Complexity Information Flow
in a Multi-threaded Imperative Language

Jean-Yves Marion and Romain Péchoux
Université de Lorraine, CNRS and INRIA
LORIA, UMR 7503
Nancy, France
{jean-yves.marion,romain.pechoux}@loria.fr

Abstract. In this paper, 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.

Keywords: Implicit computational complexity, Ptime, multi-threaded imperative language, non-interference, type system.

1 Introduction

The objective of this paper is to study the notion of complexity flow analysis introduced in [2013] 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 outcome 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 5). Moreover, the runtime upper bound holds for all thread interactions. As a simple example, consider the two-thread program:

\[
\begin{align*}
x & : \text{while}(X^1 == Y^1)\{\text{skip}\} \\
& \quad C; \\
X^1 & := \neg X^1
\end{align*}
\]

\[
\begin{align*}
y & : \text{while}(X^1 \neq Y^1)\{\text{skip}\} \\
& \quad C'; \\
Y^1 & := \neg Y^1
\end{align*}
\]
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 boolean negation. Both threads are safe if commands \(C\) and \(C'\) are safe with respect to the same typing environment. Our first result establishes 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 9). 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 outcome of this paper is that a safe multi-threaded program which is terminating wrt a deterministic and quiet scheduler, runs in polynomial time. Despite the fact that it is not strongly terminating, the two-thread program (see below) terminates under a round-robin scheduler, if \(C\) and \(C'\) terminate.

\[
x : \text{while}(X^1 > 0) \quad y : \text{while}(Z^1 > 0)
\{C; \quad \{C';
\quad Z^1:=0 : 1 \} : 1 \quad X^1:=0 : 1 \} : 1
\]

As a result, if commands \(C\) and \(C'\) are safe, then this two-thread program runs in polynomial time wrt to a round-robin scheduler. The last outcome consists in 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 rationale behind our type system comes from data-ramification concept of Bellantoni and Cook [5] and Leivant [17]. The type system has two atomic types 0 and 1 that we call 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 to 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 the data-ramification concept of [5,17], tier 1 parameters correspond to variables on which recursion is performed whereas tier 0 parameters correspond to variables on which recursion is forbidden.

The second rationale behind our type system comes from secure flow analysis. In order to have an overview on information flow analysis, see Sabelfeld and Myers survey [24]. In [26] for sequential imperative programs and in [25] for multi-threaded imperative programming language, Irvine, Smith and Volpano define a type system to certify confidentiality policies. Types are based on security levels say H (High) and L (Low). The type system prevents the leak of information from level H to level L. This is similar to the principle governing our type system: 0 (resp. 1) corresponds to H (resp. L). In fact, our approach rather coincides with an integrity policy [6] (i.e the rule ”no read down”) than with a confidentiality one [4]. A key notion is non-interference. We establish a first non-interference result which states that values stored in tier 1 variables are