to describe now the nature of operators used, that will presented in the next section (Definitions 1 and 2). The get operation on arrays, expressed above by the assignation $f := A[r]$, is a neutral operation of type $\mathbf{0} \longrightarrow \mathbf{0} \longrightarrow \mathbf{0}$ and it can be in *Ntr* because of its type. The length n and the index r are identified by their binary notations. So the operators $2 * r + 2$ and $2 * r + 1$ correspond to appending a letter and are positive operators of type $\mathbf{0}, \mathbf{0} \longrightarrow \mathbf{0}$, and so are in *Pos* . Similarly, the operator $\texttt{half}(n)$ divides the length n by two, that is it deletes a letter. So, it is a neutral operator of type $\mathbf{1} \longrightarrow \mathbf{1}$, and is in *Ntr*. The predicates `or`, $\neq$ are typical neutral operators of types resp. $\mathbf{0} \longrightarrow \mathbf{0} \longrightarrow \mathbf{0}$ and $\mathbf{1} \longrightarrow \mathbf{1} \longrightarrow \mathbf{1}$. But the typing implies that $\texttt{or} \in Ntr$ or $\in Pos$, and $\neq \in Ntr$. Lastly, `max` is a max operator of type $-\mathbf{1} \longrightarrow -\mathbf{1} \longrightarrow -\mathbf{1}$ and is in *Max*. In `max_reduce`, n is the length the the array A of tier $\mathbf{1}$. Notice that the typing of commands uses the subtyping rule *CSub*.

```
max_reduce(n¹,A⁰) ::=
  r⁰ := 0: 0;
  f⁻¹ := A[r]⁰: −1;
  flag⁰ := tt: 0;
  while (n¹ ≠ 1)¹ do {
      if flag⁰ then {          // if not found
          pidl⁰ := fork(): 0
          if (pidl >0)⁰ then { // father process
              r⁰ := 2∗r+2: 0;
```

```
            pidr^0 := fork(): 0}
            else { r^0 := 2*r+1: 0 } // left son
        if (pidr==0)^0 or (pidl==0)^0 then { f^-1 := A[r]^0: 0; }
            else {
                flag^0 := ff: 0; // father
                xl^-1 := wait(pidl): 0;
                xr^-1 := wait(pidr): 0;
                f^-1 := max(f^-1, max(xl,xr)): 0; } }
    n^1 := half(n)^1: 1 } // end of while
return f:-1
```

4 Safe processes, type inference and complexity

4.1 Neutral, max, and positive operators

The type system guarantees that information flow goes from tier $\mathbf{1}$ to tier $-\mathbf{1}$, and prevents any flow in the other way from a lower tier to higher tier. But this is not sufficient to bound process resource. We need fix the class of operator interpretations based on their typing.

Definition 1.

1. *An operator op is* neutral *if:*
   (a) *either op computes a binary predicate (i.e. the codomain of op is* $\{\mathtt{tt}, \mathtt{ff}\}$*).*
   (b) *or* $\forall d_1, \ldots, d_n,\ \exists i \in \{1, \ldots, n\},\ [\![op]\!](d_1, \ldots, d_n) \trianglelefteq d_i$.
2. *An operator op is* max *if* $\forall (d_i)_{1,n}\ |[\![op]\!](d_1, \ldots, d_n)| \leq \max_{i\in[1,n]} |d_i|$.
3. *An operator op is* positive *if* $\forall (d_i)_{1,n},\ |[\![op]\!](d_1, \ldots, d_n)| \leq \max_{i\in[1,n]} |d_i| + c$, *for a constant* c.

Say that a type $\alpha_1 \longrightarrow \ldots \longrightarrow \alpha_n \longrightarrow \alpha$ is decreasing if $\alpha \preceq \wedge_{i=1,n}\alpha_i$. We now give a partition of operators into three classes which depend both on their types and on their growth rates.

Definition 2 (Safe operator typing environment). *An operator typing environment* Δ *is* safe *if each type given by* Δ *is decreasing and there exist three disjoint classes of operators Ntr, Max and Pos such that for any operator op and* $\forall \alpha_1 \longrightarrow \ldots \longrightarrow \alpha_n \longrightarrow \alpha \in \Delta(op)$*, the following conditions hold:*

- *If* $op \in Ntr$ *then op is a neutral operator.*
- *If* $op \in Max$ *then op is a max operator and* $\alpha \neq \mathbf{1}$.
- *If* $op \in Pos$ *then op is a positive operator and* $\alpha = \mathbf{0}$.

Intuitively, expressions in while guards are of tier $\mathbf{1}$ and so the iteration length just depends on the number of possible tier $\mathbf{1}$ configurations. Inside a while loop, we can perform operations on variables of other tiers. The tier $-\mathbf{1}$ values are return values and processes are confined in the sense that the information flow of a process does not depend on a return value.

4.2 Main result

Proposition 1 (Type inference). *Given a safe operator typing environment Δ, deciding if there exists a variable typing environment Γ satisfying the typing rules can be done in time linear in the size of the program.*

Proof. We encode the tier of each variable X by 3 boolean variables $x_{\mathbf{1}}$, $x_{\mathbf{0}}$ and $x_{-\mathbf{1}}$. We enforce that variable X is of exactly one tier will be encoded by $(\neg x_{\mathbf{0}} \vee \neg x_{\mathbf{1}}) \wedge (\neg x_{\mathbf{0}} \vee \neg x_{-\mathbf{1}}) \wedge (\neg x_{\mathbf{1}} \vee \neg x_{-\mathbf{1}})$. Each command gives some constraints. Thus, in the case of an assignment $X := op(Y)$ with $op \in Pos$, we need to encode $\Gamma(Y) \geq \Gamma(X)$, which can be represented as $(\neg y_{-\mathbf{1}} \vee x_{-\mathbf{1}}) \wedge (\neg y_{\mathbf{0}} \vee \neg x_{\mathbf{1}}) \wedge (\neg x_{\mathbf{1}} \vee y_{\mathbf{1}})$. As a result, the type inference problem is reduced to 2-SAT.

Definition 3 (Safe process). *Given Γ a variable typing environment and Δ an operator typing environment, we say that a process* P *is a* safe process *if:*

- P *is well-typed wrt Γ and Δ, i.e. $\Gamma, \Delta \vdash \mathrm{P} : \beta$;*
- *Δ is safe.*

The main result below is a consequence of the soundness Theorem 2 and the completeness Theorem 3

Theorem 1. *The set of polynomial space computable functions is exactly the set of functions computed by strongly normalizing, lock-free, confluent and safe processes, where a unit-cost is taken as the cost of an operator computation.*

Therefore, the process in example 1 can run within a polynomial space. It is worth noticing that the demonstration below says more about process interaction and runtime, which are, in fact, polynomially bounded as we shall see.

5 Complexity soundness

5.1 Process runtime

We establish that each process runs in polynomial time if the time measure is the number of reductions. We follow the line of the soundness proof of [23] to demonstrate a noninterference property. We first prove two lemmas 1 and 2 in order to express that an expression of tier **1** just depends on tier **1** variables, and not on lower tier variables. As a result, there is only a polynomial number of tier **1** configurations, because of neutral operator growth rate. From the security point of view, this means that a variable of tier (level) **1** can be updated by information of the same tier of integrity. Proposition 2 corresponds to a non-interference property. Intuitively, it says that the complexity of a process depends only on tier **1** variables, and so it is not modified if values of lower tiers are updated and so the process runtime do not interfere with lower tier values.

Lemma 1 (Simple security). *Given a safe process* P *wrt typing environments Γ and Δ, if $\Gamma, \Delta \vdash E : \mathbf{1}$, for an expression E in* P*, then for each $X \in \mathcal{V}(E)$, $\Gamma(X) = \mathbf{1}$ and all operators in E are neutral.*

Proof. By induction on E.

Given a typing environment Γ say that $|\mu|_{\mathbf{i}} = \sum_{\Gamma(X)=\mathbf{i}} |\mu(X)|$.

Lemma 2. *Given a safe process* P *wrt typing environments Γ and Δ, for each expression E in* P *such that $\Gamma, \Delta \vdash E : \mathbf{1}$, the number of distinct values taken by E during the evaluation of the initial environment $\mathscr{E}_{init}[\mathrm{P}, \mu]$ is bounded polynomially in $|\mu|_{\mathbf{1}}$.*

Proof. A store can be updated either (i) by using a `fork` instruction, (ii) by using a `wait` instruction or (iii) by assigning the result of an expression. The typing discipline prevents variables assigned to in (i) and (ii) to be of tier **1**. For (iii), we have $E = op(E_1, \ldots, E_n)$, for some neutral operator op, by Lemma 1. Consequently, the result of the computation is either `tt` or `ff` or a subword of the initial values. The number of sub-words being quadratic in the size, the number of distinct values of tier **1** variables is at most quadratic in $|\mu|_{\mathbf{1}}$.

The relations $\Rightarrow_i$ are defined to express the computation of the process of id i.

Definition 4 ($\Rightarrow_i$). *Given two environments $\mathscr{E}$ and $\mathscr{E}'$ such that $\mathscr{E} \to \mathscr{E}'$, we write $\mathscr{E} \to_i \mathscr{E}'$ if the transition $\to$ corresponds to the evaluation of the process in the configuration $\mathscr{E}_i$ of $\mathscr{E}$. In the same way define the transition $\mathscr{E} \to_{\neq i} \mathscr{E}'$ if there exists $k \neq i$ such that $\mathscr{E} \to_k \mathscr{E}'$. Now define $\Rightarrow_i$ by $\mathscr{E} \Rightarrow_i \mathscr{E}'$ if there are environments $\mathscr{E}^1, \mathscr{E}^2$ such that $\mathscr{E} \to^*_{\neq i} \mathscr{E}^1 \to_i \mathscr{E}^2 \to^*_{\neq i} \mathscr{E}'$.*

The proposition below expresses that the number of instructions performed by a single process is bounded by a polynomial in the tier **1** initial values. Intuitively, this means that each process runs in polynomial time, in the tier **1** initial values, if we do not count the waiting time due to forks.

Proposition 2 (Non-interference wrt time). *Given a strongly normalizing and safe process* P*, there is a polynomial Q s.t., for each initial environment $\mathscr{E}_{init}[\mathrm{P}, \mu]$, $\forall i \in \mathbb{N}$, if $\mathscr{E}_{init}[\mathrm{P}, \mu] \Rightarrow_i^k \mathscr{E}$ then $k \leq Q(|\mu|_{\mathbf{1}})$.*

Proof. The typing discipline enforces all the expressions in while-loop guards of a safe process to be of tier **1**. The evaluation of tier **1** values do not depend on lower tier values By Lemma 2, the number of values taken by tier **1** variables during the evaluation is bounded by $Q(|\mu|_{\mathbf{1}})$ for some polynomial Q. If a process enters twice in the same configuration with the same tier **1** values, then the computation loops forever. Consequently, if P is strongly normalizing then all the while loops are executed at most $Q(|\mu|_{\mathbf{1}})$ times.

5.2 Process spawning

Now, we demonstrate that given a strongly normalizing and safe process, the number of children of a process (Lemma 3), the number of process genrations (Lemma 4) and the size of a configuration (Lemma 5) are polynomially bounded.

Definition 5. *Given a finite environment $\mathscr{E}$, the* process tree $T(\mathscr{E})$ *is defined:*

- *the nodes are the configurations* $\{\mathscr{E}_1, \ldots, \mathscr{E}_{\sharp\mathscr{E}}\}$ *and the root is* $\mathscr{E}_1$*;*
- *for each* $l \in [1, \sharp\mathscr{E}]$*, there is an edge from* $\mathscr{E}_l = (\mathrm{P}, \mu)_\rho$ *to* $\mathscr{E}_k$*, if* $k \in \rho$*.*

The degree $d(T)$ corresponds to the number of children generated by fork instructions of a given process. The height $h(T)$ is the number of nested processes.

Lemma 3. *Given a strongly normalizing and safe process* P*, there exists a polynomial* Q *such that, for each initial environment* $\mathscr{E}_{init}[\mathrm{P}, \mu]$*, if* $\mathscr{E}_{init}[\mathrm{P}, \mu] \to^* \mathscr{E}$ *then* $d(T(\mathscr{E})) \leq Q(|\mu|_{\mathbf{1}})$*. In other words, for each* $\mathscr{E}_i = (\mathrm{P}_i, \mu_i)_{\rho_i}$*,* $i \leq \sharp\mathscr{E}$*, the number of subprocesses is bounded by* $Q(|\mu|_{\mathbf{1}})$*, i.e.* $\sharp\rho_i \leq Q(|\mu|_{\mathbf{1}})$*.*

Proof. By Proposition 2, there is a polynomial Q such that the transition $\to_i$ is taken at most $Q(|\mu|_{\mathbf{1}})$ times, which bounds the number of executed `fork`().

Lemma 4. *Given a strongly normalizing and safe process* P*, there exists a polynomial* Q *such that, for each initial environment* $\mathscr{E}_{init}[\mathrm{P}, \mu]$*, if* $\mathscr{E}_{init}[\mathrm{P}, \mu] \to^* \mathscr{E}$ *then* $h(T(\mathscr{E})) \leq Q(|\mu|_{\mathbf{1}})$*.*

Proof. By Proposition 2, there is a polynomial Q which bounds the number of executed instructions and which just depends on the size of tier **1** initial values. Now, when $X :=$ `fork`() is performed, the parent process store is duplicated in the child process store, except for the value of X. But X is of tier 0 and so has no impact on the computational time of both processes. Next, a fork command has been executed in the parent process. So, the runtime of the child generated is strictly less than the runtime of its father. Therefore, we deduce that Q bounds the height of the process tree.

We end by showing that subprocess stores are polynomially bounded in the size of the initial store, which is a consequence of the non-interference property, as stated in Proposition 2.

Lemma 5. *Given a strongly normalizing and safe process* P*, there exists a polynomial* S *such that, for each initial environment* $\mathscr{E}_{init}[\mathrm{P}, \mu]$*, if* $\mathscr{E}_{init}[\mathrm{P}, \mu] \to^* \mathscr{E}$ *then* $\forall i \leq \sharp\mathscr{E}$*, if* $\mathscr{E}_i = (\mathrm{P}_i, \mu_i)_{\rho_i}$ *then* $|\mu_i| \leq S(|\mu|_{\mathbf{1}}) + |\mu|_{\mathbf{0}} + |\mu|_{-\mathbf{1}}$*.*

Proof. There are three cases to consider. First, a variable of tier **1** is updated by an expression of tier **1**. By lemma 1, a tier **1** expression just consists in neutral operators and tier **1** variables. So, tier **1** variables are always sub-words of tier **1** initial values. Thus, the size of a variable of tier **1** is always bounded by $|\mu|_{\mathbf{1}}$. Second, take a tier **0** variable X, which is either updated inside a process by a composition of operators or updated by a pid return of a fork. By combining proposition 2 and lemmas 3 and 4, we obtain a polynomial R which depends on operators and on the program such that the size of X is bounded by $R(Q(|\mu|_{\mathbf{1}})) + |\mu|_{\mathbf{0}}$ in each process computation. Third, take a tier $-\mathbf{1}$ variable Y. Suppose that the variable Y is assigned to max operators and variables of any tier. Observe that the case when $Y :=$ `wait`(E') is a particular case of an assignation if we see globally all processes together. Hence Y is bounded by $R(Q(|\mu|_{\mathbf{1}}) + |\mu|_{\mathbf{0}} + |\mu|_{-\mathbf{1}}$.

5.3 PSpace abiding evaluation strategy

We define a deterministic evaluation strategy $\twoheadrightarrow$ starting from the initial environment and evaluating it until it reaches a wait instruction for a process n. Then, the strategy runs the process n until it returns a value. Formally, define a state $s = (\mathscr{E}, l)$ to be a pair of an environment $\mathscr{E}$ and a stack l of ids representing the queued processes. The initial state is $(\mathscr{E}_{init}[\mathrm{P}, \mu], [1])$. We denote :: the stack constructor.

1. If $\mathscr{E} \rightarrow_h \mathscr{E}'$ for some rule (R) of Figure 2, $R \neq Wait$, then $(\mathscr{E}, h :: q) \twoheadrightarrow (\mathscr{E}', h :: q)$
2. If $\mathscr{E}_h = (X := \texttt{wait}(E); \mathrm{P}, \mu)_\rho$ and $(E, \mu) \xrightarrow{\mathsf{e}} \underline{n}$ then $(\mathscr{E}, h :: q) \twoheadrightarrow (\mathscr{E}, n :: h :: q)$
3. If $\mathscr{E}_n = (\texttt{return}\ X, \mu)_\rho$ and $\mathscr{E}_h = (Y{:=}\texttt{wait}(E); \mathrm{P}, \mu')_{\rho'}$ then $(\mathscr{E}, n :: h :: q) \twoheadrightarrow (\mathscr{E}', h :: q)$ where $\mathscr{E} \rightarrow_h \mathscr{E}'$ for the (Wait) rule of Figure 2.

Notice that the rule (2) implies that $(E, \mu') \xrightarrow{\mathsf{e}} \underline{n}$ in rule (3).

Lemma 6 (Correction Lemma). *Given a strongly normalizing and confluent process* P *and an initial environment* $\mathscr{E}_{init}[\mathrm{P}, \mu]$*, if there exists* $\mathscr{E}$ *such that* $\mathscr{E}_{init}[\mathrm{P}, \mu] \rightarrow^* \mathscr{E}[1 := (\texttt{return}\ X, \mu')_\rho]$ *then there exists* $\mathscr{E}'$ *s.t.* $(\mathscr{E}_{init}[\mathrm{P}, \mu], [1]) \twoheadrightarrow^* (\mathscr{E}'[1 := (\texttt{return}\ X, \mu'')_{\rho'}], [1])$ *and* $\mu''(X) = \mu'(X)$*, where* $[1]$ *is the stack containing the single process* 1*.*

The size of a stack $|l|$ is the number of elements in it (e.g. $|[1]| = 1$).

Lemma 7. *Given a strongly normalizing and safe process* P*, there is a polynomial* Q *such that for each initial environment* $\mathscr{E}_{init}[\mathrm{P}, \mu]$ *and each stack* l*, if* $(\mathscr{E}_{init}[\mathrm{P}, \mu], [1]) \twoheadrightarrow^* (\mathscr{E}, l)$ *then (i)* $|l| \leq Q(|\mu|)$ *and (ii)* $\|\mathscr{E}\| \leq Q(|\mu|)$*.*

Proof. The $\twoheadrightarrow$ strategy explores the process tree using a stack. The height of the stack corresponds to the number of nested processes. The first inequality follows Lemma 4. Next, by combining Lemmata 3 and 5, we obtain the second inequality.

Theorem 2 (Soundness). *A strongly normalizing, confluent and safe process* P *can be evaluated in polynomially bounded space using strategy* $\twoheadrightarrow$*, where a unit-cost is taken as the cost of an operator computation.*

6 Completeness

Let us show that each polynomial space computable function can be computed by a strongly normalizing and safe process.

Theorem 3. *Every polynomial space function is computable by a strongly normalizing, lock-free and safe process* P*.*

Proof. We present how to compute the value of a quantified boolean formula (QBF). The program below may be seen as a skeleton from which we may simulate, for example, an alternating Turing machine running in polynomial time. Consequently, the computation of a polynomial space function necessitates to compute its output bit by bit, which may be uniformly performed by generating each address and querying the output bit. The program below generates 2^n forks where n is number of variables. We suppose that the concrete syntax of a QBF is implemented as a string `phi` of tier $\mathbf{1}$. Two neutral operators `kind` and `variable` of type $\mathbf{1} \longrightarrow \mathbf{1}$ return the quantifier kind and the variable bound at the root of `phi`. The neutral operator `next` of type $\mathbf{1} \longrightarrow \mathbf{1}$ erases the root quantifier and moves to the next one. The positive operator `evaluate` of type $\mathbf{1}, \mathbf{0} \longrightarrow \mathbf{0}$ computes a boolean formula with respect to an evaluation encoded as an array of booleans, which is a NC^1 complete problem [4]. As in example 1, arrays are words of type $\mathbf{0}$, whose length is bounded by `phi`. The set array operator, that we conveniently note `vartab[x] := tt`, is a positive operator of type $\mathbf{0}, \mathbf{0} \longrightarrow \mathbf{0}$. We consider the array to be pre-allocated using a `calloc` operator that can be implemented using a positive operator in a loop.

```
qbf(phi) ::=
    psi¹ := phi¹ : 1;
    q¹ := kind(phi¹) : 1;
    x¹ := variable(phi¹) : 1;
    i⁰ := 0⁰ : 0; // son number of a process
    pidtab⁰ := calloc(phi¹, phi¹) : 0;
    vartab⁰ := calloc(phi¹, tt) : 0;
    while (q¹ == '∃') or (q¹ == '∀') do {
        phi¹ := next(phi¹) : 1;
        pid⁰ := fork() : 0;
        if pid⁰>0 then {                          // father process
            pidtab[i] := (pid,q) : 0;
            vartab[x] := tt : 0;
            i⁰ := i + 1⁰ :0
        } else {                                  // son process
            vartab[x] := ff : 0;
            i⁰ := 0⁰ : 0} // end of else
        q¹ := kind(phi¹) : 1;
        x¹ := variable(phi¹) : 1;} // end of while
    res⁻¹ := evaluate(phi¹, vartab)⁰ : 0;
    while(state(psi¹) == '∃') or (state(psi¹) == '∀') do {
        psi¹ := next(psi¹) : 1;
        if (i>0) then {// for each son
            i⁰ := i - 1⁰ : 0;
            (pid⁰,op⁰) := pidtab[i]: 0;
            res_son⁻¹ := wait(pid⁰)⁻¹ : -1
            if op⁰ == '∃' then
                res⁻¹ := or(res, res_son)⁻¹ :0;
            else
                res⁻¹ := and(res, res_son)⁻¹ :0; } }
    return res⁻¹
```

7 Conclusion

We established that a typed process will generate only a polynomial number of offspring processes and that each process runs within polynomial time bounds. This work may have practical applications to determine whether or not a process runs within certain limited computing resources. Indeed, it may be a tool to control the spawning mechanism, and so to prevent, for example, denial of service attacks.

Let us come back to Example 1 describing a max-reduce algorithm. The runtime is $O(log^2(n))$. The reason of this apparent shortcoming is that we can not allow a break instruction inside a while-loop. In this case, a return instruction would "disrupt" the information flow and would interfere with tier **1** values. As a result, the runtime would not be anymore bounded. However, if the goal is to analyze program complexity, then we can devise a program transformation Θ, which, putting things quickly, moves break instructions outside while loops, whenever it is possible. Such program transformation may be efficiently performed by a tree transducer. Moreover, there is no reason to preserve the program semantics because we are just interested by resource bounds. So, we may omit implementation details, which may increase drastically the expressivity of the considered programs. Therefore, we may think of this overall scenario: A system receives a process P to run. First, it computes an abstraction of P using a program transformation Θ. Second, it checks that the abstraction is well-typed. As a result, and even if the system do not know precisely an upper-bound on the complexity, it gets efficiently some confidence on the resource usage of P.

The program solving QBF gives another limitation because we are not allowed to make a loop bounded by the number of subprocesses, which is a tier **0** value. However, we know that this number is bounded by a tier **1** value. To solve this, Hofmann suggested to manage the garbage collector thanks to a type system. We may think to allow iterations on controlled unsafe values by a kind of declassification rule.

The above questions suggest the need to delve deeper into the question of determining the amount of information, which can be declassified while guarantying complexity. In regards to future research, the relationship between information, information flow control and complexity should be better understood.

References

1. R. M. Amadio and F. Dabrowski. Feasible reactivity for synchronous cooperative threads. *Electron. Notes Theor. Comput. Sci.*, 154(3):33–43, 2006.
2. R. M. Amadio and F. Dabrowski. Feasible reactivity in a synchronous pi-calculus. In *Proceedings of the 9th ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '07, pages 221–230, New York, NY, USA, 2007. ACM.
3. K. Biba. Integrity considerations for secure computer systems. Technical report, Mitre corp Rep., 1977.

4. S. R. Buss. The boolean formula value problem is in alogtime. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, STOC '87, pages 123–131, New York, NY, USA, 1987. ACM.
5. A. Chandra, D. Kozen, and L. Stockmeyer. Alternation. *J. ACM*, 28(1):114–133, 1981.
6. R. Cockett and B. Redmond. A categorical setting for lower complexity. *Electron. Notes Theor. Comput. Sci.*, 265:277–300, 2010.
7. M. Gaboardi, J.-Y. Marion, and S. Ronchi Della Rocca. A logical account of pspace. In *POPL '08*, pages 121–131. ACM, 2008.
8. N. D. Jones. The expressive power of higher-order types or, life without cons. *J. Funct. Program.*, 11(1):5–94, 2001.
9. N. D. Jones and L. Kristiansen. A flow calculus of *wp*-bounds for complexity analysis. *ACM Trans. Comput. Log.*, 10(4), 2009.
10. Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *J. ACM*, 27(4):831–838, October 1980.
11. Y. Lafont. Soft linear logic and polynomial time. *Theor. Comput. Sci.*, 318(1-2):163–180, 2004.
12. U. Dal Lago and P. Di Giamberardino. Soft session types. In *EXPRESS 2011*, volume 64 of *EPTCS*, pages 59–73, 2011.
13. U. Dal Lago, S. Martini, and D. Sangiorgi. Light logics and higher-order processes. In *EXPRESS'10*, volume 41 of *EPTCS*, pages 46–60, 2010.
14. D. Leivant and J.-Y. Marion. Ramified recurrence and computational complexity II: Substitution and poly-space. In *CSL*, volume 933 of *LNCS*, pages 486–500, 1994.
15. D. Leivant and J.-Y. Marion. Predicative functional recurrence and poly-space. In *TAPSOFT'97*, volume 1214 of *LNCS*, pages 369–380. Springer, 1997.
16. A. Madet. A polynomial time lambda-calculus with multithreading and side effects. In *PPDP*, 2012.
17. J.-Y. Marion. A type system for complexity flow analysis. In *LICS*, pages 123–132, 2011.
18. J.-Y. Moyen. Resource control graphs. *ACM Trans. Comput. Logic*, 10(4):29:1–29:44, 2009.
19. K.-H. Niggl and H. Wunderlich. Certifying polynomial time and linear/polynomial space for imperative programs. *SIAM J. Comput.*, 35(5):1122–1147, 2006.
20. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, 2003.
21. W. J. Savitch. Relationship between nondeterministic and deterministic tape classes. *JCSS*, 4:177–192, 1970.
22. G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *POPL*, pages 355–364. ACM, 1998.
23. D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2/3):167–188, 1996.

A Appendix

An example to illustrate the fork/wait/return mechnism

We illustrate how to define distinct code that will be executed by the child but not the parent process and conversely. For the parent, X contains the pid of its child, for the child, X contains 0.

```
P:    X := fork();                    // X contains the child pid
      if X > 0 then {      // father's code (X>0)
            Y := wait(X);
            Y := "father"
      } else {                        // child"s code (X=0)
            Y := "child"
      }
      return Y   // The father returns "father" and the child returns "child"
```