Type-based heap and stack space analysis in Java

Emmanuel Hainry, Romain Péchoux

Fighting against "Out Of Memory" and "Stack Overflow" errors in Java

Emmanuel Beautiful Hainry and Romain Péchoux

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

Abstract. We fight.

Keywords: Secure Information Flow, Type system, Secured resource consumption

1 Introduction

The main intuition is as follows. The heap is represented in term of a graph structure where nodes are object adresses and edges relate an object address to its attribute addresses. The type system splits variables in two universes, tier **0** universe and tier **1** universe. Whereas tier **1** variables are pointers to nodes of the initial heap, tier **0** variables may point to newly created addresses. The information may flow from tier **1** to tier **0**, that is a variable that never points to newly created nodes can do it. However our type system precludes flows from **0** to **1**. Indeed once a variable has stored a newly created instance, it has escaped from the initial heap. Naively, tier **1** variables are the one that can be used either as guards of a while loop or as a recursive argument in a method call. If such a variable falls to tier **0** then it will no longer be used in these ways. Indeed tier **0** variables are just used as storages for computed data. Since the initial graph structure has size bounded linearly by the input size, that is $O(n)$, the number of distinct possible configurations for k tier **1** variables is at most $O(n^k)$. Consequently, we know that a terminating program will stop in a polynomial number of steps, based on the assumption that loops and recursive calls are only controlled by tier **1** variables.

2 Core Java

In this section, we introduce the syntax and semantics of the considered core Java language.

2.1 Well-formed programs

Expressions, instructions, constructors, methods and classes are defined by the following grammar:

$$
\begin{array}{lll}
\text{Expressions} \ni E & ::= & X \mid \texttt{null} \mid \texttt{this} \mid \texttt{true} \mid \texttt{false} \mid op(E_1, \ldots, E_n) \mid \\
& & \texttt{new } \mathrm{C}(E_1, \ldots, E_n) \mid E.m(E_1, \ldots, E_n) \\
\text{Instructions} \ni I & ::= & X{:=}E; \mid I_1\ I_2 \mid E.m(E_1, \ldots, E_n); \mid \\
& & \texttt{while } E \texttt{ do}\{I\} \mid \texttt{if } E \texttt{ then } I \texttt{ else } J \\
\text{Methods} \ni M_{\mathrm{C}} & ::= & \tau\ m(\tau_1\ X_1, \ldots, \tau_n\ X_n)\{I \texttt{ return } X;\} \\
& & \mid \texttt{void}\ m(\tau_1\ X_1, \ldots, \tau_n\ X_n)\{I\} \\
\text{Constructors} \ni K_{\mathrm{C}} & ::= & \mathrm{C}(\tau_1 X_1, \ldots, \tau_n\ X_n)\{\texttt{this}.X_1 := X_1; \ldots \texttt{this}.X_n = X_n;\} \\
\text{Classes} \ni \mathrm{Class} & ::= & \mathrm{C}\{\tau_1\ X_1; \ldots; \tau_n\ X_n;\ K_{\mathrm{C}}\ M_{\mathrm{C}}^1 \ldots M_{\mathrm{C}}^k\} \\
\text{Executables} \ni \mathrm{Exe} & ::= & \mathrm{Exe}\{\texttt{main}()\{\tau_1\ X_1 = E_1; \ldots; \tau_n\ X_n = E_n; I\}\}
\end{array}
$$

where $\texttt{this}.X, X \in \mathbb{V}$, $op \in \mathbb{O}, \mathrm{C} \in \mathbb{C}, m \in \mathbb{M}$, $\mathbb{V}$ being the set of variables, $\mathbb{O}$ the set of operators, $\mathbb{M}$ the set of method names and $\mathbb{C}$ the set of class names. The τs are type annotations ranging over $\mathbb{C} \cup \{\texttt{void}, \texttt{boolean}\}$. A program is a collection of classes defined by the above grammar together with exactly one executable class. In what follows, we will only consider well-formed programs, that is programs such that:

- Each class name C appearing in the collection of classes corresponds to exactly one class of name C within the collection.
- Each method name m appearing in the collection of classes is defined in at least one of the classes.
- The method `main` is never called.

Comments: Note that, for readability, we have restricted classes to have at most one constructor. Moreover the only primitive data considered in expressions are boolean values `true` and `false`. There are no subclasses and, consequently, no overrides. However overload is possible.

2.2 Semantics

2.3 Tiered types and environments

Types The set of types is defined to be the set including a type for each class Class and the two primitive types `void` and `boolean`: $\mathtt{T} = \{\texttt{void}, \texttt{boolean}\} \cup \mathbb{C}$. For simplicity, we have only considered boolean values but all the other Java primitive types such as floats, integers and characters could be considered without any restriction.

Tiers *Tiers* are two elements of the lattice $(\{\mathbf{0}, \mathbf{1}\}, \vee, \wedge)$ where $\wedge$ and $\vee$ are the greatest lower bound operator and the least upper bound operator, respectively. The induced order, denoted $\preceq$, is such that $\mathbf{0} \preceq \mathbf{1}$. In what follows, let $\alpha, \beta, \ldots$ denote tiers in $\{\mathbf{0}, \mathbf{1}\}$.

Tiered types A tiered type is a pair consisting of a type $\tau \in \mathtt{T}$ together with a tier $\alpha \in \{\mathbf{0}, \mathbf{1}\}$ and will be denoted by $\tau(\alpha)$.

Environments A *variable typing environment* Γ maps each variable in $\mathbb{V}$ to a tiered type. Intuitively, each tier variable has a specific role to play: while tier **1** variables will be used as guards of while loops - Consequently, they should not be allowed to take more than a polynomial number of distinct values, tier **0** variables may increase and cannot be used as while loop guards - they play the role of a store.

Operator types σ are generated using the grammar:

$$\sigma ::= \tau(\alpha) \mid \tau(\alpha) \longrightarrow \sigma$$

where $\longrightarrow$ is right associative as usual. An *operator typing environment* Δ maps each operator *op* to a set of operator types $\Delta(op)$, where the operator types corresponding to an operator of arity n are of the shape $\tau_1(\alpha_1) \longrightarrow \ldots \longrightarrow \tau_n(\alpha_n) \longrightarrow \tau(\alpha)$.

2.4 Type system

For simplicity, we will suppose that the considered programs are transformed up-to α-conversion so that each variable (local variable, attribute or parameter) has a distinct name (except for constructor parameters).

Tiered types will be used to type both expressions and commands. Given a variable typing environment Γ and an operator typing environment Δ, typing judgments are of the shape:

- $\Gamma, \Delta \vdash D : \sigma$, if D is an expression or an instruction, i.e. $D \in \text{Expressions} \cup \text{Instructions}$
- $\Gamma, \Delta \vdash f : \tau_1(\alpha_1) \longrightarrow \ldots \longrightarrow \tau_n(\alpha_n) \longrightarrow \tau(\alpha)$, if f is a method or a constructor of arity n, i.e. $f \in \text{Constructors} \cup \text{Methods}$
- $\Gamma, \Delta \vdash S : \diamond$, if S is a set of class and executable classes, i.e. $S \in \mathcal{P}(\text{Classes} \cup \text{Executables})$

The two first judgments provide the type of the considered expression, instruction, method or constructor wrt the considered environments whereas the last judgment means that a set of classes is well-typed (denoted $\diamond$) under the considered environments. The corresponding typing rules are provided in Figure 1. There are some important points to explain in this type system. First, the typing discipline precludes values from flowing from tier α to tier β, whenever $\alpha \preceq \beta$. Consequently, the guards of while loops are enforced to be of tier **1** in rule (CW). Moreover, in a (CB) rule, we enforce the tier of the guard to be equal to the tier of both branches. Also note that the subtyping rule (CSub) is restricted to commands in order not to break this preclusion. On the opposite, information may flow from tier **1** to tier **0** then to tier $-\mathbf{1}$. This point is underlined by the side condition of the (CA) rule. The (F) rule enforces the tier of the variable storing the process id

$$\frac{}{\Gamma, \Delta \vdash \texttt{true} : \texttt{boolean}(\alpha)}\ (T) \qquad \frac{}{\Gamma, \Delta \vdash \texttt{false} : \texttt{boolean}(\alpha)}\ (F)$$

$$\frac{}{\Gamma, \Delta \vdash \texttt{null} : \texttt{void}(\alpha)}\ (N) \qquad \frac{\Gamma(X) = \tau(\alpha) \quad \nu \in \{\emptyset; \tau\}}{\Gamma, \Delta \vdash \nu\ X : \tau(\alpha)}\ (V)$$

$$\frac{\forall i,\ \Gamma, \Delta \vdash E_i : \tau_i(\alpha_i) \qquad \tau_1(\alpha_1) \times \ldots \times \tau_n(\alpha_n) \longrightarrow \tau(\alpha) \in \Delta(op)}{\Gamma, \Delta \vdash op(E_1, \ldots, E_n) : \tau(\alpha)}\ (O)$$

$$\frac{\forall i,\ \Gamma, \Delta \vdash E_i : \tau_i(\alpha_i) \quad \Gamma, \Delta \vdash K_{\mathrm{C}} : \tau_1(\alpha_1) \times \ldots \times \tau_n(\alpha_n) \longrightarrow \mathrm{C}(\alpha)}{\Gamma, \Delta \vdash \texttt{new}\ \mathrm{C}(E_1, \ldots, E_n) : \mathrm{C}(\alpha)}\ (K_C)$$

$$\frac{\Gamma, \Delta \vdash m_{\mathrm{C}} : \tau_1(\alpha_1) \times \ldots \times \tau_n(\alpha_n) \longrightarrow \tau(\alpha) \quad \Gamma, \Delta \vdash E : \mathrm{C}(\beta) \quad \forall i,\ \Gamma, \Delta \vdash E_i : \tau_i(\alpha_i)}{\Gamma, \Delta \vdash E.m(E_1, \ldots, E_n) : \tau(\alpha)}\ (m_C)$$

$$\frac{\Gamma, \Delta \vdash \nu\ X : \tau(\alpha) \qquad \Gamma, \Delta \vdash E : \tau(\beta)}{\Gamma, \Delta \vdash \nu\ X{:=}E : \texttt{void}(\alpha)}\ \alpha \preceq \beta\ \ (A) \qquad \frac{\Gamma, \Delta \vdash I : \alpha}{\Gamma, \Delta \vdash I : \beta}\ \alpha \preceq \beta\ \ (S)$$

$$\frac{\forall i,\ \Gamma, \Delta \vdash I_i : \texttt{void}(\alpha_i)}{\Gamma, \Delta \vdash I_1\ I_2 : \texttt{void}(\alpha_1 \vee \alpha_2)}\ (I) \qquad \frac{\Gamma, \Delta \vdash E : \texttt{boolean}(\mathbf{1}) \quad \Gamma, \Delta \vdash I : \texttt{void}(\alpha)}{\Gamma, \Delta \vdash \texttt{while}(E)\texttt{do}\{I\} : \texttt{void}(\mathbf{1})}\ (W)$$

$$\frac{\Gamma, \Delta \vdash E : \texttt{boolean}(\alpha) \quad \forall i,\ \Gamma, \Delta \vdash I_i : \texttt{void}(\alpha)}{\Gamma, \Delta \vdash \texttt{if}\ E\ \texttt{then}\ I_1\ \texttt{else}\ I_2 : \texttt{void}(\alpha)}\ (B)$$

$$\frac{\forall \mathrm{Class} \in S,\ \Gamma, \Delta \vdash \mathrm{Class} : \diamond}{\Gamma, \Delta \vdash \mathrm{Class} : \diamond}\ (C)$$

$$\frac{\Gamma, \Delta \vdash \tau_1\ X_1 = E_1; \ldots; \tau_n\ X_n = E_n; I : \texttt{void}(\mathbf{1})}{\Gamma, \Delta \vdash \mathrm{Exe}\{\texttt{main}()\{\tau_1\ X_1 = E_1; \ldots; \tau_n\ X_n = E_n; I\}\} : \diamond}\ (P)$$

Fig. 1. Type system for core Java programs

to be of tier **0** since the value stored will increase dynamically during the process execution. Finally, the tier of the variable storing the result returned by a child process (rule (W)) has to be of tier $-\mathbf{1}$, which means that no information may flow from a variable of a child process to tier **0** and tier **1** variables of its parent process.

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

Note that subtyping is useless for expressions because of the (A) rule.

2.5 Semantics of expressions, configurations and environments.

Domain. Let $\mathbb{W}$ be the set of words over a finite alphabet Σ including two symbols `true` and `false` that denote truth values true and false. Let ε be the empty word. The length of a word d is denoted $|d|$. As usual, we set $|\varepsilon| = 0$. Define $\trianglelefteq$ as the sub-word relation over $\mathbb{W}$, by $v \trianglelefteq w$, iff there are u and u' of

$\mathbb{W}$ s.t. $w = u.v.u'$, where . is the concatenation. We write $\underline{n}$ to mean the binary word encoding the natural number n.

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

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

Expressions and configurations. Each operator of arity n is interpreted by a total function $[\![op]\!] : \mathbb{W}^n \mapsto \mathbb{W}$. The expression evaluation relation $\xrightarrow{\mathsf{e}}$ and the sequential command evaluation relation are $\xrightarrow{\mathsf{c}}$ are described in Figure 2.

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

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

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

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

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

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

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

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

Fig. 2. Small step semantics of expressions and configurations

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

Semantics. The transition $\rightarrow$ for process evaluation is provided in Figure 3. The (Fork) rule creates a new configuration, a new process, say of id n, with a new store, and adds it to the environment. The parent process records its child id $\underline{n}$

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

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

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

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

Fig. 3. Semantics of Environments

2.6 Strong normalization, lock-freedom and confluence

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

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

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

3 Safe processes, type inference and complexity

3.1 Neutral, max, and positive operators

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

Definition 1.

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

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

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

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

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

3.2 Main result

4 Conclusion

We conclude.