A type system for complexity flow analysis

Jean-Yves Marion

A type system for complexity flow analysis

Jean-Yves Marion
Nancy-Université
INPL-ENSMN
LORIA

***Abstract*—We propose a type system for an imperative programming language, which certifies program time bounds. This type system is based on secure flow information analysis. Each program variable has a level and we prevent information from flowing from low level to higher level variables. We also introduce a downgrading mechanism in order to delineate a broader class of programs. Thus, we propose a relation between security-typed language and implicit computational complexity. We establish a characterization of the class of polynomial time functions.**

I. Introduction

This paper introduces a new static analysis method for controlling and certifying imperative program runtime. We propose a type system to reflect information flow and to explore the computational complexity properties of well-typed programs. We exhibit a relation between Implicit Computational Complexity approaches and research on language-based information flow security.

The subject of Implicit Computational Complexity (ICC) is the characterization of complexity classes without referring to machines and so with no externally imposed resource bounds on time or space. Early examples of implicit characterization of PTIME are via bounded recursion by Cobham [1], fixpoint logic by Immerman [2], positive-existential comprehension in second-order logic by Leivant [3], safe recursion by Bellantoni & Cook [4], ramified recursion by Leivant [5], restricted lambda-calculi by Leivant & Marion [6], light linear logic by Girard [7] and Lafont [8], linear types by Hofmann [9], and syntactically restricted induction [10], to mention a few.

Since Bell & La Padula [11] and Biba [12], secure information flow uses a lattice of security levels to ensure confidentiality and integrity. Each variable is assigned a security level. The type system guarantees the security policy induced by the lattice. An important property, called non-interference [13], consists in demonstrating that values of high level variables are independent from values of low level variables during a program execution. The use of type systems in this context was pioneered by Volpano, Smith and Irvine [14]. Type systems for secure information flow were investigated extensively see e.g. Sabelfeld and Myers survey [15].

Another important issue is declassification. Sometimes it is necessary to release information, which is usually done by downgrading the security level of variables. In the context of integrity, declassification is also called endorsement. For a comprehensive survey, see the article by Sabelfeld and Sands [16] .

Related works

There are several works related to ICC on resource analysis for imperative programming language. In [17] Jones characterizes PTIME by means of simple constructor free programs. Data flow analysis to bound variables growth rate is performed by matrix calculus in Jones & Kristiansen [18], Niggl & Wunderlich [19], and Ben-Amram, Jones & Kristiansen [20]. However the language is restricted to loops of fixed length, and operators are successor-like functions. Finally, Marion & Péchoux [21] extend interpretation methods [22] to object oriented languages.

There are other recent approaches to compute resource consumption bounds. Jost, Hammond, Loidl & Hofmann [23] work on automatic amortized cost analysis using type systems, which is related to the study of non-size increasing systems. Hughes, Pareto & Sabry [24] propose sized types in order to determine bound on data structure sizes. Albert, Alonso, Arenas, Genaim, & Puebla [25] design cost analyser for java byte codes. Lastly, the approach of Gulwani, Mehra & Chilimbi [26] consists in automatically finding loop invariants and is related to abstract interpretation methods.

Contributions

We present a type system for an imperative programming language over words. Programs are built from while loops and expressions are built from predicates, constructor and destructor operators. To control the size of the information flow, the type system is based on a complexity lattice. Each variable is assigned a type, which is a pair of tiers, i.e. elements of the complexity lattice. There is a declassification mechanism, which allows applying safely operators inside loops. We demonstrate that the type system is sound with respect to a timed structural operational semantics. For this, we establish first a non-interference result, which shows that values of higher tier variables do not depend on values of lower tier variables. Then, we show that a consequence is a theorem of temporal non-interference: the length of loops depends only on values of higher tier variables. Another consequence is a weak polynomial time termination procedure. Finally, we demonstrate that terminating and well-typed programs are computable in polynomial time over the two tier lattice $(\{\mathbf{0},\mathbf{1}\},\preceq,\mathbf{0})$. Conversely, each polynomial time function is implemented by a well-typed program.

Informal motivation

We give an informal presentation of the main ideas behind a type system for complexity flow analysis. Our point of departure is the definition of functions by ramified recursion as suggested by Leivant in [5] and in essence in [27] twenty years ago. As a result, some kinds of circular definitions are not allowed. This underlying concept of circularity is one of the most important in implicit computational complexity. Thus, Simmons [28] proposed a fine-grained study of diagonalisation to delineate primitive recursion definitions from multiple recursive definitions. His work is extended in [29] to show how an Ackermann style construction may be used to diagonalise away from the class of polynomial time functions. Another illustration is given by Light Linear Logic in which modalities are essential in order to prevent circular definitions, as Girard explains it in his seminal work [7] . It is a guideline in order to define an information flow typing to control complexity of a simple imperative programming language. What we have in mind is to get inspiration from security-type systems like the one of Volpano, Smith and Irvine [14]. A type system is defined with respect to a security policy, which specifies the use of data. Each variable is assigned a security type. Let us examine a typing rule which expresses by subtyping an explicit downward flow from E to X :

$$\frac{\Gamma \vdash X : \eta \qquad \Gamma \vdash E : \zeta}{\Gamma \vdash X{:=}E : \zeta}\, \eta \leq \zeta$$

There is a data flow from type ζ to η, which corresponds to the *no-read-down-rule* of Biba's integrity policy [12]. The converse should not be possible. There are implicit information flows in while-loop, which should also be controlled. Consider the following program which copies the value of the variable X into Y:

```
Y:=0;
while (X > 0)
      { X:=X − 1;
        Y:=Y + 1;}
```

Here, the variable X controls the while loop. In order to bound the complexity data flow, we should require that the level of Y is strictly less than the one of X. Otherwise, a circular data flow is created, which violates the ICC concept of data-ramification principle by allowing an upward flow. So, the typing rule of a while-command looks like

$$\frac{\Gamma, \Delta \vdash E : \zeta \qquad \Gamma, \Delta \vdash C : \eta}{\Gamma, \Delta \vdash \texttt{while}(E)\{C\} : \eta}\, \eta \prec \zeta$$

On the one hand, the guard E is of level ζ. This implies that X is also of level ζ. On the other hand, the block of commands $X{:=}X - 1$; $Y{:=}Y + 1$; is of type η and $\eta \prec \zeta$. Therefore, we have to downgrade or to declassify X in such a way that the command $X{:=}X - 1$ is of type η.

In order to deal with declassification, we must extend the assignment of single level tags to expressions and to commands. From now on, the type of an expression or of a command is a pair (α, β) of tiers. Roughly speaking, α is the starting level. β indicates that the information is allowed to flow down to level β. An expression E of type (α, β) is downgraded if $\beta \prec \alpha$. A downgraded expression may be safely assigned in a loop of tier α if $\beta \prec \alpha$. Since the runtime is controlled within a polynomial bound, variable and operator typing rules control the declassification process. Thus, we can only apply neutral operators to a declassified expression. We call an operator neutral if it does not increase the size of a value, as in $X - 1$. On the other hand, some operators (called positive operators) reclassify an expression because they are unsafe in a loop, like $X + 1$. We have informally and briefly presented ideas that we are now describing in a more formal way.

II. A COMPLEXITY FLOW TYPE SYSTEM

A. Syntax of expressions and commands

We consider a simple imperative programming language. The syntax of expressions is given by the grammar $E_1, \ldots, E_n \in \text{Exp} ::= X \mid op(E_1, \ldots, E_n)$ where X is a variable from a set $\mathbb{V}$, and op is a n-ary operator from a set $\mathbb{O}$. Constants are 0-ary operators. The set of variables in the expression E is denoted $\mathcal{V}(E)$. The grammar of commands is

$$\begin{aligned} C, C' \in \text{Cmd} \quad ::= \quad & X{:=}E \mid \texttt{while}(E)\{C\} \\ & \mid \texttt{if } E \texttt{ then } C \texttt{ else } C' \\ & \mid C \ ; \ C' \end{aligned}$$

We denote by $|E|$ ($|C|$) the size of an expression E (resp. of a command C).

B. Complexity lattices

A complexity lattice is a finite lattice $(\mathbb{SC}, \preceq, \mathbf{0})$ where $\preceq$ is a quasi-ordering on $\mathbb{SC}$, $\mathbf{0}$ is the least element with respect to $\mathbb{SC}$, such that for any two elements α and β, there is a unique greatest lower bound $\alpha \wedge \beta$ and a unique least upper bound $\alpha \vee \beta$ w.r.t. $\preceq$. Elements of $\mathbb{SC}$ are also dubbed *tier*, for which we use discourse variables $\alpha, \beta, \ldots$ In particular, we shall focus on the two tier lattice $(\{\mathbf{0}, \mathbf{1}\}, \preceq, \mathbf{0})$. We posit a fixed complexity lattice $(\mathbb{SC}, \preceq, \mathbf{0})$, which is implicit in the rest of this paper.

C. Expression types

The *type* of an expression is a pair (α, β) of tiers $\alpha, \beta \in \mathbb{SC}$. The typing derivation system is determined by the typing rules in Figure 1 and two typing environments: (i) a variable typing environment $\Gamma : \mathbb{V} \mapsto \mathbb{SC}$, which assigns to each variable a single tier, and (ii) an operator typing environment Δ, which is a relation that associates to each operator one or several operator types. The grammar of operator types is the following:

$$\rho \in \text{Operator types} \quad ::= \quad (\alpha, \beta) \mid (\alpha, \beta) \to \rho$$

We write $dom(\Gamma)$ (resp. $dom(\Delta)$) to mean the set of variables typed by Γ (resp. the set of operators typed by Δ). Lastly, we occasionally omit parenthesis using right associativity of $\to$.

$$\textit{Variable}\quad \frac{\Gamma(X)=\alpha}{\Gamma,\Delta\vdash X:(\alpha,\beta)}\ \textit{where}\ \beta\preceq\alpha$$

$$\textit{Op}\quad \frac{\Gamma,\Delta\vdash E_1:(\alpha_1,\beta_1)\ldots\Gamma,\Delta\vdash E_n:(\alpha_n,\beta_n)}{\Gamma,\Delta\vdash op(E_1,\ldots,E_n):(\alpha,\beta)}$$

$$\textit{where}\ (\alpha_1,\beta_1)\to\ldots\to(\alpha_n,\beta_n)\to(\alpha,\beta)\in\Delta(op)$$

Fig. 1. Type system for expressions

D. Command types

Figure 2 gives the typing rules for commands. As we shall see later, any program that is well-typed according to these rules satisfies non-interference properties as well as the ramification conditions of the ICC tiering discipline. The typing rules control information in way, which is related to Volpano, Irvine and Smith's system [14]. A *typing judgement* is of the form $\Gamma,\Delta\vdash C:(\alpha,\beta)$, and means that the command C is of type (α,β), where α and β are tiers of $\mathbb{SC}$, under Γ and Δ.

III. A TIMED BIG STEP SEMANTICS

We present a big step semantics, which takes the computation time into account. In an imperative language, a computation is a sequence of transition steps where at each step corresponds a store update. A transition performs a fixed number of basic operations or tests, which are counted as a unit cost. So, it is reasonable to define a time measure which corresponds to the number of transition steps.

Let $\mathbb{W}$ be the set of words over a finite alphabet. Expressions, commands and programs are interpreted over $\mathbb{W}$. We take two words $\mathtt{tt}$ and $\mathtt{ff}$ of $\mathbb{W}$ to denote respectively true and false. A store μ is a finite mapping from $\mathbb{V}$ to $\mathbb{W}$. We write $\mu[\vec{X}\leftarrow\vec{d}]$ for the store σ such that $\sigma(X_i)=d_i$ and $\sigma(Y)=\mu(Y)$ for $Y\neq X_i$, where $i=1,n$ for some fixed n. We write $\mathit{dom}(\mu)$ to mean the domain of μ.

Rules to evaluate expressions are given in Figure 3. Each operator of arity n is interpreted by a total function $[\![op]\!]:\mathbb{W}^n\mapsto\mathbb{W}$. The relation $\mu\vDash E\Rightarrow^t d$ means that the expression E is evaluated within t steps to $d\in\mathbb{W}$, where each variable in $\mathcal{V}(E)$ is in $\mathit{dom}(\mu)$. We just write $\mu\vDash E\Rightarrow\mu'$ if we don't care about the running time.

$$\textit{Variables}\quad \frac{}{\mu\vDash X\Rightarrow^1\mu(X)}$$

$$\textit{Op}\quad \frac{\mu\vDash E_1\Rightarrow^{t_1}d_1\quad\ldots\quad\mu\vDash E_n\Rightarrow^{t_n}d_n}{\mu\vDash op(E_1,\ldots,E_n)\Rightarrow^{1+\sum_{i=1}^n t_i}[\![op]\!](d_1,\ldots,d_n)}$$

Fig. 3. Time semantics of Expressions

Command execution rules are given in Figure 4. Given a store μ, the relation $\mu\vDash C\Rightarrow^t\mu'$ expresses that the command C returns the store μ' and terminates within t steps. We write $\mu\vDash C\Rightarrow\mu'$ if we don't care about the running time. A command C does not terminate in a given store μ, if there is no μ' and no t such that $\mu\vDash C\Rightarrow^t\mu'$. In this case, we write $\mu\vDash C\Rightarrow\bot$.

A program $\mathbf{P}$ is given by a command C, a list of input variables $X_1,\ldots,X_n$, and an output variable Y. A program computes a partial function $[\![\mathbf{P}]\!]$ from $\mathbb{W}^n$ to $\mathbb{W}$ defined by

$$[\![\mathbf{P}]\!](d_1,\ldots,d_n)=\mu(Y)\qquad\text{iff}\ \mu_0[\vec{X}\leftarrow\vec{d}]\vDash C\Rightarrow\mu$$

where for each variable Z in C, $\mu_0(Z)=\mathtt{ff}$. A program is terminating iff $[\![\mathbf{P}]\!]$ is a total function.

The program running time is the partial function $\mathit{Time}_{\mathbf{P}}$ from $\mathbb{W}^n$ to $\mathbb{N}$ defined by

$$\mathit{Time}_{\mathbf{P}}(d_1,\ldots,d_n)=t\qquad\text{iff}\ \mu_0[\vec{X}\leftarrow\vec{d}]\vDash C\Rightarrow^t\mu$$

The length of a word d is denoted $|d|$. A program $\mathbf{P}$ is running in polynomial time if there is a polynomial Q such that for all $d_1,\ldots,d_n$, $\mathit{Time}_{\mathbf{P}}(d_1,\ldots,d_n)\leq Q(\max_i(|d_i|))$.

IV. SAFE PROGRAMS

A. Neutral and positive operator interpretations

We present a restriction on operator interpretations. For this we shall define two kinds of operator interpretations called neutral and positive. But before, we present some typical operators with their interpretations over $\mathbb{W}$ to illustrate definitions.

- For each word v, we have an operator eq_v which tests whether or not a word begins with a given prefix v.
$$[\![eq_v]\!](u)=\begin{cases}\mathtt{tt} & \text{if } u=v.d \text{ for some } d\\ \mathtt{ff} & \text{otherwise}\end{cases}$$
- We have an operator pred, which deletes the first letter of a word such that
$$[\![\mathit{pred}]\!](u)=\begin{cases}\epsilon & \text{if } u=\epsilon\\ u & \text{if } u=i.d \text{ and for some } d \text{ and letter } i\end{cases}$$
- For each word v, we have an operator suc_v satisfying $[\![suc_v]\!](u)=v.u$.

Now, define $\trianglelefteq$ as the sub-word relation over $\mathbb{W}$ by $u\trianglelefteq d$, iff there are v and v' such that $d=v.u.v'$ where . is the concatenation.

An operator op has a *neutral interpretation* if either
1) either $[\![op]\!]:\mathbb{W}\to\{\mathtt{tt},\mathtt{ff}\}$ is a predicate; or
2) or for all $d_1,\ldots,d_n$, $[\![op]\!](d_1,\ldots,d_n)\trianglelefteq d_i$ for some $i\leq n$.

The operators eq_v and pred have a neutral interpretation.

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

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

The operator suc_v has a positive interpretation.

Remark 1: A neutral interpretation is also a positive interpretation. But the converse is false.

Assign

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

Compose

$$\frac{\Gamma, \Delta \vdash C : (\alpha, \beta) \qquad \Gamma, \Delta \vdash C' : (\alpha', \beta')}{\Gamma, \Delta \vdash C\ ;\ C' : (\alpha \vee \alpha', \beta \vee \beta')}$$

If

$$\frac{\Gamma, \Delta \vdash E : (\rho, \rho') \qquad \Gamma, \Delta \vdash C : (\alpha, \beta) \qquad \Gamma, \Delta \vdash C' : (\alpha, \beta)}{\Gamma, \Delta \vdash \texttt{if}\ E\ \texttt{then}\ C\ \texttt{else}\ C' : (\alpha, \beta)}\ \textit{where}\ \alpha \preceq \rho$$

While

$$\frac{\Gamma, \Delta \vdash E : (\alpha, \alpha') \qquad \Gamma, \Delta \vdash C : (\alpha, \beta)}{\Gamma, \Delta \vdash \texttt{while}(E)\{C\} : (\alpha, \beta)}\ \textit{where}\ \beta \prec \alpha$$

Fig. 2. Type system for commands

Update

$$\frac{\mu \vDash E \Rightarrow^{t} d}{\mu \vDash X{:=}E \Rightarrow^{t+1} \mu[X \leftarrow d]}$$

Branch

$$\frac{\mu \vDash E \Rightarrow^{t} \texttt{tt} \qquad \mu \vDash C \Rightarrow^{t'} \mu'}{\mu \vDash \texttt{if}\ E\ \texttt{then}\ C\ \texttt{else}\ C' \ \Rightarrow^{t+t'+1}\ \mu'} \qquad \frac{\mu \vDash E \Rightarrow^{t} \texttt{ff} \qquad \mu \vDash C' \Rightarrow^{t'} \mu''}{\mu \vDash \texttt{if}\ E\ \texttt{then}\ C\ \texttt{else}\ C' \ \Rightarrow^{t+t'+1}\ \mu''}$$

While

$$\frac{\mu \vDash E \Rightarrow^{t} \texttt{ff}}{\mu \vDash \texttt{while}(E)\{C\} \ \Rightarrow^{t}\ \mu} \qquad \frac{\mu \vDash E \Rightarrow^{t} \texttt{tt} \quad \mu \vDash C \ \Rightarrow^{t'}\ \mu' \quad \mu' \vDash \texttt{while}(E)\{C\} \ \Rightarrow^{t''}\ \mu''}{\mu \vDash \texttt{while}(E)\{C\} \ \Rightarrow^{t+t'+t''+1}\ \mu''}$$

Sequence

$$\frac{\mu \vDash C \Rightarrow^{t} \mu' \qquad \mu' \vDash C' \Rightarrow^{t'} \mu''}{\mu \vDash C\ ;\ C' \Rightarrow^{t+t'} \mu''}$$

Fig. 4. Timed semantics of Commands

B. Positive and neutral operators and safe environment

Assume that Δ is an operator typing environment. A constant is neutral if all its types the type w.r.t. Δ satisfy (α, β) where $\beta \preceq \alpha$. An $n+1$-ary operator is neutral if all its types w.r.t. Δ satisfy

$$(\alpha_1, \beta_1) \ldots \rightarrow (\alpha_n, \beta_n) \rightarrow (\wedge_{i=1,n}\alpha_i, \vee_{i=1,n}\beta_i)$$

where $\vee_{i=1,n}\beta_i \preceq \wedge_{i=1,n}\alpha_i$.

A n-ary operator is positive if all its types w.r.t. Δ satisfy

$$(\alpha_1, \beta_1) \rightarrow \ldots \rightarrow (\alpha_n, \beta_n) \rightarrow (\wedge_{i=1,n}\alpha_i, \wedge_{i=1,n}\alpha_i)$$

Next, if each operator *op* in the domain of Δ is neutral or positive then we say that Δ is a *safe* typing environment. Throughout this article, we consider safe operator typing environments.

Remark 2: Depending on an operator typing environment Δ, an operator may be neutral and positive at the same time.

Lemma 1 (Tier unicity): Assume that Γ is a variable typing environment and Δ is a safe operator typing environment. There is a unique tier α such that $\Gamma, \Delta \vdash E : (\alpha, \beta)$.

Proof: By induction on E. If E is a variable, the conclusion follows immediately. Suppose that $E = op(E_1, \ldots, E_n)$. Since Δ is safe, *op* is either neutral or positive. In either case, $\alpha = \wedge_{i=1,n}\alpha_i$, where the type of each E_i is (α_i, β_i). By induction hypothesis, α_i is unique. So α is also unique by lattice definition. ■

Remark 3: In contrast to this, β is not unique because it represents the current level of classification of an object, and so this level may vary depending on the context in which an expression is used. Typically, a variable of tier $\mathbf{1}$ can be an expression of type $(\mathbf{1}, \mathbf{0})$ or $(\mathbf{0}, \mathbf{0})$.

C. Main Result

Assume that Δ is a safe operator typing environment, and Γ a variable typing environment. A program $\mathbf{P}$ defined by a command C is *well-typed* if $\Gamma, \Delta \vdash C : (\alpha, \beta)$ where (α, β) is the type of $\mathbf{P}$.

We now define a relation between the operator interpretation and its types given by Δ. A program is *safe* if (i) it is well-typed, if (ii) each operator has a neutral or a positive interpretation and if (iii) each operator which has a neutral (positive) interpretation is neutral (positive) w.r.t. Δ.

Theorem 1: Let $(\{\mathbf{0}, \mathbf{1}\}, \preceq, \mathbf{0})$ be the complexity lattice. A terminating and safe program is computable in polynomial

time. Conversely, every polynomial time function over the set of words $\mathbb{W}$ is computable by a terminating and safe program.

Proof: If the type of a safe program over $(\{\mathbf{0},\mathbf{1}\},\preceq,\mathbf{0})$ is $(\mathbf{0},\mathbf{0})$ then the running time is constant because there is no loop. If its type is $(\mathbf{1},\mathbf{0})$, then we use Lemma 11. Finally, if its type is $(\mathbf{1},\mathbf{1})$, then we use Lemma 13.

The converse is a consequence of Theorem 2. ∎

D. Examples

We present three examples over the natural numbers. We posit that natural numbers are encoded by words in unary. The complexity lattice is $(\{\mathbf{0},\mathbf{1}\},\preceq,\mathbf{0})$. We consider a positive operator $+1$ in infix notation. Types of the operator $+1$ w.r.t. Δ are $(\mathbf{0},\mathbf{0})\to(\mathbf{0},\mathbf{0})$, $(\mathbf{1},\mathbf{0})\to(\mathbf{1},\mathbf{1})$, and $(\mathbf{1},\mathbf{1})\to(\mathbf{1},\mathbf{1})$. We also consider two neutral operators : -1 and a unary predicate >0, both in infix notation. Types w.r.t. Δ are $(\mathbf{0},\mathbf{0})\to(\mathbf{0},\mathbf{0})$, $(\mathbf{1},\mathbf{0})\to(\mathbf{1},\mathbf{0})$ and $(\mathbf{1},\mathbf{1})\to(\mathbf{1},\mathbf{1})$. Thus, the typing environment Δ is safe. So all programs below are safe.

The type of each command is written at the end of the line. We use labels for tiers. For example $X^{\mathbf{1}}$ means that the tier of X is $\mathbf{1}$.

1) Addition: Let us now examine the addition.

```
Add(X¹, Y⁰)
{while (X¹ > 0) {
          X¹:=X¹ − 1 : (1,0)
          Y⁰:=Y⁰ + 1 : (0,0)
 }   : (1,0)
```

The typing derivation is given in Figure 5. We see that the while-loop is controlled by X, which is of tier $\mathbf{1}$. The while-typing rule enforces the body-loop is of type $(\mathbf{1},\mathbf{0})$. Therefore, all commands inside the body-loop are of type $(\mathbf{1},\mathbf{0})$ or $(\mathbf{0},\mathbf{0})$. As a result, the variable Y must be of tier $\mathbf{0}$ and the operator $+1$ of type $(\mathbf{0},\mathbf{0})\to(\mathbf{0},\mathbf{0})$. The assignment $X^{\mathbf{1}}{:=}X^{\mathbf{1}}-1$ is a typical case of declassification. The variable X of tier $\mathbf{1}$ is first downgraded to an expression of type $(\mathbf{1},\mathbf{0})$. Then, we apply a neutral operator, here -1 with the type $(\mathbf{1},\mathbf{0})\to(\mathbf{1},\mathbf{0})$. We get a declassified expression $X-1$ of type $(\mathbf{1},\mathbf{0})$, that is assigned to X by a command of type $(\mathbf{1},\mathbf{0})$. One can intuitively see that Y can not be of tier β because in this case $Y^{\mathbf{0}}{:=}Y^{\mathbf{0}}{+}1$ should be of type $(\mathbf{1},\mathbf{1})$, which will violate the typing condition on while-loop.

2) Multiplication: Both inputs are of tier $\mathbf{1}$ and the output Z is of $\mathbf{0}$.

```
Mul(X¹, Y¹)
{  Z⁰:=0  : (0,0)
while  (X¹ > 0)
        {X¹:=X¹ − 1  : (1,0)
        U¹:=Y¹  : (1,0)
        while  (Y¹ > 0)
          {  Y¹:=Y¹ − 1  : (1,0)
             Z⁰:=Z⁰ + 1  : (0,0)
          }   : (1,0)
        Y¹:=U¹  : (1,0)
        }  : (1,0)
 }  : (1,0)
```

The typing derivation of $U^{\mathbf{1}}{:=}Y^{\mathbf{1}}$ is

$$\dfrac{\Gamma(U)=\mathbf{1} \qquad \dfrac{\Gamma(Y)=\mathbf{1}}{\Gamma,\Delta\vdash Y:(\mathbf{1},\mathbf{0})}}{\Gamma,\Delta\vdash U{:=}Y:(\mathbf{1},\mathbf{0})}$$

3) Greatest Common Divisor: The program below computes the greatest common divisor of X and Y. The result is stored in Z. For this, we need subtraction that we define as a neutral operator. Indeed, we have $[\![d-u]\!]\trianglelefteq d$ w.r.t. the unary encoding of natural numbers. We also need the predicate $X>Y$. We assign to both of them the neutral type $(\mathbf{1},\mathbf{0})\to(\mathbf{1},\mathbf{0})\to(\mathbf{1},\mathbf{0})$ in the typing derivation of $\mathbf{Gcd}$ presented below.

```
Gcd(X¹, Y¹)
{ if (X¹ > 0) then
{while (Y¹ > 0)
 {if (X¹ > Y¹)
          then X¹:=X¹ − Y¹  :(1,0)
         else Y¹:=Y¹ − X¹  :(1,0)
 }   :(1,0)
  Z¹:=X¹  :(1,0)
  }  :(1,0)
else Z¹:=Y¹  :(1,0)
}  :(1,0)
```

4) Search: The program below searches for v in a word X. For this, take two constants $\mathbf{tt}$ and $\mathbf{ff}$ to denote `tt` and `ff` of type $(\mathbf{1},\mathbf{0})$.

```
Search(X¹)
{ find:=ff;  :(1,0)
  loop:=tt;  :(1,0)
  while (loop)
  { if eq_v(X) then find:=tt; loop:=ff;
     else  if (X == ϵ) then loop:=ff;
     else   X:=pred(X); }
  }  :(1,0)
```

It is necessary that the tier of X is $\mathbf{1}$ in order to be able to modify the tier $\mathbf{1}$ variable `loop` which guards the while-loop. As a consequence, each command inside the two nested if-then-else is of type $(\mathbf{1},\mathbf{0})$.

V. Characterizing polynomial time functions

We now provide a simulation of polynomial time Turing machines by safe and terminating programs.

Theorem 2: Every polynomial time function over the set of words $\mathbb{W}$ can be computed by a safe and terminating program.

Proof: A unary polynomial time function f on $\mathbb{W}$ is computed by a Turing Machine M, with one tape and one head, within $(n+1)^k$ steps for some constant k where n is the input size. The tape of M is represented by two variables `Left` and `Right` which contains respectively the left side of the tape and the right side of the tape. States are encoded by fixed sized words and the variable `State` contains the current state. The tier of the three variables holding a configuration of M is $\mathbf{0}$. A one step transition is simulated by a finite cascade of if-commands of the form:

$$\dfrac{\dfrac{\Gamma(X)=\mathbf{1}}{\Gamma,\Delta\vdash X>0:(\mathbf{1},\mathbf{1})} \qquad \dfrac{\dfrac{\dfrac{\Gamma(X)=\mathbf{1}}{\Gamma,\Delta\vdash X:(\mathbf{1},\mathbf{0})}}{\Gamma,\Delta\vdash X-1:(\mathbf{1},\mathbf{0})}}{\Gamma,\Delta\vdash X{:=}X-1:(\mathbf{1},\mathbf{0})} \quad \dfrac{\dfrac{\dfrac{\Gamma(Y)=\mathbf{0}}{\Gamma,\Delta\vdash Y:(\mathbf{0},\mathbf{0})}}{\Gamma,\Delta\vdash Y+1:(\mathbf{0},\mathbf{0})}}{\Gamma,\Delta\vdash Y{:=}Y+1:(\mathbf{0},\mathbf{0})}}{\Gamma,\Delta\vdash \{X{:=}X-1;\ Y{:=}Y+1\}:(\mathbf{1},\mathbf{0})}}{\Gamma,\Delta\vdash \texttt{while}(X>0)\{X{:=}X-1;\ Y{:=}Y+1\}:(\mathbf{1},\mathbf{0})}$$

Fig. 5. Typing derivation of add

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

The command above 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 $(\mathbf{0},\mathbf{0})$, the type of the if-command is also $(\mathbf{0},\mathbf{0})$.

The iteration is made by nested k-while-loops. For this, we use k variables X1, ..., Xk of type $\mathbf{1}$.

```
X1^1:=X^1 // the initial value
while (X1^1 > 0)
      { X1^1:=X1^1 - 1 :(1,0)
        X2^1:=X^1 :(1,0)
      while (X2^1 > 0)
       { X2^1:=X2^1 - 1 :(1,0)
         X3^0:=X^1 :(1,0)
        ⋮
                } :(1,0)
 } :(1,0)
```

The guard of each loop is of tier $\mathbf{1}$. On the other hand, the body of each loop is of tier $(\mathbf{1},\mathbf{0})$ because first the step command is of type $(\mathbf{0},\mathbf{0})$ and second each variable of type $(\mathbf{1},\mathbf{0})$ is assigned neutral operators or variables of the same type. ■

VI. Type soundness for expressions

We begin with a lemma, which says that the type system for expressions is sound, with respect to declassification policy.

Lemma 2 (Expression declassification): Assume that Γ is a variable typing environment and that Δ is a safe operator typing environment. If $\Gamma,\Delta\vdash E:(\alpha,\beta)$, then $\beta\preceq\alpha$.

Proof: The proof goes by induction on expressions and uses the fact that operators are either neutral or positive. ■

The following lemma, called simple security, says that only variables at level α or higher will have their contents read in order to evaluate an expression E of type (α,β). Over the two tier lattice $(\{\mathbf{0},\mathbf{1}\},\preceq,\mathbf{0})$, if $\alpha=\mathbf{1}$, E is evaluated without reading any tier $\mathbf{0}$ variables. This property corresponds to the rule "no read down" of Biba's model. This reveals also some similarities with Myers' and Liskov's decentralized label model [30] where each object has an owner and a list of readers which this owner permits to read the data.

Lemma 3 (Simple security): Assume that Γ is a variable typing environment. Assume also that Δ is a safe operator typing environment. If $\Gamma,\Delta\vdash E:(\alpha,\beta)$, then for every $X\in\mathcal{V}(E)$, we have $\Gamma(X)\succeq\alpha$.

Proof: The proof is done by structural induction on E. The base case is when E is a variable. The typing judgement yields $\Gamma(X)\succeq\alpha$. Suppose that $E=op(E_1,\ldots,E_n)$. There are two cases to examine since Δ is safe.

- *op* is a neutral operator. Then its type is $(\alpha_1,\beta_1)\to\ldots\to(\alpha_n,\beta_n)\to(\wedge_{i=1,n}\alpha_i,\vee_{i=1,n}\beta_i)$. where $\beta=\vee_{i=1,n}\beta_i\preceq\alpha=\wedge_{i=1,n}\alpha_i$. By induction hypothesis, for each X in E_i we have $\Gamma(X)\succeq\alpha_i$. So, $\Gamma(X)\succeq\wedge_{i=1,n}\alpha_i$.
- *op* is positive and its type is $(\alpha_1,\beta_1)\to\ldots\to(\alpha_n,\beta_n)\to(\wedge_{i=1,n}\alpha_i,\wedge_{i=1,n}\alpha_i)$. By induction hypothesis, for each X in E_i we have $\Gamma(X)\succeq\alpha_i$. Again, $\Gamma(X)\succeq\wedge_{i=1,n}\alpha_i$.

■

VII. Type soundness for commands

The two following lemmas state properties on command types which are useful in other proofs.

Lemma 4 (Command declassification): Assume that Γ is a variable typing environment and that Δ is a safe operator typing environment. Assume also $\Gamma,\Delta\vdash C:(\alpha,\beta)$. Then $\beta\preceq\alpha$.

Proof: By Lemma 2 and by observing that if $\beta\preceq\alpha$ and $\beta'\preceq\alpha'$ then $\beta\vee\beta'\preceq\alpha\vee\alpha'$. ■

Lemma 5 (Monotonicity): Assume also that C' is a sub-command of C and that $\Gamma,\Delta\vdash C:(\alpha,\beta)$ where Δ is safe. Then, $\Gamma,\Delta\vdash C':(\alpha',\beta')$, where $\alpha'\preceq\alpha$ and $\beta'\preceq\beta$.

Proof: By induction on typing derivation length. ■

The following lemma corresponds to the write access rule of Biba's model, which states that a command can write data of lower level than itself. In our context, if a command C has the type (α,β) then Confinement lemma says no variable of rank above tier α is updated in C.

Lemma 6 (Confinement): Assume that Γ is a variable typing environment and that Δ is a safe operator typing environment. Assume also $\Gamma, \Delta \vdash C : (\alpha, \beta)$. Then, for each variable X assigned to in C, $\Gamma(X) \preceq \alpha$.

Proof: By induction on C. Suppose that $\Gamma, \Delta \vdash X{:=}E : (\alpha, \beta)$ by *(Assign)*. There is α' such that $\Gamma(X) = \alpha' \preceq \alpha$. The other cases follow directly by induction. ■

We are now ready to establish type soundness theorem, which states a first non-interference property. Over $(\{\mathbf{0}, \mathbf{1}\}, \preceq, \mathbf{0})$, type soundness theorem states that if X is variable of tier $\mathbf{1}$, then we can arbitrarily alter the value of a variable Y of tier $\mathbf{0}$, execute again the command C, and the value of X will be the same. As we shall see in a short while, this non-interference property has a strong consequence on the loop length.

Theorem 3 (type soundness): Assume that Γ is a variable typing environment and that Δ is a safe operator typing environment. Assume also that

1) $\Gamma, \Delta \vdash C : (\alpha, \beta)$
2) $\mu \vDash C \Rightarrow \mu'$,
3) $\sigma \vDash C \Rightarrow \sigma'$,
4) $dom(\mu) = dom(\sigma) = dom(\Gamma)$,
5) $\mu(X) = \sigma(X)$ for each variable X such that $\Gamma(X) \succeq \tau$.

Then for every variable X such that $\Gamma(X) \succeq \tau$, we have $\mu'(X) = \sigma'(X)$.

Proof: By induction on the structure of the derivation $\mu \vDash C \Rightarrow \mu'$.

Suppose that the last rule of the evaluation under μ is *(Update)* :

$$\frac{\mu \vDash E \Rightarrow d}{\mu \vDash X{:=}E \Rightarrow \mu[X \leftarrow d]}$$

and that the evaluation under σ ends with

$$\frac{\sigma \vDash E \Rightarrow d'}{\sigma \vDash X{:=}E \Rightarrow \sigma'[X \leftarrow d']}$$

The typing ends with an application of rule *(Assign)*

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

There are two cases to consider:

- $\alpha' \succeq \tau$. Simple security Lemma 3 states that for every $Y \in \mathcal{V}(E)$, we have $\Gamma(Y) \succeq \alpha$. So $\Gamma(Y) \succeq \alpha'$ and $\Gamma(Y) \succeq \tau$ by transitivity. From hypothesis 5, we have $\mu(Y) = \sigma(Y)$ for every variable Y in E. So $\mu \vDash E \Rightarrow d$ and $\sigma \vDash E \Rightarrow d$. Therefore, for every variable Y such that $\Gamma(Y) \succeq \tau$, $\mu'(Y) = \mu[X \leftarrow d](Y) = \sigma[X \leftarrow d](Y) = \sigma'(Y)$.
- $\alpha' \not\succeq \tau$. No variable Y such that $\Gamma(Y) \succeq \tau$ is modified. Therefore, we have $\mu'(Y) = \sigma'(Y)$, for all variables Y such that $\Gamma(Y) \succeq \tau$ by Hypothesis (5).

Suppose that $\mu \vDash \texttt{while}(E)\{C\} \ \Rightarrow^t \ \mu'$ and that $\sigma \vDash \texttt{while}(E)\{C\} \ \Rightarrow^{t'} \ \sigma'$ and the typing ends with

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

There are again two cases to consider.

- $\alpha \succeq \tau$. By simple security Lemma 3, for each variable $Y \in \mathcal{V}(E)$, $\Gamma(Y) \succeq \tau$. Therefore, $\mu \vDash E \Rightarrow d$ and $\sigma \vDash E \Rightarrow d$.
  The conclusion is immediate in the case $d = \texttt{ff}$. On the other case, $d = \texttt{tt}$, the evaluation under μ ends with
  $$\frac{\mu \vDash C \ \Rightarrow \mu'' \qquad \mu'' \vDash \texttt{while}(E)\{C\} \ \Rightarrow \ \mu'}{\mu \vDash \texttt{while}(E)\{C\} \ \Rightarrow \ \mu'}$$
  and the evaluation under σ ends with
  $$\frac{\sigma \vDash C \ \Rightarrow \ \sigma'' \qquad \sigma'' \vDash \texttt{while}(E)\{C\} \ \Rightarrow \ \sigma'}{\sigma \vDash \texttt{while}(E)\{C\} \ \Rightarrow \ \sigma'}$$
  By induction, $\mu''(X) = \sigma''(X)$ for all X such that $\Gamma(X) \succeq \tau$. So we can again apply the induction hypothesis. We have $\mu'(X) = \sigma'(X)$ for all X such that $\Gamma(X) \succeq \tau$.
- $\alpha \not\succeq \tau$. For every X assigned to in C, we have $\Gamma(X) \preceq \alpha$ by confinement Lemma 6. Since $\preceq$ is transitive, $\Gamma(X) \not\succeq \tau$. Therefore, no variable of type $\succeq \tau$ is updated during the execution of this loop.

The other cases are similar. ■

VIII. Upper bounds on loop length

We now move on the issue of bounding the length of while loops for terminating commands. Given a variable typing environment Γ and a safe operator typing environment Δ, a loop $\texttt{while}(E)\{C\}$ is of tier τ if $\Gamma, \Delta \vdash E : (\tau, \tau')$. Note that Lemma 1 implies that loop tiers are unique. Note also that the tier of each loop in C is less or equal than τ, because of Lemma 5.

We define a loop length semantics in Figure 6, which measures the length of loops of tier τ forgetting, intuitively, all loops of different tiers. The derivation $\mu \vdash_{\Gamma,\Delta} C \ \rightarrow^t_\tau \ \mu'$ means that the run of C under μ performs t iterations of tier τ. This loop length semantics is correct in two ways. First it is sound w.r.t. big step semantics, that is if $\mu \vDash C \ \Rightarrow^t \ \mu'$ then $\mu \vdash_{\Gamma,\Delta} C \ \rightarrow^{t_\tau}_\tau \ \mu'$ and $t_\tau \leq t$ for any tier τ. Second, it is correct with respect to time usage, which is established by Theorem 4 below.

Lemma 7: if $\mu \vDash E \ \Rightarrow^t \ d$, then $t \leq |E|$

Proof: Because the cost of an operator is 1. ■

Theorem 4: Let $(\mathbb{SC}, \preceq, \mathbf{0})$ be a complexity lattice. Assume that Γ is a variable typing environment and that Δ is a safe operator typing environment. Given a command C, there is a constant k such that if $\mu \vDash C \ \Rightarrow^t \ \mu'$ then $t \leq k \cdot T + k$ where $T = \sum_{\alpha \succ \mathbf{0}} t_\alpha$ and $\mu \vdash_{\Gamma,\Delta} C \ \rightarrow^{t_\alpha}_\alpha \ \mu'$.

Proof: Set $k = |C|$. By induction on the derivation $\mu \vDash C \Rightarrow^t \mu'$. ■

Temporal non-interference is a consequence of non-interference Theorem 3, which claims that values of tier $\mathbf{1}$ variables don't depend on values of lower tier variables. Temporal non-interference theorem says the length loop of tier τ depends only on variables of tier $\succeq \tau$. Over the two tier lattice $(\{\mathbf{0}, \mathbf{1}\}, \preceq, \mathbf{0})$, all loops are of tier $\mathbf{1}$. So, Temporal

$$\textit{Update}\ \frac{\mu \vDash E \Rightarrow d}{\mu \vdash_{\Gamma,\Delta} X{:=}E \rightarrow^0_\tau \mu[X \leftarrow d]} \qquad \textit{Sequence}\ \frac{\mu \vdash_{\Gamma,\Delta} C \rightarrow^t_\tau \mu' \qquad \mu' \vdash_{\Gamma,\Delta} C' \rightarrow^{t'}_\tau \mu''}{\mu \vdash_{\Gamma,\Delta} C\ ;\ C' \rightarrow^{t+t'}_\tau \mu''}$$

$$\textit{Branch}\ \frac{\mu \vDash E \Rightarrow d \qquad \mu \vdash_{\Gamma,\Delta} C_d \rightarrow^t_\tau \mu'}{\mu \vdash_{\Gamma,\Delta} \texttt{if}\ E\ \texttt{then}\ C_{\texttt{tt}}\ \texttt{else}\ C'_{\texttt{ff}} \rightarrow^t_\tau \mu'} \qquad \textit{While}\ \frac{\mu \vDash E \Rightarrow \texttt{ff}}{\mu \vdash_{\Gamma,\Delta} \texttt{while}(E)\{C\} \rightarrow^0_\tau \mu}$$

$$\frac{\mu \vDash E \Rightarrow \texttt{tt} \qquad \mu \vdash_{\Gamma,\Delta} C \rightarrow^{t'}_\tau \mu' \qquad \mu' \vdash_{\Gamma,\Delta} \texttt{while}(E)\{C\} \rightarrow^{t''}_\tau \mu''}{\mu \vdash_{\Gamma,\Delta} \texttt{while}(E)\{C\} \rightarrow^{t'+t''}_\tau \mu''}\ \Gamma,\Delta \vdash E : (\alpha,\alpha')\ \textit{and}\ \alpha \neq \tau$$

$$\frac{\mu \vDash E \Rightarrow \texttt{tt} \qquad \mu \vdash_{\Gamma,\Delta} C \rightarrow^{t'}_\tau \mu' \qquad \mu' \vdash_{\Gamma,\Delta} \texttt{while}(E)\{C\} \rightarrow^{t''}_\tau \mu''}{\mu \vdash_{\Gamma,\Delta} \texttt{while}(E)\{C\} \rightarrow^{1+t'+t''}_\tau \mu''}\ \Gamma,\Delta \vdash E : (\tau,\tau')$$

Fig. 6. Loop length semantics of tier τ

non-interference means that the length of the loop depends on tier $\mathbf{1}$ variable. A modification in the value of a variable of tier $\mathbf{0}$ does not affect loop lengths. Note that temporal non-interference is related to Lemma 4.1 of [4], which states that the computation length is bounded by a polynomial in the size of normal inputs.

Theorem 5 (Temporal non-interference): Assume that Γ is a variable typing environment and that Δ is a safe operator typing environment. Assume also that

1) $\Gamma,\Delta \vdash C : (\alpha,\beta)$
2) $\mu \vdash_{\Gamma,\Delta} C \rightarrow^t_\tau \mu'$,
3) $\sigma \vdash_{\Gamma,\Delta} C \rightarrow^{t'}_\tau \sigma'$,
4) $dom(\mu) = dom(\sigma) = dom(\Gamma)$,
5) $\mu(X) = \sigma(X)$ for every variable X such that $\Gamma(X) \succeq \tau$.

Then $t = t'$.

Proof: By induction on the structure of the derivation $\mu \vdash_{\Gamma,\Delta} C \rightarrow^t_\tau \mu'$.

Suppose that $\mu \vdash_{\Gamma,\Delta} \texttt{while}(E)\{C\} \rightarrow^t_\tau \mu'$ and that $\sigma \vdash_{\Gamma,\Delta} \texttt{while}(E)\{C\} \rightarrow^{t'}_\tau \sigma'$ and the typing ends with

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

There are three cases to consider.

- $\alpha \succeq \tau$. By simple security Lemma 3, for each variable $Y \in \mathcal{V}(E)$, $\Gamma(Y) \succeq \tau$. Therefore, $\mu \vDash E \Rightarrow d$ and $\sigma \vDash E \Rightarrow d$.
  In the case $d = \texttt{ff}$, we have $t = t' = 0$.
  On the other case, $d = \texttt{tt}$, the evaluation under μ ends with
  $$\frac{\mu \vdash_{\Gamma,\Delta} C \rightarrow^{t_1}_\tau \mu' \qquad \mu' \vdash_{\Gamma,\Delta} \texttt{while}(E)\{C\} \rightarrow^{t_2}_\tau \mu''}{\mu \vdash_{\Gamma,\Delta} \texttt{while}(E)\{C\} \rightarrow^{1+t_1+t_2}_\tau \mu''}$$
  and the evaluation under σ ends with
  $$\frac{\sigma \vdash_{\Gamma,\Delta} C \rightarrow^{t'_1}_\tau \sigma' \qquad \mu' \vdash_{\Gamma,\Delta} \texttt{while}(E)\{C\} \rightarrow^{t'_2}_\tau \sigma''}{\sigma \vdash_{\Gamma,\Delta} \texttt{while}(E)\{C\} \rightarrow^{1+t'_1+t'_2}_\tau \sigma''}$$
  Soundness Theorem 3 implies that for every variable X such that $\Gamma(X) \succeq \tau$, $\mu'(X) = \sigma'(X)$. So by induction hypothesis, $t_1 = t'_1$ and $t_2 = t'_2$. Therefore $t = 1 + t_1 + t_2 = 1 + t'_1 + t'_2 = t'$.
- $\alpha \not\succ \tau$. There is no loop of tier τ in C. So $t = t' = 0$.

Other cases are consequences of Soundness Theorem 3 ∎

Temporal non-interference provides a weak termination criterion. Take again the two tier lattice $(\{\mathbf{0},\mathbf{1}\}, \preceq, \mathbf{0})$. Theorem 5 says the length of a loop depends only on configurations of tier $\mathbf{1}$ variables. So, if we enter twice into a loop with the same configuration of variables of tier $\mathbf{1}$, we are sure that the loop will never terminate. Indeed, this configuration of tier $\mathbf{1}$ variables will never change. This argument is formalized in the next corollary.

Corollary 1: Assume that $\Gamma,\Delta \vdash E : (\alpha,\alpha')$ and $\Gamma,\Delta \vdash C : (\alpha,\beta)$ where Δ is a safe operator typing environment. Assume that $(\mu_i)_i$ is sequence satisying $\mu_i \vDash C \Rightarrow \mu_{i+1}$ such that there is $k > 0$ satisfying both conditions:

1) for each $i < k$, $\mu_i \vDash E \Rightarrow \texttt{tt}$.
2) for all $X \in \mathbb{V}(C)$, $\Gamma(X) \succeq \beta$, $\mu_0(X) = \mu_k(X)$.

Then, $\mu_0 \vDash \texttt{while}(E)\{C\} \Rightarrow \bot$.
In other words, $\texttt{while}(E)\{C\}$ does not terminate from μ_0.

Proof: Suppose for the sake of contradiction that there is T such that $\mu_0 \vDash \texttt{while}(E)\{C\} \Rightarrow^T \mu$. There are t and t' such that $\mu_0 \vdash \texttt{while}(E)\{C\} \rightarrow^t_\alpha \mu$ and $\mu_k \vdash \texttt{while}(E)\{C\} \rightarrow^{t'}_\alpha \mu$. Since $\alpha \succeq \beta$, Theorem 5 implies that $t = t'$. On the other hand, we have $t < t'$ since $k > 0$. This leads to a contradiction. So, $\texttt{while}(E)\{C\}$ does not terminate from μ_0. ∎

IX. Neutral terms

A neutral term is an expression which is just built up from variables and neutral operators.

Lemma 8: Assume that Γ is a variable typing environment. Assume also that Δ is a safe operator typing environment. If $\Gamma,\Delta \vdash E : (\alpha,\beta)$, where $\beta \prec \alpha$ then E is a neutral term.

Proof: We show by induction on E that if E contains a positive operator, then $\alpha = \beta$. As a result, the lemma will be

proved. Suppose that $E = op(E_1, \ldots, E_n)$. It is immediate if op is a positive operator. Otherwise, op is a neutral operator of type $(\alpha_1, \beta_1) \to \ldots \to (\wedge_{i=1,n}\alpha_i, \vee_{i=1,n}\beta_i)$, where $\vee_{i=1,n}\beta_i \preceq \wedge_{i=1,n}\alpha_i$ and which contains a sub-expression E_i with a positive operator. By induction hypothesis, we have $\alpha_i = \beta_i$. So, we have $\wedge_{i=1,n}\alpha_i \preceq \alpha_i = \beta_i \preceq \vee_{i=1,n}\beta_i$. Therefore, $\wedge_{i=1,n}\alpha_i = \vee_{i=1,n}\beta_i$. ■

Lemma 9 (Neutrality): Assume that Γ is a variable typing environment and that Δ is a safe operator typing environment. Assume also $\Gamma, \Delta \vdash C : (\alpha, \beta)$, where $\beta \prec \alpha$. Let X be a variable such that $\Gamma(X) \succ \beta$. Then, if X is assigned to in C, then it is always assigned to a neutral term.

Proof: Take an assignment of C where $\Gamma(X) \succ \beta$ and consider its typing rule:

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

By Lemma 5, we have $\beta' \preceq \beta$. So, $\Gamma(X) \succ \beta'$. Since $\Gamma(X) \preceq \alpha'$, we see that $\beta' \prec \alpha'$. Lemma 8 implies that E is a neutral term. ■

The last step is to determine an upper bound on the number of possible values of variables of tier strictly greater than β, which are computable by iterating a command C of type (α, β). Then it follows from Corollary 1 that this upper bound is also an upper bound on the length of $\texttt{while}(E)\{C\}$.

For this, we consider a command C and a store μ. We define the set $R^k_\mu(C)$ of reachable stores from C and μ in k steps by

$$R^0_\mu(C) = \{\mu\}$$
$$R^{k+1}_\mu(C) = \{\mu' \mid \sigma \vDash C \Rightarrow \mu' \textit{where } \sigma \in R^k_\mu(C)\}$$

and $R_C(\mu) = \cup_k R^k_\mu(C)$ as the set of reachable stores from C and μ.

Let Γ be a variable typing environment such that $\mathit{dom}(\Gamma) = \mathit{dom}(\mu)$. Define $\mu^{\uparrow\beta}$ as the restriction of μ to variables of tier strictly greater than β:

$$\mu^{\uparrow\beta}(X) = \begin{cases} \mu(X) & \Gamma(X) \succ \beta \\ \text{undefined} & \text{otherwise} \end{cases}$$

and then define, $R_C(\mu)^{\uparrow\beta} = \{\mu^{\uparrow\beta} \mid \mu \in R_C(\mu)\}$.

Lemma 10: Let Δ be a safe operator typing environment. Assume that $\mathbf{P}$ is a safe program given by a command C and that $\Gamma, \Delta \vdash C : (\alpha, \beta)$ where $\beta \prec \alpha$. Then, there is a polynomial Q such that for all stores μ, $\#R_C(\mu)^{\uparrow\beta} \leq Q(n)$where $n = \max_{\Gamma(X) \succ \beta}(|\mu(X)|)$. ($\#S$ is the cardinal of the set S.)

Proof: Let X be a variable assigned to in C such that $\Gamma(X) \succ \beta$. Lemma 9 states that X is assigned to a neutral term E. From this observation and because of the interpretation of neutral operators, we show by induction on k that for each μ' in $R^k_\mu(C)$ and each variable X, $\Gamma(X) \succ \beta$, $\mu'(X)$ is either in the range of some finite neutral interpretation (i.e. predicates), or there is Y such that $\mu'(X) \trianglelefteq \mu(Y)$ and $\Gamma(Y) \succ \beta$ by lemma 3.

Since the number of sub-words of a word of size n is bounded by n^2, $\#R_C(\mu)^{\uparrow\beta}$ is bounded by $Q(n)$ which only depends on C. ■

Thus, the length of a while-loop $\texttt{while}(E)\{C\}$ is bounded by the number of stores in $R_C(\mu)^{\uparrow\beta}$ where the type of C is (α, β).

X. Complexity bounds

Assume that $\mathbf{P}$ is a *safe* program given by a command C. We establish upper bounds on program runtime with respect to the program type.

Lemma 11: Let Δ be a safe operator typing environment. Assume that $\Gamma, \Delta \vdash C : (\alpha, \mathbf{0})$. There is a polynomial T such that for all stores μ, if $\mu \vDash C \Rightarrow^t \mu'$ then $t \leq T(n)$ where $n = \max_{\Gamma(X) \succ \mathbf{0}}(|\mu(X)|)$.

Proof: All while loops in C are of tier $\succ \mathbf{0}$. By Theorem 5, the length of loops depends only on variables of tier $\succ \mathbf{0}$. By setting $\beta = \mathbf{0}$ in Lemma 10 , we see that the number of reachable values of tier $\succ \mathbf{0}$ is bounded by $Q(n)$, where Q is a polynomial. And Corollary 1 states that length of loops is bounded by the number of reachable values of tier $\succ \mathbf{0}$. Therefore, the length of loops is bounded by $Q(n)$. By Theorem 4, the runtime is bounded by $T(n) = k \cdot Q(n) + k$, for some constant k. ■

Lemma 12: Let Δ be a safe operator typing environment. Assume that $\Gamma, \Delta \vdash C : (\alpha, \alpha)$. There is a constant c such that for all stores μ, if $\mu \vDash C \Rightarrow^t \mu'$ then for every X such that $\Gamma(X) = \alpha$, we have $|\mu'(X)| \leq \max_{\Gamma(Y) \succeq \alpha}(|\mu(Y)|) + c$.

Proof: Take an assignment $X{:=}E$ occurring in C, where $\Gamma(X) = \alpha$. There are two cases. First, the type of E is (α, α). So, $X{:=}E$ is not inside a loop body. Otherwise, the typing rule *(While)* would be violated. Second, the type of E is (α, β), where $\beta \prec \alpha$ by Lemma 2. By Lemma 8, E is a neutral term. So, simple security Lemma 3 implies that, in both cases, we have $|\mu'(X)| \leq \max_{\Gamma(Y) \succeq \alpha}(|\mu(Y)|) + c$ for some constant c. ■

Lemma 13: Assume that the complexity lattice is $(\{\mathbf{0}, \mathbf{1}\}, \preceq, \mathbf{0})$. Let Δ be a safe operator typing environment. Assume that $\Gamma, \Delta \vdash C : (\mathbf{1}, \mathbf{1})$. There is a polynomial T such that for all stores μ, if $\mu \vDash C \Rightarrow^t \mu'$ then $t \leq T(n)$ where $n = \max_{\Gamma(X) = \mathbf{1}}(|\mu(X)|)$.

Proof: By combining Lemmas 11 and 12. ■

XI. A criterion of weak termination

In our situation, non-interference property provides a termination criterion in the following sense:

Theorem 6: Let Δ be a safe operator typing environment. Assume that $\Gamma, \Delta \vdash C : (\alpha, \mathbf{0})$. There is a polynomial decision procedure which given a store μ decides whether or not there is μ' such that $\mu \vDash C \Rightarrow \mu'$.

Proof: Corollary 1 implies that it suffices to check if there are two stores whose values assigned to variables of tier $\succ \mathbf{0}$ are identical. The runtime of the decision procedure is polynomial time computable because Lemma 10 implies that it suffices to memorize a polynomial number of stores. ■

XII. Conclusion

We think that concepts underlying type systems for secure information flow analysis provide new directions to characterize program complexity. There are several issues to investigate like the ones that are listed below.

A major question is intensionality and language expressiveness, which is crucial in order to apply methods related from ICC. Security typed languages are now expressive and we could maybe take advantage of works already made in order to analyze resource usage of a broader class of sequential or parallel algorithms. A first step would be to enlarge the class of neutral terms by, for example, re-using des-allocated spaces, like this has been done by Hofmann [9].

The role of complexity lattice was reduced in this paper because our study is on polynomial time computation, which is a robust sequential class. On the other hand, complexity lattice may be an interesting tool to study small complexity classes and to to study the complexity of process calculus computation with respect to causality see e.g. [31].

We restrict the domain of values to a set of words. Results may be extended easily for several data-structure like array of words, lists, ... However to go further, it should be interesting to follow Dal Lago & Martini [32] and to deal with more complex data structures. Another direction is also to study the introduction of weak operators like *parity* which declassify (or classify) a small amount of information.

A practical approach may combine other approaches that we have already mentioned in the discussion on related works. In particular, a static analysis based on our approach could be made by compiling a source program into an abstract program with respect to resource consumption. Here, we follow ideas suggested in [20] and [23]. Then, the complexity of a source program could be obtained from a type inference algorithms.

Conversely, we may wonder if methods developed in ICC, and in particular light linear logic approaches, can be used and bring something new to security-type systems.

Acknowledgment

The author would like to thank Gérard Boudol and Daniel Leivant.

References

[1] A. Cobham, "The intrinsic computational difficulty of functions," in *International Conference on Logic, Methodology, and Philosophy of Science*. North, 1964, pp. 24–30.

[2] N. Immerman, "Relational queries computable in polynomial time," *Information and Control*, vol. 68, no. 1-3, pp. 86–104, 1986.

[3] D. Leivant, "Predicative recurrence and computational complexity i: Word recurrence and poly-time," in *Feasible Mathematrics II*, P. Clote and J. Remmel, Eds., 1994.

[4] S. Bellantoni and S. Cook, "A new recursion-theoretic characterization of the poly-time functions," *Computational Complexity*, vol. 2, pp. 97–110, 1992.

[5] D. Leivant, "A foundational delineation of poly-time," *Inf. Comput.*, vol. 110, no. 2, pp. 391–420, 1994.

[6] D. Leivant and J.-Y. Marion, "Lambda calculus characterizations of poly-time," *Fundam. Inform.*, vol. 19, no. 1/2, pp. 167–184, 1993.

[7] J.-Y. Girard, "Light linear logic," *Inf. Comput.*, vol. 143, no. 2, pp. 175–204, 1998.

[8] Y. Lafont, "Soft linear logic and polynomial time," *Theor. Comput. Sci.*, vol. 318, no. 1-2, pp. 163–180, 2004.

[9] M. Hofmann, "Linear types and non-size-increasing polynomial time computation," *Inf. Comput.*, vol. 183, no. 1, pp. 57–85, 2003.

[10] J.-Y. Marion, "Actual arithmetic and feasibility," in *CSL*, ser. Lecture Notes in Computer Science, vol. 2142. Springer, 2001, pp. 115–129.

[11] D. E. Bell and L. L. Padula, "Secure computer system: unified exposition and multics interpretation," Mitre corp Rep., Tech. Rep., 1976.

[12] K. Biba, "Integrity considerations for secure computer systems," Mitre corp Rep., Tech. Rep., 1977.

[13] J. Goguen and J. Meseguer, "Security policies and security models," in *IEEE Symposium on Security and Privacy*, vol. 13, 1982, pp. 11–20.

[14] D. Volpano, C. Irvine, and G. Smith, "A sound type system for secure flow analysis," *Journal of Computer Security*, vol. 4, no. 2/3, pp. 167–188, 1996.

[15] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE J. Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, Jan. 2003.

[16] A. Sabelfeld and D. Sands, "Declassification: Dimensions and principles," *J. Comput. Secur.*, vol. 17, pp. 517–548, October 2009.

[17] N. D. Jones, "The expressive power of higher-order types or, life without cons," *J. Funct. Program.*, vol. 11, no. 1, pp. 5–94, 2001.

[18] N. D. Jones and L.Kristiansen, "A flow calculus of *mwp*-bounds for complexity analysis," *ACM Trans. Comput. Log.*, vol. 10, no. 4, 2009.

[19] K.-H. Niggl and H. Wunderlich, "Certifying polynomial time and linear/polynomial space for imperative programs," *SIAM J. Comput.*, vol. 35, no. 5, pp. 1122–1147, 2006.

[20] A. Ben-Amram, N. D. Jones, and L. Kristiansen, "Linear, polynomial or exponential? complexity inference in polynomial time," in *Logic and Theory of Algorithms, Fourth Conference on Computability in Europe, CiE*, ser. LNCS, Springer, Ed., vol. 5028, 2008, pp. 67–76.

[21] J.-Y. Marion and R. Péchoux, "Analyzing the implicit computational complexity of object-oriented programs," in *FSTTCS*, ser. LIPIcs, vol. 2. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2008, pp. 316–327.

[22] J.-Y. Marion and J.-Y. Moyen, "Efficient first order functional program interpreter with time bound certifications," in *LPAR*, ser. Lecture Notes in Computer Science, vol. 1955, 2000, pp. 25–42.

[23] S. Jost, K. Hammond, H.-W. Loidl, and M. Hofmann, "Static determination of quantitative resource usage for higher-order programs," in *POPL*, 2010, pp. 223–236.

[24] J. Hughes, L. Pareto, and A. Sabry, "Proving the correctness of reactive systems using sized types," in *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '96. New York, NY, USA: ACM, 1996, pp. 410–423. [Online]. Available: http://doi.acm.org/10.1145/237721.240882

[25] E. Albert, D. Alonso, P. Arenas, S. Genaim, and G. Puebla, "Asymptotic resource usage bounds," in *Proceedings of the 7th Asian Symposium on Programming Languages and Systems*, ser. APLAS '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 294–310. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-10672-9_21

[26] S. Gulwani, K. K. Mehra, and T. M. Chilimbi, "Speed: precise and efficient static estimation of program computational complexity," in *POPL*, 2009, pp. 127–139.

[27] D. Leivant, "A foundational delineation of computational feasiblity," in *Proceedings of the Sixth IEEE Symposium on Logic in Computer Science (LICS'91)*, 1991.

[28] H. Simmons, "The realm of primitive recursion," *Archive for Mathematical Logic*, vol. 27, pp. 177–188, 1988.

[29] J.-Y. Marion, "On tiered small jump operators," *Logical Methods in Computer Science*, vol. 5, no. 1, 2009.

[30] A. Myers and B. Liskov, "Protecting privacy using the decentralized label model," *ACM Trans. Softw. Eng. Methodol.*, vol. 9, no. 4, pp. 410–442, 2000.

[31] R. Demangeon, D. Hirschkoff, and D. Sangiorgi, "Static and dynamic typing for the termination of mobile processes," in *IFIP TCS*, 2008, pp. 413–427.

[32] U. Dal Lago and S. Martini, "An invariant cost model for the lambda calculus," in *Logical Approaches to Computational Barriers, Second Conference on Computability in Europe*, ser. Lecture Notes in Computer Science, B. L. Arnold Beckmann, Ulrich Berger and J. Tucker, Eds., vol. 3988. Springer, 2006, pp. 105–114.