Complexity Information Flow in a Multi-threaded Imperative Language

Jean-Yves Marion, Romain Péchoux

Complexity Information Flow in a Multi-threaded Imperative Language

Jean-Yves Marion and Romain Péchoux

Université de Lorraine, CNRS and INRIA
LORIA
jean-yves.marion@loria.fr,romain.pechoux@loria.fr

Abstract. We propose a type system to analyze the time consumed by multi-threaded imperative programs with a shared global memory, which delineates a class of safe multi-threaded programs. We demonstrate that a safe multi-threaded program runs in polynomial time if (i) it is strongly terminating wrt a non-deterministic scheduling policy or (ii) it terminates wrt a deterministic and quiet scheduling policy. As a consequence, we also characterize the set of polynomial time functions. The type system presented is based on the fundamental notion of data tiering, which is central in implicit computational complexity. It regulates the information flow in a computation. This aspect is interesting in that the type system bears a resemblance to typed based information flow analysis and notions of non-interference. As far as we know, this is the first characterization by a type system of polynomial time multi-threaded programs.

1 Introduction

The objective of this paper is to study the notion of complexity flow analysis introduced in [18] in the setting of concurrency. Our model of concurrency is a simple multi-threaded imperative programming language where threads communicate through global shared variables. The measure of time complexity that we consider for multi-threaded programs is the processing time. That is the total time for all threads to complete their tasks. As a result, the time measure gives an upper bound on the number of scheduling rounds. The first contribution of this paper is a novel type system, which guarantees that each strongly terminating safe multi-threaded program runs in polynomial time (See Section 3.2 and Theorem 6). Moreover, the runtime upper bound holds for all thread interactions. As a simple example, consider the two-thread program:

$$\begin{array}{ll} x: \texttt{while}(X^{\mathbf{1}} == Y^{\mathbf{1}})\{\texttt{skip}\} & y: \texttt{while}(X^{\mathbf{1}} \neq Y^{\mathbf{1}})\{\texttt{skip}\} \\ \quad C; & \quad C'; \\ \quad X^{\mathbf{1}} := \neg X^{\mathbf{1}} & \quad Y^{\mathbf{1}} := \neg Y^{\mathbf{1}} \end{array}$$

This example illustrates a simple synchronization protocol between two threads x and y. Commands C and C' are critical sections, which are assumed not to modify X and Y. The operator $\neg$ denotes the boolean negation. Both threads

are safe if commands C and C' are safe with respect to the same typing environment. Our first result states that this two-thread program runs in polynomial time (in the size of the initial shared variable values) if it is strongly terminating and safe.

Then, we consider a class of deterministic schedulers, that we call quiet (see Section 8). The class of deterministic and quiet schedulers contains all deterministic scheduling policies which depend only on threads. A typical example is a round-robin scheduler. The last contribution of this paper is that a safe multi-threaded program which is terminating wrt to a deterministic and quiet scheduler, runs in polynomial time. Despite the fact it is not strongly terminating, the two-thread program below terminates under a round-robin scheduler, if C and C' terminate.

$$\begin{array}{ll} x: & \texttt{while}(X^{\mathbf{1}} > 0) \\ & \quad \{C; \\ & \quad Z^{\mathbf{1}} := 0 : \mathbf{1}\} : \mathbf{1} \end{array} \qquad \begin{array}{ll} y: & \texttt{while}(Z^{\mathbf{1}} > 0) \\ & \quad \{C'; \\ & \quad X^{\mathbf{1}} := 0 : \mathbf{1}\} : \mathbf{1} \end{array}$$

If commands C and C' are safe, then this two-thread program runs in polynomial time wrt to a round-robin scheduler. The last contribution is that if we just consider one-thread programs, then we characterize exactly FPtime, which is the class of polynomial time functions. (See Theorem 7)

The first rational behind our type system comes from data-ramification concept of Bellantoni and Cook [5] and Leivant [16]. The type system has two atomic types **0** and **1** that we called tiers. The type system precludes that values flow from tier **0** to tier **1** variables. Therefore, it prevents circular algorithmic definitions, which may possibly lead to an exponential length computation. More precisely, explicit flow from **0** to **1** is forbidden by requiring that the type level of the assigned variable is less or equal than the type level of the source expression. Implicit flow is prevented by requiring that (i) branches of a conditional are of the same type and (ii) guard and body of while loops are of tier **1**. If we compare with data-ramification concept of [5,16], tier **1** parameters correspond to variables on which a ramified recursion is performed whereas tier **0** parameters correspond to variables on which recursion is forbidden.

The second rational behind our type system comes from secure flow analysis. See Sabelfeld and Myers survey [21] to have an overview on information flow analysis. In [23] for sequential imperative programs and in [22] for multi-threaded imperative programming language, Irvine, Smith and Volpano give a type system to certify a confidentiality policy. Types are based on security levels say H (High) and L (Low). The type system prevents that there is no leak of information from level H to level L, which is similar to our type system: **0** (resp. **1**) corresponds to H (resp. L). In fact, our approach rather coincides with an integrity policy [6] (i.e "no read down" rule) than with a confidentiality one [4]. A key property is the non-interference, which says that values of level L don't changed values of level H. We demonstrate a similar non-interference result which states that values stored in tier **1** variables are independent from tier **0** variables. See Section 4 for a precise statement. From this, we demonstrate a temporal non-interference properties which expresses that the number

of unfolded (i.e. the length) while loops only depends on tier **1** variables, see Section 5. The temporal non-interference property is the crucial point to establish complexity bounds.

From a practical standpoint, an important issue is the expressivity of the class of safe multi-threaded programs. With this work and [18], we introduce a new approach in implicit computational complexity based on a type system. This study focuses on the intrinsic mechanisms which lead to analyze computational complexity. This approach seems promising because it treats common algorithmic control structures like while-loops as well as sequential and parallel composition. Several examples are presented in Appendix.

Related works. An important source of inspiration comes from Implicit Computational Complexity (ICC). Beside the works of Bellantoni, Cook and Leivant already cited, there are works on light logics [10,3], on linear types [11], and interpretation methods [7,19], just to mention a few. There are also works on resource control of imperative language like [12,13,20]. Only a few studies based on ICC methods are related to resource control of concurrent computational models. In [2], a bound on the resource needed by synchronous cooperative threads in a functional framework is computed. The paper [1] provides a static analysis for ensuring feasible reactivity in a synchronous π-calculus. In [17] an elementary affine logic is introduced to tame the complexity of a modal call-by-value lambda calculus with multi-threading and side effects. There are also works on the termination of multi-threaded imperative languages [9]. In this paper, we separate complexity analysis from termination analysis but the tools on termination can be combined with our results since most of them require strong normalization of the considered process as an assumption. Finally our type system in this paper may be seen as a simplification of the type system of [18] for imperative language but in return there is no declassification mechanism.

2 A complexity flow type system

2.1 A multi-threaded programming language

We introduce a multi-threaded imperative programming language similar to the language of [22,8] and which is an extension of the simple while-imperative programming language of [14]. A multi-threaded program consists in a finite set of threads where each thread is a while-program. Threads run concurrently on a common shared memory. A thread interacts with other threads by reading and writing on the shared memory.

Commands and expressions are built from a set $\mathbb{V}$ of variables, and a set $\mathbb{O}$ of operators of fixed arity including constants (operators of arity 0) as follows:

$$\begin{array}{lll} \textit{Expressions} & E_1, \ldots, E_n ::= X \mid op(E_1, \ldots, E_n) & X \in \mathbb{V}, op \in \mathbb{O} \\ \textit{Commands} & C, C' ::= X{:=}E \mid C \,;\, C' \mid \texttt{skip} \mid \texttt{if } E \texttt{ then } C \texttt{ else } C' & \\ & \quad \mid \texttt{while}(E)\{C\} & \end{array}$$

A multi-threaded program M (or just program when there is no ambiguity) is a finite map from thread identifiers $x, y, \ldots$ to commands. We write $dom(M)$

to denote the set of thread identifiers. Note also that we do not consider the ability of generating new threads. Let $\mathcal{V}(I)$ be the set of variables occurring in I, where I is an expression, a command or a multi-threaded program.

2.2 Semantics

We give a standard small step operational semantics for multi-threaded programs. Let $\mathbb{W}$ be the set of words[1] over a finite alphabet Σ including two words `tt` and `ff` that denote true and false. The length of a word d is denoted $|d|$. A store μ is a finite mapping from $\mathbb{V}$ to $\mathbb{W}$. We write $\mu[X_1 \leftarrow d_1, \ldots, X_n \leftarrow d_n]$ to mean the store μ' where X_i is updated to d_i.

The evaluation rules for expressions and commands are given in Figure 1. Each operator of arity n is interpreted by a total function $[\![op]\!] : \mathbb{W}^n \mapsto \mathbb{W}$. The judgment $\mu \vDash E \xrightarrow{\mathtt{e}} d$ means that the expression E is evaluated to the word $d \in \mathbb{W}$ wrt μ. A configuration c is either a pair of store and command, $\mu \vDash C$, or a store μ. The judgment $\mu \vDash C \xrightarrow{\mathtt{s}} \mu'$ expresses that C terminates and outputs the store μ'. $\mu \vDash C \xrightarrow{\mathtt{s}} \mu' \vDash C'$ means that the evaluation of C is still in progress: the command has evolved to C' and the store has been updated to μ'.

For a multi-threaded program M, the store μ plays the role of a global memory shared by all threads. The store μ is the only way for threads to communicate. The definition of the global relation $\xrightarrow{\mathtt{g}}$ is given in Figure 1, where $M - x$ is the restriction of M to $dom(M) - \{x\}$ and $M[x := C_1]$ is the map M where the command assigned to x is updated to C_1. At each step, a thread x is chosen non-deterministically. Then, one step of x is performed and the control returns to the upper level. Note that the rule *(Stop)* halts the computation of a thread. In what follow, let $\emptyset$ be a notation for the (empty) multi-threaded program (i.e. all threads have terminated). We will discuss of deterministic scheduling policy in the last section.

A multi-threaded program M is strongly terminating, noted $M\Downarrow$, if for any store, all reduction sequences starting from M are finite. Let $\xrightarrow{\mathtt{h}}^k$ be the k-fold self composition and $\xrightarrow{\mathtt{h}}^*$ be the reflexive and transitive closure of the relation $\xrightarrow{\mathtt{h}}$, $\mathtt{h} \in \{\mathtt{s},\mathtt{g}\}$. The running time of a strongly terminating program M is the function $Time_M$ from $\mathbb{W}^n$ to $\mathbb{N}$ defined by:

$$Time_M(d_1, \ldots, d_n) = \max\{k \mid \mu_0[X_1 \leftarrow d_1, \ldots, X_n \leftarrow d_n] \vDash M \xrightarrow{\mathtt{g}}^k \mu \vDash \emptyset\}$$

where μ_0 is the empty store that maps each variable to the empty word $\epsilon \in \mathbb{W}$. A strongly terminating multi-threaded program M is running in polynomial time if there is a polynomial Q such that for all $d_1, \ldots, d_n \in \mathbb{W}$, $Time_M(d_1, \ldots, d_n) \leq Q(\max_{i=1,n} |d_i|)$. Observe that, in the above definition, the time consumption of an operator is considered as constant, which is fair if operators are supposed to be computable in polynomial time.

[1] Our result could be generalized to other domains such as binary trees or lists. However we have restricted this study to words in order to lighten our presentation.

$$\frac{}{\mu \vDash X \xrightarrow{e} \mu(X)} \qquad \frac{\mu \vDash E_1 \xrightarrow{e} d_1 \quad \dots \quad \mu \vDash E_n \xrightarrow{e} d_n}{\mu \vDash op(E_1, \dots, E_n) \xrightarrow{e} [\![op]\!](d_1, \dots, d_n)}$$

$$\frac{}{\mu \vDash \texttt{skip} \xrightarrow{s} \mu} \qquad \frac{\mu \vDash E \xrightarrow{e} d}{\mu \vDash X{:=}E \xrightarrow{s} \mu[X \leftarrow d]} \qquad \frac{\mu \vDash C_1 \xrightarrow{s} \mu_1}{\mu \vDash C_1 \ ; \ C_2 \xrightarrow{s} \mu_1 \vDash C_2}$$

$$\frac{\mu \vDash C_1 \xrightarrow{s} \mu_1 \vDash C_1'}{\mu \vDash C_1 \ ; \ C_2 \xrightarrow{s} \mu_1 \vDash C_1'; C_2} \qquad \frac{\mu \vDash E \xrightarrow{e} w, \ w \in \{\texttt{tt}, \texttt{ff}\}}{\mu \vDash \texttt{if } E \texttt{ then } C_{\texttt{tt}} \texttt{ else } C_{\texttt{ff}} \xrightarrow{s} \mu \vDash C_w}$$

$$\frac{\mu \vDash E \xrightarrow{e} \texttt{ff}}{\mu \vDash \texttt{while}(E)\{C\} \xrightarrow{s} \mu} \qquad \frac{\mu \vDash E \xrightarrow{e} \texttt{tt}}{\mu \vDash \texttt{while}(E)\{C\} \xrightarrow{s} \mu \vDash C; \texttt{while}(E)\{C\}} (W_{\texttt{tt}})$$

$$\frac{M(x) = C \quad \mu \vDash C \xrightarrow{s} \mu_1}{\mu \vDash M \xrightarrow{g} \mu_1 \vDash M - x} \textit{(Stop)} \qquad \frac{M(x) = C \quad \mu \vDash C \xrightarrow{s} \mu_1 \vDash C_1}{\mu \vDash M \xrightarrow{g} \mu_1 \vDash M[x := C_1]} \textit{(Step)}$$

Fig. 1. Small step semantics of expressions, commands and multi-threads

2.3 Type system

Atomic types are elements of the boolean lattice $(\{\mathbf{0}, \mathbf{1}\}, \preceq, \mathbf{0}, \vee, \wedge)$ where $\mathbf{0} \preceq \mathbf{1}$. We call them *tiers* accordingly to the data ramification principle of [15]. We use $\alpha, \beta, \dots$ for tiers. A variable typing environment Γ is a finite mapping from $\mathbb{V}$ to $\{\mathbf{0}, \mathbf{1}\}$, which assigns a single tier to each variable. An operator typing environment Δ is a mapping that associates to each operator op a set of operator types $\Delta(op)$, where the operator types corresponding to an operator of arity n are of the shape $\alpha_1 \to \dots \alpha_n \to \alpha$ with $\alpha_i, \alpha \in \{\mathbf{0}, \mathbf{1}\}$ using implicit right associativity of $\to$. We write $dom(\Gamma)$ (resp. $dom(\Delta)$) to denote the set of variables typed by Γ (resp. the set of operators typed by Δ). Figure 2 gives the typing discipline for expressions, commands and multi-threaded programs. Given a multi-threaded program M, a variable typing environment Γ and an operator typing environment Δ, M is *well-typed* if for every $x \in dom(M)$, $\Gamma, \Delta \vdash M(x) : \alpha$ for some tier α.

Notice that the subject reduction property is not valid, because we don't explicitly have any subtyping rule. However, a weak subject reduction property holds: If $\mu \vDash C \xrightarrow{s} \mu' \vDash C'$ then $\Gamma, \Delta \vdash C' : \beta$ where $\beta \preceq \alpha$.

3 Safe multi-threaded program

3.1 Neutral and positive operators

As in [18], we define two classes of operators called neutral and positive. For this, let $\trianglelefteq$ be the sub-word relation over $\mathbb{W}$, which is defined by $v \trianglelefteq w$, iff there are u and u' such that $w = u.v.u'$, where . is the concatenation.
An operator *op* is *neutral* if:

$$\frac{\Gamma(X) = \alpha}{\Gamma, \Delta \vdash X : \alpha} \qquad \frac{\Gamma, \Delta \vdash X : \beta \qquad \Gamma, \Delta \vdash E : \alpha}{\Gamma, \Delta \vdash X := E : \beta} \beta \preceq \alpha$$

$$\frac{\Gamma, \Delta \vdash E_1 : \alpha_1 \ldots \Gamma, \Delta \vdash E_n : \alpha_n \qquad \alpha_1 \to \ldots \to \alpha_n \to \alpha \in \Delta(op)}{\Gamma, \Delta \vdash op(E_1, \ldots, E_n) : \alpha}$$

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

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

Fig. 2. Type system for expressions, commands

1. either $[\![op]\!] : \mathbb{W} \to \{\texttt{tt}, \texttt{ff}\}$ is a predicate;
2. or for all $d_1, \ldots, d_n \in \mathbb{W}$, $\exists i \in \{1, \ldots, n\}$, $[\![op]\!](d_1, \ldots, d_n) \trianglelefteq d_i$.

An operator *op* is *positive* if there is a constant c_{op} such that:

$$|[\![op]\!](d_1, \ldots, d_n)| \leq \max_i |d_i| + c_{op}$$

A neutral operator is always a positive operator but the converse is not true. In the remainder, we assume that operators are all neutral or positive.

3.2 Safe environments and safe multi-threaded programs

An operator typing environment Δ is *safe* if for each $op \in dom(\Delta)$ of arity n and for each $\alpha_1 \to \ldots \to \alpha_n \to \alpha \in \Delta(op)$, we have $\alpha \preceq \wedge_{i=1,n}\alpha_i$, and if the operator *op* is positive but not neutral, then $\alpha = \mathbf{0}$.

Now, given Γ a variable typing environment and Δ a operator typing environment, we say that M is a *safe multi-threaded program* if M is well-typed wrt Γ and Δ and Δ is safe.

Intuitively, a tier $\mathbf{0}$ argument is unsafe. This means that it cannot be used as a loop guard. So for "loop-safety" reasons, if an operator has a tier $\mathbf{0}$ argument then the result is necessarily of tier $\mathbf{0}$. In return, a positive operator can increase the size of its arguments. On the other hand, a neutral operator does not increase the size of its arguments. So, we can apply it safely everywhere. The combination of the type system, which guarantees some safety properties on the information flow, and of operator specificities provides time bounds.

Example 1. Given a word d, the operator eq_d tests whether or not its agument begins with the prefix d and *pred* computes the predecessor.

$$[\![eq_d]\!](u) = \begin{cases} = \texttt{tt} & \text{if } u = dw \\ = \texttt{ff} & \text{otherwise} \end{cases} \qquad [\![pred]\!](u) = \begin{cases} = \epsilon & \text{if } u = \epsilon \\ = w & \text{if } u = \ell.w, \ \ell \in \Sigma \end{cases}$$

Both operators are neutral. This means that their types satisfy $\Delta(pred), \Delta(eq_u) \subseteq \{\mathbf{0} \to \mathbf{0}, \mathbf{1} \to \mathbf{1}, \mathbf{1} \to \mathbf{0}\}$ wrt to a safe environment Δ. The operator suc_d adds a prefix d. It is positive, but not neutral. So, $\Delta(suc_d) \subseteq \{\mathbf{1} \to \mathbf{0}, \mathbf{0} \to \mathbf{0}\}$:

$$\textit{(Positive)}\ [\![suc_d]\!](b) = d.b \quad d \in \Sigma$$

4 Sequential and concurrent non-interferences

In this section, we demonstrate that classical non-interference results are obtained through the use of the considered type system. For that purpose, we introduce some intermediate lemmata. The confinement Lemma expresses the fact that no tier **1** variables are modified by a command of tier **0**.

Lemma 1 (Confinement). *Let Γ be a variable typing environment and Δ be a safe operator typing environment. If $\Gamma, \Delta \vdash C : \mathbf{0}$, then every variable assigned to in C is of type $\mathbf{0}$, and C does not contain while loops.*

Proof. By induction on the structure of C. □

The following lemma, called simple security, says that only variables at level **1** will have their content read in order to evaluate an expression E of type **1**.

Lemma 2 (Simple security). *Let Γ be a variable typing environment and Δ be a safe operator typing environment. If $\Gamma, \Delta \vdash E : \mathbf{1}$, then for every $X \in \mathcal{V}(E)$, we have $\Gamma(X) = \mathbf{1}$. Moreover, all operators in E are neutral.*

Proof. By induction on E, and using the fact that E is necessarily only composed of operators of type $\mathbf{1} \to \ldots \to \mathbf{1} \to \mathbf{1}$, because the environment is safe. □

Definition 1. *Let Γ be a variable typing environment and Δ be an operator typing environment.*

- *The equivalence relation $\approx_{\Gamma,\Delta}$ on stores is defined as follows:*
  $\mu \approx_{\Gamma,\Delta} \sigma$ iff for every $X \in dom(\Gamma)$ s.t. $\Gamma(X) = \mathbf{1}$ we have $\mu(X) = \sigma(X)$
- *The relation $\approx_{\Gamma,\Delta}$ is extended to commands as follows:*
  1. *If $C = C'$ then $C \approx_{\Gamma,\Delta} C'$*
  2. *If $\Gamma, \Delta \vdash C : \mathbf{0}$ and $\Gamma, \Delta \vdash C' : \mathbf{0}$ then $C \approx_{\Gamma,\Delta} C'$*
  3. *If $C \approx_{\Gamma,\Delta} C'$ and $D \approx_{\Gamma,\Delta} D'$ then $C; D \approx_{\Gamma,\Delta} C'; D'$*
- *Finally, it is extended to configurations as follows:*
  If $C \approx_{\Gamma,\Delta} C'$ and $\mu \approx_{\Gamma,\Delta} \sigma$ then $\mu \vDash C \approx_{\Gamma,\Delta} \sigma \vDash C'$

Remark 1. A consequence of Lemma 2 is that if $\mu \approx_{\Gamma,\Delta} \sigma$ and if $\Gamma, \Delta \vdash E : \mathbf{1}$, then computations of E are identical under the stores μ and σ , that is $\mu \vDash E \xrightarrow{e} d$ and $\sigma \vDash E \xrightarrow{e} d$.

We now establish a sequential non-interference Theorem which states that if X is variable of tier **1** then the value stored in X is independent from variables of tier **0**.

Theorem 1 (Sequential non-interference). *Assume that Γ is a variable typing environment and Δ is a safe operator typing environment s.t. $\Gamma, \Delta \vdash C : \alpha$ and $\Gamma, \Delta \vdash D : \alpha$. Assume also that $\mu \vDash C \approx_{\Gamma,\Delta} \sigma \vDash D$. Then, we have:*

- *if $\mu \vDash C \xrightarrow{s} \mu' \vDash C'$ then there exists σ' and D' such that $\sigma \vDash D \xrightarrow{s}{}^{*} \sigma' \vDash D'$ and $\mu' \vDash C' \approx_{\Gamma,\Delta} \sigma' \vDash D'$,*
- *if $\mu \vDash C \xrightarrow{s} \mu'$ then there exists σ' such that $\sigma \vDash D \xrightarrow{s}{}^{*} \sigma'$ and $\mu' \approx_{\Gamma,\Delta} \sigma'$*

Proof. First suppose that $\alpha = \mathbf{0}$. Confinement Lemma 1 implies that $\mu' \approx_{\Gamma,\Delta} \sigma'$ since no tier **1** variable is changed. Second suppose that $\alpha = \mathbf{1}$. We proceed by induction on C. Suppose that C is $\texttt{while}(E)\{C_1\}$ and the evaluation under μ is:

$$\frac{\mu \vDash E \xrightarrow{e} \texttt{tt}}{\mu \vDash \texttt{while}(E)\{C_1\} \xrightarrow{s} \mu \vDash C_1; \texttt{while}(E)\{C_1\}} \quad (W_{\texttt{tt}})$$

By Remark 1, the evaluation of E under σ is necessarily $\texttt{tt}$. Since C is an atomic command, $C \approx_{\Gamma,\Delta} D$ implies $C = D$. As a result, $\sigma \vDash \texttt{while}(E)\{C_1\} \xrightarrow{s} \sigma \vDash C_1; \texttt{while}(E)\{C_1\}$. We have $\mu' \approx_{\Gamma,\Delta} \sigma'$ because $\mu = \mu'$ and $\sigma = \sigma'$. We conclude that both configurations are equivalent, that is $\mu' \vDash C' \approx_{\Gamma,\Delta} \sigma' \vDash D'$. The other cases are treated similarly. □

Sequential non-interference can be adapted to multi-threaded programs. For that purpose, we extend the equivalence $\approx_{\Gamma,\Delta}$ to multi-threaded programs by:

- If $\forall x \in dom(M) = dom(M'),\ M(x) \approx_{\Gamma,\Delta} M'(x)$ then $M \approx_{\Gamma,\Delta} M'$
- If $M \approx_{\Gamma,\Delta} M'$ and $\mu \approx_{\Gamma,\Delta} \sigma$ then $\mu \vDash M \approx_{\Gamma,\Delta} \sigma \vDash M'$

Theorem 2 (Concurrent Non-interference). *Assume that Γ is a variable typing environment, that Δ is a safe operator typing environment such that M is well-typed. Assume also that $\mu \vDash M_1 \approx_{\Gamma,\Delta} \sigma \vDash M_2$. Then, if $\mu \vDash M_1 \xrightarrow{g} \mu' \vDash M_1'$ then there are σ' and M_2' s.t. $\sigma \vDash M_2 \xrightarrow{g}{}^{*} \sigma' \vDash M_2'$ and $\mu' \vDash M_1 \approx_{\Gamma,\Delta} \sigma' \vDash M_2$.*

Proof. Consequence of Theorem 1. □

5 Sequential and concurrent temporal non-interferences

Now we establish a property named temporal non-interference. This property ensures that the length of while-loops does not depend on variables of tier **0**, and depends only on tier **1** variables. Consequently, a change in the value of a variable of tier **0** does not affect loop lengths.

For this, we define a loop length measure in Figure 3 based on the small step semantics of Figure 1. $\sigma \vDash_0 C \xrightarrow{s}{}^{*} \sigma' \vDash_t C'$ holds if t is the number of while-loops, which are unfolded to reach $\sigma' \vDash C'$ from $\sigma \vDash C$, that is t is the number of applications of the rule $(TW_{\texttt{tt}})$ in a computation. It is convenient to define the relation $\Rightarrow_t$ by $\sigma \vDash C \Rightarrow_t \sigma' \vDash C'$ iff $\sigma \vDash_0 C \xrightarrow{s}{}^{*} \sigma' \vDash_t C'$.

$$\frac{\mu \vDash E \xrightarrow{\mathsf{e}} d}{\mu \vDash_t X{:=}E \xrightarrow{\mathsf{s}} \mu[X \leftarrow d]} \qquad \frac{}{\mu \vDash_t \mathtt{skip} \xrightarrow{\mathsf{s}} \mu} \qquad \frac{\mu \vDash_t C_1 \xrightarrow{\mathsf{s}} \mu_1}{\mu \vDash_t C_1 \,;\, C_2 \xrightarrow{\mathsf{s}} \mu_1 \vDash_t C_2}$$

$$\frac{\mu \vDash_t C_1 \xrightarrow{\mathsf{s}} \mu_1 \vDash_{t'} C_1'}{\mu \vDash_t C_1 \,;\, C_2 \xrightarrow{\mathsf{s}} \mu_1 \vDash_{t'} C_1'; C_2} \qquad \frac{\mu \vDash E \xrightarrow{\mathsf{e}} w,\ w \in \{\mathtt{tt}, \mathtt{ff}\}}{\mu \vDash_t \mathtt{if}\ E\ \mathtt{then}\ C_{\mathtt{tt}}\ \mathtt{else}\ C_{\mathtt{ff}} \xrightarrow{\mathsf{s}} \mu \vDash_t C_w}$$

$$\frac{\mu \vDash E \xrightarrow{\mathsf{e}} \mathtt{ff}}{\mu \vDash_t \mathtt{while}(E)\{C\} \xrightarrow{\mathsf{s}} \mu} \qquad \frac{\mu \vDash E \xrightarrow{\mathsf{e}} \mathtt{tt}}{\mu \vDash_t \mathtt{while}(E)\{C\} \xrightarrow{\mathsf{s}} \mu \vDash_{t+1} C; \mathtt{while}(E)\{C\}} \ (TW_{tt})$$

$$\frac{M(x) = C \quad \mu \vDash_0 C \xrightarrow{\mathsf{s}} \mu'}{\mu \vDash_t M \xrightarrow{\mathsf{g}} \mu' \vDash_t M - x} \qquad \frac{M(x) = C \quad \mu \vDash_t C \xrightarrow{\mathsf{s}} \mu' \vDash_{t'} C'}{\mu \vDash_t M \xrightarrow{\mathsf{g}} \mu' \vDash_{t'} M[x := C']}$$

Fig. 3. Loop length measure for commands and multi-thread programs

Remark 2. If $\Gamma, \Delta \vdash C : \mathbf{0}$ and $\sigma \vDash C \xrightarrow{\mathsf{s}}{}^{*} \sigma' \vDash C'$ then $\sigma \vDash C \Rightarrow_0 \sigma' \vDash C'$ since there is no while loop inside C, by Lemma 1. Moreover, if $\sigma \vDash C \Rightarrow_t \sigma' \vDash C'$, then for every $k \leq t$ there are σ'' and C'' such that $\sigma \vDash C \Rightarrow_k \sigma'' \vDash C'' \Rightarrow_{t-k} \sigma' \vDash C'$.

Theorem 3 (Temporal non-interference). *Assume that Γ is a variable typing environment and Δ is a safe operator typing environment s.t. $\Gamma, \Delta \vdash C : \alpha$ and $\Gamma, \Delta \vdash D : \alpha$. Assume also that $\mu \vDash C \approx_{\Gamma,\Delta} \sigma \vDash D$. Then, if $\mu \vDash C \Rightarrow_t \mu' \vDash C'$ then there are σ' and D' s.t. $\sigma \vDash D \Rightarrow_t \sigma' \vDash D'$ and $\mu' \vDash C' \approx_{\Gamma,\Delta} \sigma' \vDash D'$.*

Proof. The proof goes by induction on t. Suppose that $t = 0$. This means that no rule *(TW_{tt})* has been fired. The conclusion is a consequence of sequential non-interference Theorem 1.

Next, suppose that $\mu \vDash C \Rightarrow_{t+1} \mu' \vDash C'$. This means that a rule *(TW_{tt})* has been applied. So suppose that $C = \mathtt{while}(E)\{C_1\}$ and that $\mu \vDash E \xrightarrow{\mathsf{e}} \mathtt{tt}$. First, $\mu \approx_{\Gamma,\Delta} \sigma$ and Lemma 2 imply that $\sigma \vDash E \xrightarrow{\mathsf{e}} \mathtt{tt}$. Second, since $C \approx_{\Gamma,\Delta} D$, we have $C = D$, by definition of $\approx_{\Gamma,\Delta}$. Since $C' = C_1; C$, we have $D' = C_1; C$. Thus, $C' \approx_{\Gamma,\Delta} D'$ and $\sigma \vDash D \Rightarrow_{t+1} \sigma' \vDash D'$ hold. Moreover, we have $\mu' = \mu$ and $\sigma' = \sigma$, which implies that $\mu' \approx_{\Gamma,\Delta} \sigma'$. We conclude that $\mu' \vDash C' \approx_{\Gamma,\Delta} \sigma' \vDash D'$. The other cases are similar. □

We extend the relation $\Rightarrow_t$ as follows: $\mu \vDash M \Rightarrow_t \mu' \vDash M'$ if and only if $\mu \vDash_0 M \xrightarrow{\mathsf{g}}{}^{*} \mu' \vDash_t M'$. As a corollary, we obtain a temporal non-interference result for multi-threaded programs.

Theorem 4 (Concurrent temporal non-interference). *Assume Γ is a variable typing environment and Δ is a safe operator typing environment s.t. M and N are well typed. Assume that $\mu \vDash M \approx_{\Gamma,\Delta} \sigma \vDash N$. Then, if $\mu \vDash M \Rightarrow_t \mu' \vDash M'$ then there are σ' and N' s.t. $\sigma \vDash N \Rightarrow_t \sigma' \vDash N'$ and $\mu' \vDash M' \approx_{\Gamma,\Delta} \sigma' \vDash N'$.*

Proof. Consequence of Theorem 3. □

6 Multi threaded program running time

An important point is that the number of tier **1** configurations in a computation is polynomially bounded in the size of tier **1** initial values.

Lemma 3. *Let M be a safe multi-threaded program wrt environments Γ and Δ. If $\mu \vDash M \Rightarrow_t \mu' \vDash M'$ then $\forall X \in \mathcal{V}(M)$ such that $\Gamma(X) = \mathbf{1}$ either $\mu'(X) \in \{\mathtt{tt}, \mathtt{ff}\}$ or $\exists Y \in \mathcal{V}(M)$ such that $\Gamma(Y) = \mathbf{1}$ and $\mu'(X) \trianglelefteq \mu(Y)$.*

Proof. Take one global computational step $\mu \vDash M \xrightarrow{g} \mu' \vDash M'$. Let X be a variable assigned to in $M(x)$, for some thread identifier x, such that $\Gamma(X) = \mathbf{1}$. X can only be assigned to an expression E of tier **1**. By simple security lemma 2, E only contains neutral operators. It means that either $\mu'(X)$ is a truth value (corresponding to the computation of a predicate) or a subterm of a value of a tier **1** variable. □

In the case where a multi-threaded program strongly terminates (i.e. $M \Downarrow$), we now establish that for all thread interactions, the maximal length of while-loops is polynomially bounded in the size of tier **1** values of the initial store. This is a consequence of the temporal non-interference property. For this, define $\|-\|_{\mathbf{1}}$ by $\|\mu\|_{\mathbf{1}} = \max_{\Gamma(X)=\mathbf{1}} |\mu(X)|$.

Theorem 5. *Let M be a safe multi-threaded program such that $M \Downarrow$. There is a polynomial T such that for all stores μ, if $\mu \vDash M \Rightarrow_t \mu' \vDash M'$ then $t \leq T(\|\mu\|_{\mathbf{1}})$.*

Proof. By Theorem 4, the length of while-loops depends only on variables of tier **1**. It implies that if we enter twice into a configuration with the same thread, say x, and the same values of tier **1**, we know that M is non-terminating. Indeed, it is possible to repeat the same transition again up to infinity by always firing the same sequence of global transitions. This contradicts the fact that $M \Downarrow$. Consequently, we never enter twice in the same thread configuration. Since the number of sub-words of a word of size n is bounded by n^2, Lemma 3 impies the number of distinct stores σ reachable from μ is bounded polynomially by $\|\mu\|_{\mathbf{1}}$. It follows the number of configurations is polynomially bounded. Consequently there exists a polynomial T such that the length of each terminating multi-threaded computation starting from μ is bounded by $T(\|\mu\|_{\mathbf{1}})$. Finally, we have that $t \leq T(\|\mu\|_{\mathbf{1}})$. □

We can now state our first main result:

Theorem 6. *Assume that M is a safe multi-threaded program. Moreover suppose that M strongly terminates. There is a polynomial Q such that:*

$$\forall d_1, \ldots, d_n \in \mathbb{W},\ \mathit{Time}_M(d_1, \ldots, d_n) \leq Q(\max_{i=1,n}(|d_i|))$$

Proof. Suppose that $\mu_0[X_1 \leftarrow d_1, \ldots, X_n \leftarrow d_n] \vDash M \Rightarrow_t \mu' \vDash \emptyset$. The overall computational time is bounded by $\mathit{Time}_M(d_1, \ldots, d_n) \leq r.t + r$, for some constant r which depends on the size of M. (Note that commands of tier **0** are computable in constant size.) We conclude by Theorem 5 and by setting $Q(X) = r.T(X) + r$. □

7 A characterization of polynomial time functions

We now come to a characterization of the set of functions computable in polynomial time. A sequential program M consists in a single thread program (i.e. $dom(M) = \{x\}$) and an output variable, say Y. The partial function $[\![M]\!]$ computed by M is then defined by:
$[\![M]\!](d_1, \ldots, d_n) = w$ iff $\mu_0[X_1 \leftarrow d_1, \ldots, X_n \leftarrow d_n] \vDash M \xrightarrow{\mathtt{g}}{}^{*} \mu \vDash \emptyset$ and $\mu(Y) = w$

Theorem 7. *The set of functions computed by strongly terminating and safe sequential programs whose operators compute polynomial time functions is exactly* FPtime*, which is the set of polynomial time computable functions.*

Proof. The polynomial runtime upper bound is a consequence of Theorem 5. The converse is a straightforward simulation of polynomial time Turing machines. The proof is postponed in Appendix.

8 Deterministic scheduling

Actually, we can extend our results to a class of deterministic schedulers. Till now, we have considered a non-deterministic scheduling policy but in return we require that multi-threaded programs strongly terminate. Define $\mu \downarrow 1$ as the restriction of the store μ to tier **1** variables. Say that a deterministic scheduler $\mathcal{S}$ is *quiet* if the scheduling policy depends only on the current multi-threaded program M and on $\mu \downarrow 1$. For example, a deterministic scheduler whose policy just depends on running threads, is quiet. Notice that $\sigma \approx_{\Gamma,\Delta} \sigma'$ iff $\sigma \downarrow 1 = \sigma' \downarrow 1$. Next, we replace the non-deterministic global transition of Figure 1 by:

$$\frac{\mathcal{S}(M, \mu\downarrow 1) = x \quad \mu \vDash M(x) \xrightarrow{\mathtt{s}} \mu'}{\mu \vDash M \xrightarrow{\mathtt{g}} \mu' \vDash M - x} \qquad \frac{\mathcal{S}(M, \mu\downarrow 1) = x \quad \mu \vDash M(x) \xrightarrow{\mathtt{s}} \mu' \vDash C'}{\mu \vDash M \xrightarrow{\mathtt{g}} \mu' \vDash M[x := C']}$$

Theorem 8. *Let M be a safe multi-threaded program s.t. M is terminating wrt a deterministic and quiet scheduler $\mathcal{S}$. There is a polynomial Q such that:*

$$\forall d_1, \ldots, d_n \in \mathbb{W},\ \ Time_M(d_1, \ldots, d_n) \leq Q(\max_{i=1,n}(|d_i|))$$

Proof. The proof follows the outline of proofs of theorems 5 and 6. Let μ be the initial store, i.e. $\mu(X_i) = d_i$ for $i = 1, n$ and $\mu(X_i) = \epsilon$ for $i > n$. Since the computation of $\mu \vDash M$ terminates wrt $\mathcal{S}$, the temporal non-interference theorem 3 implies that a loop can not reach the configurations $\sigma \vDash N$ and $\sigma' \vDash N$ where their restrictions to tier **1** values are identical. That is $\sigma \downarrow 1 = \sigma' \downarrow 1$. Now, define $Config = \{(\sigma \downarrow 1, N) \mid \mu \vDash M \xrightarrow{\mathtt{g}}{}^{*} \sigma \vDash N\}$. The total length of loops is bounded by the cardinality of *Config*. Following lemma 3, the cardinality of *Config* is bounded by a polynomial in $\|\mu\|_{\mathbf{1}}$. As a result, the runtime of $\mu \vDash M$ is bounded bounded by $Q(\max_{i=1,n}(|d_i|))$ for some polynomial Q.

References

1. R. M. Amadio and F. Dabrowski. Feasible reactivity in a synchronous pi-calculus. In *PPDP*, pages 221–230, 2007.
2. R. M. Amadio and S. Dal-Zilio. Resource control for synchronous cooperative threads. In *CONCUR*, pages 68–82, 2004.
3. Patrick Baillot and Kazushige Terui. Light types for polynomial time computation in lambda-calculus. In *LICS*, IEEE Computer Society Press, pages 266–275, 2004.
4. D. E. Bell and L.J. La Padula. Secure computer system: unified exposition and multics interpretation. Technical report, Mitre corp Rep., 1976.
5. S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2:97–110, 1992.
6. K. Biba. Integrity considerations for secure computer systems. Technical report, Mitre corp Rep., 1977.
7. G. Bonfante, J.Y. Marion, and J.Y. Moyen. Quasi-interpretations a way to control resources. *Theo. Comput. Sci.*, 2011.
8. I. Castellani and G. Boudol. Non-interference for concurrent programs. In *ICALP*, volume 2076 of *Lecture Notes in Computer Science*, pages 382–395, 2001.
9. B. Cook, A. Podelski, and A. Rybalchenko. Proving thread termination. In *PLDI*, pages 320–330, 2007.
10. J.-Y. Girard. Light linear logic. *Inf. Comput.*, 143(2):175–204, 1998.
11. M. Hofmann. Linear types and non-size-increasing polynomial time computation. *Inf. Comput.*, 183(1):57–85, 2003.
12. N. Jones. The expressive power of higher-order types or, life without cons. *J. Funct. Program.*, 11(1):5–94, 2001.
13. N. Jones and L. Kristiansen. A flow calculus of *wp*-bounds for complexity analysis. *ACM Trans. Comput. Log.*, 10(4), 2009.
14. N.D. Jones. *Computability and complexity, from a programming perspective*. MIT press, 1997.
15. D. Leivant. A foundational delineation of poly-time. *Inf. Comput.*, 110(2):391–420, 1994.
16. D. Leivant. Predicative recurrence and computational complexity i: Word recurrence and poly-time. In Peter Clote and Jeffery Remmel, editors, *Feasible Mathematrics II*. 1994.
17. A. Madet and R. M. Amadio. An elementary affine lambda-calculus with multithreading and side effects. In *TLCA*, pages 138–152, 2011.
18. J.-Y. Marion. A type system for complexity flow analysis. In *LICS*, pages 123–132, 2011.
19. J.Y. Marion and R. Péchoux. Sup-interpretations, a semantic method for static analysis of program resources. *ACM TOCL*, 10(4):27, 2009.
20. K.-H. Niggl and H. Wunderlich. Certifying polynomial time and linear/polynomial space for imperative programs. *SIAM J. Comput.*, 35(5):1122–1147, 2006.
21. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, January 2003.
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

A.1 Proofs

Characterization of polynomial time functions

Theorem 7. *The set of functions computed by strongly terminating and safe sequential programs whose operators compute polynomial time functions is exactly* FPtime*, which is the set of polynomial time computable functions.*

Proof. By Theorem 6, the execution time of a safe and strongly terminating sequential program is bounded by a polynomial in the size of the initial values. In the other direction, we show that every polynomial time function over the set of words $\mathbb{W}$ can be computed by a safe and terminating program. Consider a Turing Machine TM, with one tape and one head, which computes within n^k steps for some constant k and where n is the input size. The tape of TM is represented by two variables `Left` and `Right` which contain respectively the reversed left side of the tape and the right side of the tape. States are encoded by constant words and the current state is stored in the variable `State`. We assign to each of these three variables that hold a configuration of TM the tier $\mathbf{0}$. A one step transition is simulated by a finite cascade of if-commands of the form:

```
if eq_a(Right^0)^0
    then
       if eq_s(State^0)^0
        then
           State^0:=s'^0;: 0
           Left^0:=suc_b(Left^0);: 0
           Right^0:=pred(Right^0) : 0
        else ... : 0
   ...
```

The above command expresses that if the current read letter is a and the state is s, then the next state is s', the head moves to the right and the read letter is replaced by b. Since each variable inside the above command is of type $\mathbf{0}$, the type of the if-command is also $\mathbf{0}$. Moreover, since suc_b is a positive operator, its type is forced to be $\mathbf{0} \to \mathbf{0}$. eq_a, eq_s and $pred$ being neutral operators, they can also be typed by $\mathbf{0} \to \mathbf{0}$.
Finally, it just remains to show that every polynomial can be simulated by a safe program of tier $\mathbf{1}$. We have already provided the programs for addition and multiplication in Example 2 and we let the reader check that it can be generalized to any polynomial. □

A.2 Examples

In what follows, let E^α, respectively $C : \alpha$, be a notation meaning that the expression E, respectively command C, is of type α under the considered typing environments.

Example 2. Consider the sequential programs add_Y and mul_Z that compute respectively addition and multiplication on unary words using the positive successor operator $+1$, in infix notation, and two neutral operators, -1 and a unary predicate > 0, both in infix notation. Both programs are safe by checking that their main commands are well-typed wrt the safe operator typing environment Δ defined by $\Delta(+1) = \{\mathbf{0} \to \mathbf{0}\}$ and $\Delta(-1) = \Delta(> 0) = \{\mathbf{1} \to \mathbf{1}\}$.

add_Y :

$\texttt{while}(X^{\mathbf{1}} > 0)^{\mathbf{1}}\{$
$\quad X^{\mathbf{1}} := X^{\mathbf{1}} - 1; : \mathbf{1}$
$\quad Y^{\mathbf{0}} := Y^{\mathbf{0}} + 1 : \mathbf{0}$
$\} : \mathbf{1}$

mul_Z :

$Z^{\mathbf{0}} := 0^{\mathbf{0}}; : \mathbf{0}$
$\texttt{while}(X^{\mathbf{1}} > 0)^{\mathbf{1}}\{$
$\quad X^{\mathbf{1}} := X^{\mathbf{1}} - 1; : \mathbf{1}$
$\quad U^{\mathbf{1}} := Y^{\mathbf{1}}; : \mathbf{1}$
$\quad \texttt{while}(Y^{\mathbf{1}} > 0)^{\mathbf{1}}\{$
$\quad\quad Y^{\mathbf{1}} := Y^{\mathbf{1}} - 1; : \mathbf{1}$
$\quad\quad Z^{\mathbf{0}} := Z^{\mathbf{0}} + 1 : \mathbf{0}$
$\quad \}; : \mathbf{1}$
$\quad Y^{\mathbf{1}} := U^{\mathbf{1}} : \mathbf{1}$
$\} : \mathbf{1}$

Example 3. Consider the following multi-thread M composed of two threads x and y computing on unary numbers:

x :

$\mathbf{while}\ (X^{\mathbf{1}} > 0)^{\mathbf{1}}\{$
$\quad Z^{\mathbf{0}} := Z^{\mathbf{0}} + 1; : \mathbf{0}$
$\quad X^{\mathbf{1}} := X^{\mathbf{1}} - 1; : \mathbf{1}$
$\} : \mathbf{1}$

y :

$\mathbf{while}\ (Y^{\mathbf{1}} > 0)^{\mathbf{1}}\{$
$\quad Z^{\mathbf{0}} = 0; : \mathbf{0}$
$\quad Y^{\mathbf{1}} := Y^{\mathbf{1}} - 1; : \mathbf{1}$
$\} : \mathbf{1}$

This program is strongly terminating. Moreover, given a store μ such that $\mu(X) = n$ and $\mu(Z) = 0$, if $\mu \vDash M \xrightarrow{\mathtt{g}}{}^{k} \mu' \vDash \emptyset$ then $\mu'(Z) \in [0, n]$. M is safe using an operator typing environment Δ such that $\Delta(-1) = \Delta(> 0) = \{\mathbf{1} \to \mathbf{1}\}$ and $\Delta(+1) = \{\mathbf{0} \to \mathbf{0}\}$ and $M \Downarrow$. Consequently, by Theorem 5, there is a polynomial T such that for each store μ, $k \leq T(\|\mu\|_{\mathbf{1}})$.

Example 4. Consider the following multi-thread M that shuffles two strings given as inputs:

x :

$\mathbf{while}\ (\neg eq_\epsilon(X^{\mathbf{1}}))^{\mathbf{1}}\{$
$\quad Z^{\mathbf{0}} := concat(head(X^{\mathbf{1}}), Z^{\mathbf{0}}); : \mathbf{0}$
$\quad X^{\mathbf{1}} := pred(X^{\mathbf{1}}); : \mathbf{1}$
$\} : \mathbf{1}$

y :

$\mathbf{while}\ (\neg eq_\epsilon(Y^{\mathbf{1}}))^{\mathbf{1}}\{$
$\quad Z^{\mathbf{0}} := concat(head(Y^{\mathbf{1}}), Z^{\mathbf{0}}); : \mathbf{0}$
$\quad Y^{\mathbf{1}} := pred(Y^{\mathbf{1}}); : \mathbf{1}$
$\} : \mathbf{1}$

The negation operator $\neg$ and eq_ϵ are unary predicates and consequently can be typed by $\mathbf{1} \to \mathbf{1}$. The operator *head* returns the first symbol of a string given as input and can be typed by $\mathbf{1} \to \mathbf{0}$ since it is neutral. The *pred* operator can typed by $\mathbf{1} \to \mathbf{1}$ since its computation is a subterm of the input. Finally, the concat operator that performs the concatenation of the symbol given as first argument with the second argument can be typed by $\mathbf{0} \to \mathbf{0} \to \mathbf{0}$ since $|[\![concat]\!](u, v)| = |v| + 1$. This program is safe and strongly terminating consequently it also terminates in polynomial time.

Example 5. Consider the following multi-thread M:

x :
$\quad \mathbf{while}\ (X^{\mathbf{1}} > 0)^{\mathbf{1}}\{$
$\quad\quad Y^{\mathbf{1}} := X^{\mathbf{1}};: \mathbf{1}$
$\quad\quad X^{\mathbf{1}} := X^{\mathbf{1}} - 1;: \mathbf{1}$
$\quad \} : \mathbf{1}$

y :
$\quad \mathbf{while}\ (Y^{\mathbf{1}} > 0)^{\mathbf{1}}\{$
$\quad\quad Z^{\mathbf{0}} := Z^{\mathbf{0}} + 1;: \mathbf{0}$
$\quad\quad Y^{\mathbf{1}} := Y^{\mathbf{1}} - 1;: \mathbf{1}$
$\quad \} : \mathbf{1}$

Observe that, contrarily to previous examples, the guard of y depends on information flowing from X to Y. Given a store μ such that $\mu(X) = n$, $\mu(Y) = \mu(Z) = 0$, if $\mu \vDash M \overset{\mathtt{g}}{\to}^{k} \mu' \vDash \emptyset$ then $\mu'(Z) \in [0, n \times (n+1)/2]$. This multi-thread is safe with respect to a safe typing operator environment Δ such that $\Delta(-1) = \Delta(> 0) = \{\mathbf{1} \to \mathbf{1}\}$ and $\Delta(+1) = \{\mathbf{0} \to \mathbf{0}\}$. Moreover it strongly terminates. Consequently, it also terminates in polynomial time.

Example 6. The following program computes the exponential:

$exp_Y(X^{\mathbf{1}}, Y^{\mathbf{0}})$:
$\quad \mathtt{while}(X^{\mathbf{1}} > 0)\{$
$\quad\quad U^{?} := Y^{\mathbf{0}};: ?$
$\quad\quad \mathtt{while}(U^{?} > 0)\{$
$\quad\quad\quad Y^{\mathbf{0}} := Y^{\mathbf{0}} + 1;: \mathbf{0}$
$\quad\quad\quad U^{?} := U^{?} - 1 : ?$
$\quad\quad \};: \mathbf{1}$
$\quad\quad X^{\mathbf{1}} := X^{\mathbf{1}} - 1 : \mathbf{1}$
$\quad \};: \mathbf{1}$

It is not typable in our formalism. Indeed, suppose that it is typable. The command $Y := Y + 1$ enforces Y to be of tier $\mathbf{0}$ since $+1$ is positive. Consequently, the command $U := Y$ enforces U to be of tier $\mathbf{0}$ because of typing discipline for assignments. However, the innermost while loop enforces $U > 0$ to be of tier $\mathbf{1}$, so that U has to be of tier $\mathbf{1}$ (because $\mathbf{0} \to \mathbf{1}$ is not permitted for a safe operator typing environment) and we obtain a contradiction.

Example 7. As another counter-example, consider now the addition *badd* on binary words:

$badd_Y$:
$\quad \mathtt{while}(X^{?} > 0)^{?}\{$
$\quad\quad X^{?} := X^{?} - 1;: ?$
$\quad\quad Y^{\mathbf{0}} := Y^{\mathbf{0}} + 1 : \mathbf{0}$
$\quad \} : \mathbf{1}$

Contrarily to Example 2, the above program is not typable because the operator -1 has now type $\Delta(-1) = \{\mathbf{0} \to \mathbf{0}\}$. Indeed it cannot be neutral since binary predecessor is not a subterm operator. Consequently, -1 is positive and the assignment $X := X - 1$ enforces X to be of type $\mathbf{0}$ whereas the loop guard enforces X to be of tier $\mathbf{1}$. Note that this counter-example is not that surprising in the sense that a binary word of size n may lead to a loop of length 2^n using the -1 operator. Of course this does not imply that the considered typing discipline

rejects computations on binary words, it only means that this type system rejects exponential time programs. Consequently,"natural" binary addition algorithms are captured as illustrated by the following program that computes the binary addition on reversed binary words of equal size:

$binary\_add_Z$:
  $\texttt{while}(\neg eq_\epsilon(X^{\mathbf{1}}))^{\mathbf{1}}\{$
    $R^{\mathbf{0}}{:=}result(bit(X^{\mathbf{1}}), bit(Y^{\mathbf{1}}), bit(C^{\mathbf{1}}));:\mathbf{0}$
    $C^{\mathbf{1}}{:=}carry(bit(X^{\mathbf{1}}), bit(Y^{\mathbf{1}}), bit(C^{\mathbf{1}}));:\mathbf{1}$
    $Z^{\mathbf{0}}{:=}concat(R^{\mathbf{0}}, Z^{\mathbf{0}});:\mathbf{0}$
    $X^{\mathbf{1}}{:=}pred(X^{\mathbf{1}});:\mathbf{1}$
    $Y^{\mathbf{1}}{:=}pred(Y^{\mathbf{1}});:\mathbf{1}$
  $\}:\mathbf{1}$

As usual, *pred* is typed by $\mathbf{1} \to \mathbf{1}$. The negation operator $\neg$ and eq_ϵ are predicates and, consequently, can be typed by $\mathbf{1} \to \mathbf{1}$, since they are neutral. The operator *bit* returns `tt` or `ff` depending on whether the word given as input has first digit 1 or 0, respectively. Consequently, it can be typed by $\mathbf{1} \to \mathbf{1}$. The operators *carry* and *result*, that compute the carry and the result of bit addition, can be typed by $\mathbf{1} \to \mathbf{1} \to \mathbf{1} \to \mathbf{1}$ since they are neutral. Finally, the operator $concat(x, y)$ defined by if $[\![bit]\!](x) = i, i \in \{0, 1\}$ then $[\![concat]\!](x, y) = i.y$ is typed by $\mathbf{0} \to \mathbf{0} \to \mathbf{0}$. Indeed it is a positive operator since $|[\![concat]\!](x, y)| = |y| + 1$.