INFOM227 : Analyse de programmes pour la cybersécurité

Travail individuel : Interprétation abstraite

Johan Rochet

Université de Namur
Faculté d'informatique
Année académique 2023-2024

Table des matières

1 Description sommaire de l'analyseur

La concurrence dans les langages de programmation peut mener à beaucoup de problèmes, que ce soit les problématiques de deadlock, starvation ou bien celles de race conditions. Ces problèmes peuvent survenir lors de l'exécution en parallèle de plusieurs threads et mener à une exécution inadaptée d'un programme : boucle à l'infini, comportement non voulu. Dans ce rapport, un analyseur pour détecter les race conditions au sein d'un programme minimal mettant en place de la concurrence est présenté.

Les race conditions (situation de course) surviennent lorsque deux ou plusieurs threads tentent d'accéder et de modifier une ressource partagée. Le résultat de l'exécution du programme dépend ainsi de la race (course) entre les threads et peut mener à des cas de données corrompues, des comportements indésirables, voire, à des crashs du programme. Les principaux cas de race conditions sont les deux suivants :

1. Plusieurs threads tentent de modifier la valeur de la même ressource partagée.
2. Un thread A tente de lire la valeur d'une ressource tandis qu'un autre thread B tente de la modifier.

[1] Les accès concurrents en lecture ne sont pas considérés comme des race conditions, car ils ne modifient pas l'espace mémoire. Un accès en lecture réalisé simultanément avec une modification peut mener à des valeurs indéterminées dans le résultat de la lecture de la variable. Pour éviter ce genre de situation, l'utilisation de verrou (mutex), permet de rendre atomique toute opération sur une variable partagée au sein d'un thread.

L'analyseur statique ici présenté fonctionne en 3 grandes étapes :

1. Vérifier que le programme en entrée respecte la sémantique opérationnelle ;
2. Générer le CFG (Control Flow Graph) de tous les threads du programme à analyser. L'idée est de voir chaque thread comme un programme à part entière s'entremêlant avec une partie des instructions du programme principal ;
3. Réaliser une analyse statique des variables partagées pour détecter la présence potentielle ou non de race conditions.

L'analyseur ici proposé est basé sur une implémentation en JAVA sur base du framework ANTLR permettant de définir aisément la grammaire d'un langage, de générer un arbre syntaxique abstrait d'un programme syntaxiquement correct du langage et de fournir une implémentation par défaut d'un visiteur de cet arbre. L'analyse va ainsi, dans un premier temps, chercher à prouver la justesse sémantique du programme en entrée avec un premier visiteur. Ce premier visiteur est en fait une implémentation concrète de la sémantique opérationnelle présentée dans ce rapport, mis en place pour vérifier que les programmes en entrée respectent celle-ci. Analyser la présence de race conditions dans un programme qui ne serait pas sémantiquement correct n'a en effet pas beaucoup de sens. Ensuite, un deuxième visiteur générera le Control Flow Graph associé à l'AST du programme en entrée. Finalement, un dernier visiteur implémentera l'algorithme de détection des race conditions en récupérant les variables modifiées ou accédées en lecture par un thread et en écriture par un autre.

L'analyseur se base sur une extension du langage Small (**SmallConcurrency**) de manière à lui permettre de mettre en place la gestion d'exécution concurrente et d'accès à des variables partagées. L'ensemble de la logique mathématique est inspirée de la logique présentée dans le syllabus du cours [2].

Dans la suite, nous présenterons dans un premier temps la nouvelle syntaxe et sémantique du **SmallConcurrency**. Ensuite, nous détaillerons, de manière formelle, l'analyse dataflow mise en place, le domaine abstrait, la lattice, les fonctions de concrétisation et d'abstraction ainsi que certaines fonctions de flux. Par après, nous commenterons la solution proposée en analysant ses propriétés (soundness, forward/backward, warnings & erreurs renvoyées). Nous tenterons ensuite d'exprimer l'analyse effectué en utilisant le framework GEN/KILL pour finalement présenter un petit exemple d'exécution de l'analyseur.

2 Syntaxe et sémantique de SmallConcurrency

2.1 Syntaxe

La syntaxe originelle de Small ne permettait pas de prendre en compte des aspects de concurrence et se limitait à des programmes purement séquentielles. Afin de permettre au langage de prendre en compte des aspects de concurrence, plusieurs nouvelles constructions syntaxiques ont été mises en place.

2.1.1 Les threads

Modéliser la concurrence au sein d'un programme est rendu possible dans de nombreux langages de programmations via l'utilisation de *threads*. En SmallConcurrecny, définir un thread est assez simple et se fait avec la syntaxe suivante :

$$< threadDecl > ::= thread < sequence >$$

Un thread est ainsi déclaré et lancé en même temps par une instruction simple reprenant un bloc d'instructions à exécuter en parallèle. Le choix a été fait de ne pas mettre en place de mécanisme pour attendre la terminaison de threads.

2.1.2 Déclaration de variables, variables partagées et verrous

Afin de pouvoir modéliser l'accès à des variables partagées par différents threads, il a été nécessaire dans un premier temps de mettre en place des instructions permettant de déclarer des variables partagées/globales. Par souci d'uniformité du langage, une telle déclaration de variables a également été mise en place pour les variables locales. **Seules** les variables globales seront accessibles à l'intérieur d'un thread (à l'exception des variables locales qu'il définit bien évidemment). Ainsi, les instructions suivantes sont rajoutées à la syntaxe :

$$< globalVarDecl > ::= gVar < ID >$$

$$< localVarDecl > ::= var < ID >$$

De plus, il est rendu possible à un thread d'acquérir le verrou sur une variable globale du programme et de le relâcher par la suite, ce qui est modélisé par la syntaxe suivante.

$$< lockVarDecl > ::= lock < ID >$$

$$< unlockVarDecl > ::= unlock < ID >$$

2.1.3 Modification de la syntaxe originelle

Par volonté de simplicité, et comme la valeur prise par les variables au sein du programme avait peu d'importance pour l'analyse mise en place, la syntaxe originelle a été modifiée pour ne plus permettre aux variables d'être de type booléens, mais uniquement de type entier. Les expressions booléennes restent cependant parties intégrantes du programme, mais ne pourront pas être stockées dans des variables. Dans la même ligne d'idée, la valeur de retour d'une fonction devra, dans tous les cas, être une valeur arithmétique.

Le reste de la syntaxe de Small reste inchangée avec toutefois, l'ajout de nouvelles instructions définies ci-dessus dans la syntaxe des *statements* (*varDecl*, *lockVarDecl*, *unlockVarDecl* et *threadDecl*) et de la racine du programme (les variables globales sont supposées être déclarées avant les fonctions du programme). L'intégralité de la syntaxe de SmallConcureency est disponible sur le *Repository* en pièce jointe, dans les fichiers *antlr4/com/SmallConcurrency/SmallConurrencyGrammar.g4* et *antlr4/imports/SmallConcurrencySyntax.g4*.

2.2 Sémantique

Afin de mettre en place une sémantique opérationnelle concurrente, il est nécessaire de pouvoir représenter l'état du programme à un instant t. Pour ce faire, nous définirons un état du programme comme ceci :

$$E = (\Omega, \Gamma, S, \Sigma, \phi, \pi, \Phi, \Lambda)$$

avec :

- Ω : l'ensemble des variables globales manipulées par le programme
- Γ : l'ensemble des tuples de la forme (S, Σ, ϕ) représentant un thread où :
  - Ξ : l'ensemble des séquences d'instruction encore à exécuter pour chaque environnement local de ce thread
  - Σ : l'ensemble des environnements de variables locales possibles
  - ϕ : l'ensemble des variables globales détenues (locked) par le thread
  - π : la liste des variables dans lesquels stockés les valeurs de retour des fonctions
- Ξ : l'ensemble des séquences d'instructions encore à exécuter pour chaque environnement local du thread sélectionné pour exécution
- Σ : l'ensemble des environnements de variables locales possible du thread sélectionné pour exécution
- ϕ : l'ensemble des variables globales détenues (locked) par le thread sélectionné pour exécution
- π : la liste des variables dans lesquels stockés les valeurs de retour des fonctions du thread sélectionné pour exécution
- Φ : l'ensemble des variables globales détenues (locked) par n'importe quel thread
- Λ : l'ensemble des fonctions définies

On notera également que :

- $\Sigma \circ \sigma$ représente la séquence où σ est le dernier environnement et Σ est l'ensemble des environnements précédents ;
- $\sigma : \langle Var \rangle \mapsto \mathbb{Z} \cup True, False$
- $\lambda :< Func > \mapsto (x_n, B)$ où x_n est la liste des paramètres de la fonction et B, la séquence d'instruction de la fonction
- $\Xi \circ \xi$ représente la séquence où ξ sont les instructions dans le dernier environnement et Ξ, l'ensemble des instructions dans les environnements précédents
- s : : ξ représente la séquence d'instruction où s est la première instruction et ξ la liste d'instructions suivantes.

De plus, pour modéliser la concurrence, il est nécessaire de pouvoir déterminer un opérateur indéterministe de choix permettant de procéder à l'exécution d'une instruction de chaque thread de manière aléatoire. Pour ce faire, nous allons définir l'opérateur | entre deux états. De manière plus générale, nous définirons $||_{i=0}^{n} = E_0|E_1|...|E_{n-1}|E_n$

Ci-dessous, nous définissons les règles de transition de la sémantique opérationnelle. Concrètement, l'idée est de partir d'un état initial $(\varnothing, (\Xi, \varnothing, \varnothing, \varnothing), <>, <>, <>, <>, \varnothing, \varnothing)$ où :

— Ξ est l'ensemble des instructions du programme (déclaration de variables globales et de fonctions)

— $<>$ représente l'absence de valeur

Depuis cet état initial, on appliquera la règle de transition $multi - threading$ pour sélectionner le thread pour lequel une instruction sera exécuté (ici, il n'y en a qu'un). L'instruction sera ensuite exécutée et on reviendra à un état similaire au premier avec les environnement Γ, Φ et Λ mise à jour. La règle $multi - threading$ s'appliquera tant qu'on ne rencontre pas le symbole de fin de fichier EOF.

Dans les règles ci-dessous, la valeur KO pour l'évaluation d'une variable signifie que la variable existe, mais a été locked par un autre thread. Ainsi, la valeur KO est l'élément absorbant pour l'opération $\oplus$ représentant chacun des opérateurs binaires en SmallConcurrent.

De plus, une instruction EOT est rajoutée à la fin de chaque thread pour signifier qu'il a terminé son exécution.

2.2.1 Évaluation d'expressions simples en SmallConcurrent

$$[True] \quad \frac{}{(\Omega, \Gamma, True, \sigma, \phi, \pi, \Phi, \Lambda) \rightsquigarrow True}$$

$$[False] \quad \frac{}{(\Omega, \Gamma, False, \sigma, \phi, \pi, \Phi, \Lambda) \rightsquigarrow False}$$

$$[Int] \quad \frac{v \in \mathbb{Z}}{(\Omega, \Gamma, v, \sigma, \phi, \pi, \Phi, \Lambda) \rightsquigarrow v}$$

$$[Var - local] \quad \frac{x \in < Var > \quad x \in \sigma}{(\Omega, \Gamma, x, \sigma, \phi, \pi, \Phi, \Lambda) \rightsquigarrow \sigma(x)}$$

$$[Var - global - ok] \quad \frac{x \in < Var > \quad x \in \Omega \quad (x \notin \Phi \vee x \in \phi)}{(\Omega, \Gamma, x, \sigma, \phi, \pi, \Phi, \Lambda) \rightsquigarrow \Omega(x)}$$

$$[Var - global - ko] \quad \frac{x \in < Var > \quad (x \in \Omega \wedge x \in \Phi \wedge x \notin \phi)}{(\Omega, \Gamma, x, \sigma, \phi, \pi, \Phi, \Lambda) \rightsquigarrow KO}$$

$$[OP] \quad \frac{(\Omega, x1, \sigma, \Phi, \Lambda) \rightsquigarrow v1 \quad (\Omega, x2, \sigma, \Phi) \rightsquigarrow v2 \quad v1 \oplus v2 = v}{(\Omega, \Gamma, x1 \oplus x2, \sigma, \phi, \pi, \Phi, \Lambda) \rightsquigarrow v}$$

$$[function] \quad \frac{f \in < Func > \quad x \in \Lambda}{(\Omega, \Gamma, f, \sigma, \phi, \pi, \Phi, \Lambda) \rightsquigarrow \Lambda(f)}$$

2.2.2 Sémantique de l'exécution d'instruction en SmallConcurrent

$$[Multi - threading] \quad \frac{n = \#\Gamma \quad \forall i : 0 \leq i \leq n \ ((\Omega, \Gamma \backslash \{(\Xi_i, \Sigma_i, \Phi_i)\}, \Xi_i, \Sigma_i, \phi_i, \pi_i, \Phi, \Lambda) \rightsquigarrow (\Omega'_i, \Gamma'_i, <>, <>, <>, <>, \Phi'_i, \Lambda'_i))}{(\Omega, \Gamma, <>, <>, <>, <>, \Phi, \Lambda) \rightsquigarrow ||_{i=0}^{n}(\Omega'_i, \Gamma'_i, <>, <>, <>, \Phi'_i, \Lambda'_i)}$$

$$[EOT] \quad \frac{}{(\Omega, \Gamma, \Xi \circ (EOT :: \xi), \Sigma, \varnothing, \Phi, \pi, \Lambda) \rightsquigarrow (\Omega, \Gamma, <>, <>, <>, <>, \Phi, \Lambda)}$$

$$[Local\ assignment - ok] \quad \frac{x \in \sigma \quad (\Omega, \Gamma, e, \sigma, \phi, \pi, \Phi, \Lambda) \rightsquigarrow v \quad v \neq KO \quad \sigma' = \sigma[x \mapsto v]}{(\Omega, \Gamma, \Xi \circ ((x = e) :: \xi), \Sigma \circ \sigma, \phi, \pi, \Phi, \Lambda) \rightsquigarrow (\Omega, \Gamma \cup \{(\Xi \circ \xi, \Sigma \circ \sigma', \phi, \pi)\}, <>, <>, <>, <>, \Phi, \Lambda)}$$

$$[Local\ assignment - ko]\quad \frac{x \in \sigma \quad (\Omega, \Gamma, e, \sigma, \phi, \pi, \Phi) \rightsquigarrow v \quad v = KO}{(\Omega, \Gamma, \Xi \circ ((x = e) :: \xi), \Sigma \circ \sigma, \phi, \pi, \Phi, \Lambda) \rightsquigarrow (\Omega, \Gamma \cup \{(\Xi \circ ((x = e) :: \xi), \Sigma \circ \sigma, \phi, \pi)\}, <>, <>, <>, <>, \Phi, \Lambda)}$$

$$[Lock - ok]\quad \frac{x \in \Omega \quad x \notin \Phi \quad \phi' = \phi \cup \{x\} \quad \Phi' = \Phi \cup \{x\}}{(\Omega, \Gamma, \Xi \circ ((lock\ x) :: \xi), \Sigma, \phi, \pi, \Phi, \Lambda) \rightsquigarrow (\Omega, \Gamma \cup \{(\Xi \circ \xi, \Sigma, \phi', \pi)\}, <>, <>, <>, <>, \Phi', \Lambda)}$$

$$[Lock - ko]\quad \frac{x \in \Phi}{(\Omega, \Gamma, \Xi \circ ((lock\ x) :: \xi), \Sigma, \phi, \pi, \Phi, \Lambda) \rightsquigarrow (\Omega, \Gamma \cup \{(\Xi \circ ((lock\ x) :: \xi), \Sigma, \phi, \pi)\}, <>, <>, <>, <>, \Phi, \Lambda)}$$

$$[Unlock - ok]\quad \frac{x \in \phi \quad \phi' = \phi \backslash \{x\} \quad \Phi' = \Phi \backslash \{x\}}{(\Omega, \Gamma, \Xi \circ ((unlock\ x) :: \xi), \Sigma, \phi, \pi, \Phi, \Lambda) \rightsquigarrow (\Omega, \Gamma \cup \{(\Xi \circ \xi, \Sigma, \phi', \pi)\}, <>, <>, <>, <>, \Phi', \Lambda)}$$

$$[Unlock - ko]\quad \frac{x \in \Omega \quad x \notin \phi}{(\Omega, \Gamma, \Xi \circ ((unlock\ x) :: \xi), \Sigma, \phi, \pi, \Phi, \Lambda) \rightsquigarrow (\Omega, \Gamma \cup \{(\Xi \circ ((unlock\ x) :: \xi), \Sigma, \phi, \pi)\}, <>, <>, <>, <>, \Phi, \Lambda)}$$

$$[New - thread]\quad \frac{}{(\Omega, \Gamma, (\Xi \circ ((thread\{B\}) :: \xi), \Sigma, \phi, \pi, \Phi, \Lambda) \rightsquigarrow (\Omega, \Gamma \cup \{\Xi \circ \xi, \Sigma, \phi, \pi)\} \cup \{(B, \varnothing, \varnothing)\}, <>, <>, <>, <>, \Phi, \Lambda)}$$

$$[Global - assignment - ok]\quad \frac{x \in \Omega \quad (x \notin \Phi \vee x \in \phi) \quad (\Omega, \Gamma, e, \sigma, \phi, \pi, \Phi, \Lambda) \rightsquigarrow v \quad v \neq KO \quad \Omega' = \Omega[x \mapsto v]}{(\Omega, \Gamma, \Xi \circ ((x = e) :: \xi), \Sigma \circ \sigma, \phi, \pi, \Phi, \Lambda) \rightsquigarrow (\Omega', \Gamma \cup \{(\Xi \circ \xi, \Sigma \circ \sigma, \phi, \pi)\}, <>, <>, <>, <>, \Phi, \Lambda)}$$

$$[Global - assignment - ko]\quad \frac{(x \in \Omega \quad (x \in \Phi \wedge x \notin \phi)) \vee (x \in \Omega \quad (x \notin \Phi \vee x \in \phi) \quad (\Omega, \Gamma, e, \sigma, \phi, \pi, \Phi, \Lambda) \rightsquigarrow v \quad v = KO)}{(\Omega, \Gamma, \Xi \circ ((x = e) :: \xi), \Sigma \circ \sigma, \phi, \pi, \Phi, \Lambda) \rightsquigarrow (\Omega, \Gamma \cup \{(\Xi \circ ((x = e) :: \xi), \Sigma \circ \sigma, \phi, \pi)\}, <>, <>, <>, <>, \Phi, \Lambda)}$$

$$[While - True]\quad \frac{(\Omega, \Gamma, e, \sigma, \phi, \pi, \Phi, \Lambda) \rightsquigarrow True}{(\Omega, \Gamma, \Xi \circ ((while(e)\ B) :: \xi), \Sigma, \phi, \pi, \Phi, \Lambda) \rightsquigarrow (\Omega, \Gamma \cup \{(\Xi \circ (B :: [(while(e)\ B) :: \xi)], \Sigma, \phi, \pi)\}, <>, <>, <>, <>, \Phi, \Lambda)}$$

$$[While - false]\quad \frac{(\Omega, \Gamma, e, \sigma, \phi, \pi, \Phi, \Lambda) \rightsquigarrow False}{(\Omega, \Gamma, \Xi \circ ((while(e)\ B) :: \xi), \Sigma, \phi, \pi, \Phi, \Lambda) \rightsquigarrow (\Omega, \Gamma \cup \{\Xi \circ \xi, \Sigma, \phi, \pi)\}, <>, <>, <>, <>, \Phi, \Lambda)}$$

$$[While - ko]\quad \frac{(\Omega, \Gamma, e, \sigma, \phi, \pi, \Phi, \Lambda) \rightsquigarrow KO}{\Omega, \Gamma, \Xi \circ ((while(e)\ B) :: \xi), \Sigma, \phi, \pi, \Phi, \Lambda) \rightsquigarrow \Omega, \Gamma \cup \{(\Xi \circ ((while(e)\ B) :: \xi), \Sigma, \phi, \pi)\}, <>, <>, <>, <>, \Phi, \Lambda)}$$

$$[If - True]\quad \frac{(\Omega, \Gamma, e, \sigma, \phi, \pi, \Phi, \Lambda) \rightsquigarrow True}{(\Omega, \Gamma, \Xi \circ ((if(e1)\ B_1\ else\ B_2) :: \xi), \Sigma, \phi, \pi, \Phi, \Lambda) \rightsquigarrow (\Omega, \Gamma \cup \{\Xi \circ (B_1 :: \xi)], \Sigma, \phi, \pi)\}, <>, <>, <>, <>, \Phi, \Lambda)}$$

$$[If - False] \quad \frac{(\Omega, \Gamma, e, \sigma, \phi, \pi, \Phi, \Lambda) \rightsquigarrow False}{(\Omega, \Gamma, \Xi \circ ((if(e1)\ B_1\ else\ B_2) :: \xi), \Sigma, \phi, \pi, \Phi, \Lambda) \rightsquigarrow (\Omega, \Gamma \cup \{\Xi \circ (B_2 :: \xi)], \Sigma, \phi, \pi)\}, <>, <>, <>, <>, \Phi, \Lambda)}$$

$$[If - ko] \quad \frac{(\Omega, \Gamma, e, \sigma, \phi, \pi, \Phi, \Lambda) \rightsquigarrow KO}{\Omega, \Gamma, \Xi \circ ((if(e1)\ B_1\ else\ B_2) :: \xi), \Sigma, \phi, \pi, \Phi, \Lambda) \rightsquigarrow \Omega, \Gamma \cup \{(\Xi \circ ((if(e1)\ B_1\ else\ B_2) :: \xi), \Sigma, \phi, \pi)\}, <>, <>, <>, <>, \Phi, \Lambda)}$$

$$[Function\ call - ok] \quad \frac{x \in \Omega \vee x \in \sigma \quad (\Omega, \Gamma, f, \sigma, \phi, \pi, \Phi, \Lambda) \rightsquigarrow (x_n, B) \quad \#e_n = \#x_n \quad (\Omega, \Gamma, e_n, \sigma, \phi, \pi, \Phi, \Lambda) \rightsquigarrow a_n \quad \forall n : a_n \neq KO \quad \sigma_n = [x_n \mapsto a_n] \quad \pi' = \pi \circ x}{(\Omega, \Gamma, \Xi \circ (x = f(e_n) :: \xi), \Sigma \circ \sigma, \phi, \pi, \Phi, \Lambda) \rightsquigarrow (\Omega, \Gamma \cup \{\Xi \circ (\xi) \circ B), \Sigma \circ \sigma \circ \sigma_n, \phi, \pi')\}, <>, <>, <>, <>, \Phi, \Lambda)}$$

$$[Function\ call - ko] \quad \frac{x \in \Omega \vee x \in \sigma \quad (\Omega, \Gamma, f, \sigma, \phi, \pi, \Phi, \Lambda) \rightsquigarrow (x_n, B) \quad \#e_n = \#x_n \quad (\Omega, \Gamma, e_n, \sigma, \phi, \pi, \Phi, \Lambda) \rightsquigarrow a_n \quad \exists n : a_n = KO}{(\Omega, \Gamma, \Xi \circ (x = f(e_n) :: \xi), \Sigma \circ \sigma, \phi, \pi, \Phi, \Lambda) \rightsquigarrow (\Omega, \Gamma \cup \{(\Xi \circ (x = f(e_n) :: \xi), \Sigma \circ \sigma, \phi, \pi)\}, <>, <>, <>, <>, \Phi, \Lambda)}$$

$$[Return - ok] \quad \frac{(\Omega, \Gamma, e, \sigma\_2, \phi, \pi, \Phi, \Lambda) \rightsquigarrow v \quad v \neq KO \quad returnVar = \pi.last() \quad \pi' = \pi[: -1]}{(\Omega, \Gamma, \Xi \circ \xi \circ (return\ e :: S), \Sigma \circ \sigma\_1 \circ \sigma\_2, \phi, \pi, \Phi, \Lambda) \rightsquigarrow (\Omega, \Gamma \cup \{(\Xi \circ ((returnVar = v) :: \xi), \Sigma, \phi, \pi')\}, <>, <>, <>, <>, \Phi, \Lambda)}$$

$$[Return - ko] \quad \frac{(\Omega, \Gamma, e, \sigma, \phi, \pi, \Phi, \Lambda) \rightsquigarrow v \quad v = KO}{(\Omega, \Gamma, \Xi \circ (x = 0 :: \xi) \circ (return\ e), \Sigma \circ \sigma, \phi, \pi, \Phi, \Lambda) \rightsquigarrow (\Omega, \Gamma \cup \{(\Xi \circ (x = 0 :: \xi) \circ (return\ e), \Sigma \circ \sigma, \phi, \pi)\}, <>, <>, <>, <>, \Phi, \Lambda)}$$

$$[VarDecl - global] \quad \frac{x \notin \Omega \quad \Omega' = \Omega[x \mapsto 0]}{(\Omega, \Gamma, \Xi \circ ((global\_var\ x) :: \xi), \varnothing, \varnothing, \varnothing, \Phi, \Lambda) \rightsquigarrow (\Omega', \Gamma \cup \{(\Xi \circ \xi, \varnothing, \varnothing, \varnothing)\}, <>, <>, <>, <>, \Phi, \Lambda)}$$

$$[VarDecl - local] \quad \frac{x \notin \Omega \quad x \notin \sigma \quad \sigma' = \sigma[x \mapsto 0]}{(\Omega, \Gamma, \Xi \circ ((var\ x) :: \xi), \Sigma \circ \sigma, \phi, \pi, \Phi, \Lambda) \rightsquigarrow (\Omega, \Gamma \cup \{(\Xi \circ \xi, \Sigma \circ \sigma', \phi, \pi)\}, <>, <>, <>, <>, \Phi, \Lambda)}$$

$$[function - main - def] \quad \frac{f \notin \Lambda \quad \Lambda' = \Lambda[f \mapsto (x_n, B)]}{(\Omega, \Gamma, \Xi \circ (function\ main()\ B) :: \xi)), \varnothing, \varnothing, \varnothing, \Lambda) \rightsquigarrow (\Omega, \Gamma \cup \{(\Xi \circ (B :: \xi), \varnothing, \varnothing, \varnothing)\}, <>, <>, <>, <>, \varnothing, \Lambda')}$$

$$[function - not - main - def] \quad \frac{f \notin \Lambda \quad \Lambda' = \Lambda[f \mapsto (x_n, B)]}{(\Omega, \Gamma, \Xi \circ ((function\ f(x_n)\ B) :: \xi), \varnothing, \varnothing, \varnothing, \Lambda) \rightsquigarrow (\Omega, \Gamma \cup \{(\Xi \circ \xi, \varnothing, \varnothing, \varnothing)\}, <>, <>, <>, <>, \Phi, \Lambda')}$$

3 Domaine abstrait et lattice

Pour définir une dataflow analysis afin de détecter des situations de race conditions, il est nécessaire de faire le suivi des accès des différents threads aux variables globales du programme.

Pour ce faire, on va analyser les accès aux variables en se basant sur le domaine abstrait suivant :

$$L = \{NA, RA, WA, RC\}$$

où

— NA, représente le fait qu'une variable n'ait jamais été accédée par le thread ;
— RA, représente le fait qu'une variable ait été uniquement accédée en lecture par le thread ;
— WA représente le fait qu'une variable ait été accédée en écriture par le thread ;
— RC, représente le fait qu'il y ait une race condition possible sur la variable.

La lattice de ce domaine abstrait est la suivante :

Dans le cadre de notre analyse, un accès en écriture est considéré comme supérieur à un accès en lecture, car deux accès en lecture ne produisent pas une RC. Un accès en lecture/écriture et un en écriture, par contre, produit bien une RC. Il sera dès lors important de retenir que la variable a été accédée en écriture plutôt qu'en lecture.

4 Fonction d'abstraction et de concrétisation

Il est important de noter que l'environnement abstrait mis en place est fortement différent de l'environnement concret des variables du programme ($\mathbb{Z}$). Afin de pouvoir faire les mappings réciproques entre l'environnement abstrait et l'environnement concret, nous définissons deux fonctions :

- La fonction d'abstraction $\alpha : \mathbb{Z} \mapsto L$
- La fonction d'abstraction $\gamma : L \mapsto \mathbb{Z}$

4.1 Fonction d'abstraction

Comme le lien entre l'environnement concret et l'environnement abstrait est très peu explicite (transformer une valeur entière concrète en une valeur d'accès à une variable), l'on définira la fonction d'abstraction comme ceci, $\forall v \in \mathbb{Z}$:

$$\alpha(v) = RC$$

N'ayant pas plus d'information sur l'utilisation de la valeur, par prudence, la fonction d'abstraction mappera chaque valeur entière en une RC.

4.2 Fonction de concrétisation

Effectuer le mapping dans l'autre sens est plus facile, la valeur abstraite d'accès à une variable ne fournissant aucune information sur la valeur concrète. Chaque valeur abstraite sera mappée vers l'ensemble de toutes les valeurs concrètes possibles $\mathbb{Z}$:

$$\gamma(NA) = \mathbb{Z}$$

$$\gamma(RA) = \mathbb{Z}$$

$$\gamma(WA) = \mathbb{Z}$$

$$\gamma(RC) = \mathbb{Z}$$

5 Fonctions de flux

Les fonctions de flux associées à chaque instruction du programme permettent de gérer la transition des valeurs abstraites des variables partagées dans l'environnement.

Comme il est ici nécessaire de pouvoir connaitre de l'information sur les accès possiblement faits par les threads exécutés en parallèle du thread courant, on va mettre en place deux environnements abstraits :

— ϕ : l'environnement abstrait courant dans lequel les valeurs sont mises à jour ;

— ϕ_c : l'environnement abstrait récupéré après le passage sur un ou plusieurs threads possibles avant l'instruction courante.

Chaque couple d'environnement abstrait n'est valable que dans le contexte d'un seul thread, nous définirons la pile Γ stockant les threads du programme. Le dernier élément de la pile sera le couple d'environnements abstraits courants.

$$\Gamma = \langle(\phi_0, \phi_{0_c}), ..., (\phi_n, \phi_{n_c})\rangle$$

Un exemple simple pour comprendre l'environnement ϕ_c est la possibilité d'analyser une instruction pour laquelle on a précédemment lancé un thread en parallèle. L'analyse va dans un premier temps analyser les accès aux variables dans le thread en question et reportée ces accès dans l'environnement ϕ_c. Cet environnement va permettre de définir si une lecture ou une écriture dans une variable doit être considéré comme une race condition ou non.

Dans des cas complexes, comme le lancement de threads en chaines ou de thread en séries, il est nécessaire de pouvoir mettre en place un moyen de non seulement reporter les accès faits par un thread, mais aussi vérifier que deux threads, qui pourraient s'exécuter en même temps, ne puissent pas produire de race condition. Pour ce faire, nous définissons un opérateur de détection des RC entre des threads s'exécutant en parallèle x $\oplus$ y avec $x, y \in L$. Pour rappel, une RC apparait lorsqu'il y a au minimum un thread qui modifie et un thread qui lit l'état de la variable de manière simultanée.

$$NA \oplus NA = NA$$

$$RA \oplus RA = RA$$

$$RA \oplus NA = RA$$

$$WA \oplus NA = WA$$

$$x \oplus RC = RC \ avec \ x \in L$$

$$WA \oplus RA = RC$$

$$WA \oplus WA = RC$$

Cet opérateur va nous permettre de définir comment évaluer les résultats des accès concurrents entre les threads.

Dans le cas des threads lancés en chaine (thread lancé dans un thread), il est nécessaire de pouvoir résumer les accès d'un thread à tous les accès qui sont effectués à l'intérieur de lui-même et de ses sous-threads. Pour ce faire, l'opérateur $\sqcup$ (least upper bound) permet de retenir du thread les accès les plus élevés qui sont faits.

Dans la suite, nous présenterons des exemples de fonctions de flux résumant plutôt bien le cas de la lecture et la modification de l'état d'une variable. **Ceux-ci ne sont uniquement valables quand la variable n'est pas locked par le thread courant**. Si la variable est locked, alors la fonction de flux ne modifie pas Γ.

Prenons les fonctions de flux pour l'assignation simple (à une constante et à une variable).

$$f[x = c](\Gamma \circ (\phi, \phi_c)) \quad = \quad \Gamma \circ (\phi[x \mapsto (WA \oplus \phi_c(x)) \sqcup \phi(x)], \phi_c) \qquad \text{où} \quad c \in \mathbb{Z}$$

$$f[x = y](\Gamma \circ (\phi, \phi_c)) \quad = \quad \Gamma \circ (\phi[x \mapsto (WA \oplus \phi_c(x)) \sqcup \phi(x), y \mapsto (RA \oplus \phi_c(y)) \sqcup \phi(y)], \phi_c) \quad \text{où} \quad y \in \langle Var \rangle$$

Dans les deux fonctions de flux précédentes, la logique est donc la suivante :

1. Calculer le résultat de l'opération de détection de RC entre l'accès fait (lecture ou écriture) et l'accès fait par le(s) thread(s) s'exécutant en parallèle.
2. Calculer le least upper bound entre le nouvel accès et l'ancien accès effectué.

Voyons maintenant les deux fonctions de flux qui manipulent la pile de couples d'environnements Γ

$$f[thread](\Gamma \circ (\phi, \phi_c)) \quad = \quad \Gamma \circ (\phi, \phi_c) \circ (\phi_t, \phi_{t_c}) \quad \text{où } (\phi_t | \forall v \in \phi : \phi_t(v) = NA) \ \wedge \ (\phi_{t_c} | \forall v \in \phi : \phi_{t_c}(v) = NA)$$

$$f[endThread](\Gamma \circ (\phi, \phi_c) \circ (\phi_t, \phi_{t_c})) \quad = \quad \Gamma \circ (\phi, \phi_c[\forall v \in \phi_c : v \mapsto \phi_c(v) \oplus (\phi_t(v) \sqcup \phi_{t_c}(v))$$

Ces deux fonctions de flux illustrent l'empilement et le dépilement effectués lors de l'entrée et la sortie d'un thread.

L'empilement d'un nouveau couple d'environnements empile un environnement courant ϕ_t et un environnement concurrent ϕ_{t_c} où toutes les variables de l'environnement original ϕ sont mises à NA, un nouveau thread est créé.

Le dépilement est plus complexe et inclus plusieurs modifications :

- Dépiler un couple d'environnements
- Calculer le nouvel environnement ϕ_c
  1. Calculer le least upper bound entre ϕ_t et ϕ_{t_c}
  2. Calculer le résultat de l'opération de détection de RC entre le résultat précédemment récupéré et les accès concurrents stockés précédemment.

6 Soundness & Completeness

En analyse de programme, deux des propriétés les plus intéressantes et difficilement cumulables sont la **soundness** et la **completeness**. La **soundness** assure qu'une analyse ne générera jamais de faux positif, mais pourrait ne pas repérer certains cas. La **completeness**, elle, se concentre sur le fait de repérer tous les cas possibles même si elles se plantent certaines fois, il n'y pas de faux négatifs.

L'analyseur ici présenté n'est pas sound, mais il est **complet**. S'il existe une possibilité qu'une race condition survienne, l'analyseur pourra la repérer. Cependant, il se peut aussi que l'analyse reporte la présence de race condition alors qu'en réalité celle-ci n'arrivera jamais à l'exécution. Un exemple simple pour bien comprendre comment l'analyse fonctionne est le suivant (que vous pouvez retrouver dans les tests sur le repo).

```
gVar x;
gVar y;

function main(){
    thread {
        x = 1;
        thread {
            var i ;
            i = 0;

            if (i==0){
                i = 1;
            }
            else{
                x = 0;
            }
        }
        x=2;

    }

}
```

Dans le code ci-dessus, il est clair qu'aucune exécution du programme n'empruntera jamais la branche else dans le thread. Comme l'analyse ne retient pas l'état de la variable, elle visite les instructions *if else* sans savoir si le code sera exécuté. Ainsi, l'analyseur détectera une race condition sur la variable x bien que dans les faits pour toute trace T du programme, la race condition n'apparaitra jamais.

Bien qu'elle ne soit pas sound, l'analyse l'est **localement** pour certaines fonctions de flux. Pour le démontrer, prenons la fonction de flux suivante :

$$f[x = c](\Gamma \circ (\phi, \phi_c)) \quad = \quad \Gamma \circ (\phi[x \mapsto (WA \oplus \phi_c(x)) \sqcup \phi(x)], \phi_c) \quad \text{où} \quad c \in \mathbb{Z}$$

Prouvons qu'elle est locally sound :

Soit σ, tq $\alpha(\sigma) \sqsubseteq \phi$

On a :

$$\phi(x) \in L$$

$$\Rightarrow \alpha(\sigma)(x) \in L$$

$$\Rightarrow \sigma(x) \in \mathbb{Z}$$

Par les règles sémantiques et la fonction d'abstraction $\alpha(v) = RC$:

$$\sigma' = \{x \to c, v_0 \to \sigma(v_0), ..., v_n \to \sigma(v_n)\}$$

$$\alpha(\sigma') = \{x \to RC, le\ reste\ inchang\acute{e}\ par\ rapport\ \grave{a}\ \alpha(\sigma)\}$$

$$\sqsubseteq \phi' = \{x \to ((WA \oplus \phi_c(x) \sqcup \phi(x)), le\ reste\ inchang\acute{e}\ par\ rapport\ \grave{a}\ \alpha(\sigma)\}$$

$$\Leftrightarrow \begin{cases} \alpha(\sigma') \sqsubseteq \{x \to WA, le\ reste\ inchang\acute{e}\ par\ rapport\ \grave{a}\ \alpha(\sigma)\} \\ \alpha(\sigma') \sqsubseteq \{x \to RC, le\ reste\ inchang\acute{e}\ par\ rapport\ \grave{a}\ \alpha(\sigma)\} \end{cases}$$

7 Détection d'erreurs et warnings

Le programme fourni permet ainsi la détection de race condition dans le code, mais également de détecter certaines erreurs sémantiques grâce à l'implémentation de la sémantique opérationnelle. L'idée étant de ne pas lancer l'analyse sur un code qui ne soit pas un minimum syntaxiquement et sémantiquement correct, les exceptions capturées par le parser et l'analyseur sémantique mettront fins à l'exécution de l'analyse en loggant les erreurs rencontrées. L'analyse de race conditions, elle, ne va pas interrompre l'exécution du programme, mais reporter les erreurs sous forme de warning dans les logs. Voyons en détail les erreurs et warnings que le programme permet de détecter.

7.1 Parser et sémantique opérationnelle

Le parser, généré par ANTLR sur base du fichier *SmallConcurrencyGrammar.g4* et *SmallConcurrencySyntax.g4*, permet de détecter les erreurs de syntaxe dans le code fourni en entrée. Il effectue une analyse complète du code fourni et log l'ensemble des erreurs qu'il a détectées avant de mettre fin à l'exécution.

L'implémentation de la sémantique opérationnelle tente de simuler l'exécution du programme en empruntant les règles sémantiques. Cette analyse permet de détecter plusieurs erreurs :

- Fonction appelée, mais non déclarée
- Vérifier qu'un thread a bien lâché le verrou d'une variable avant de terminer
- Vérifier que les variables globales et locales ont été déclarées
- Vérifier qu'une instruction *return* ne se trouve pas dans la fonction main
- Vérifier que les fonctions sont appelées avec le bon nombre d'arguments
- Vérifier que les conditions sont bien booléennes
- Vérifier qu'on ne déclare pas deux fois une fonction avec le même nom (surcharge non autorisée)
- Vérifier qu'une variable n'est pas déclarée deux fois
- Vérifier qu'une variable locale ne peut pas être locked et unlocked
- Vérifier qu'une instruction unlock se trouve après le lock de la variable en question.

7.2 Analyse statique : détection des situations de courses

L'analyse statique permet essentiellement de détecter la présence de race conditions sur les variables partagées du code. Lorsque l'analyse détecte une possible race condition, elle log l'erreur sous forme de warning en indiquant la ligne à laquelle la possible situation de course a été détectée. À la fin de l'analyse, le programme log à son tour l'ensemble des variables qui ont été considérées comme potentiellement à risque durant l'analyse. À noter que les race conditions détectées entre des threads en série ne sont pas logguées mais bien enregistrées pour le logging final.

```
Finished analyzing file: RC4.smallConcurrent
The program is semantically correct!
Starting cfg generation...
Finished cfg generation!
2024-01-07 00:09:31.681 [main] WARN  com.SmallConcurrency.staticAnalysis.StaticAnalysisVisitor - Race condition detected on variable x at line 14:
Read may happen during write in another thread
2024-01-07 00:09:31.683 [main] WARN  com.SmallConcurrency.staticAnalysis.StaticAnalysisVisitor - Race condition detected on variable y at line 14:
Write may happen during read or write in another thread
2024-01-07 00:09:31.683 [main] WARN  com.SmallConcurrency.main.Main - Race condition detected on variable x!
2024-01-07 00:09:31.683 [main] WARN  com.SmallConcurrency.main.Main - Race condition detected on variable y!

Process finished with exit code 0
```

8 Formulation selon le framework GEN/KILL

Le framework GEN/KILL permet d'exprimer une analyse statique sur base de l'utilisation d'ensembles de variables respectant certaines propriétés du programme. L'idée ici n'est plus de sauvegarder la valeur abstraite pour chaque variable, mais de placer une variable dans un ensemble particulier s'il respecte une propriété. Pour représenter notre analyse avec ce framework, il est nécessaire de définir dans un premier temps le type d'analyse à mettre en place et dans un second temps comment adapter le framework GEN/KILL à la propriété évaluée.

8.1 Type d'analyse

L'analyse implémentée étant complète et pas sound, il parait plus intéressant de mettre en place une **may analysis**. En effet, l'objectif de l'analyse sera de déterminer s'il est possible qu'il y ait une race condition sur une variable du programme.

8.2 Modélisation de l'analyse

Mettre en place une analyse GEN/KILL traditionnelle semble assez limité, par rapport à la propriété analysée. Notre analyse se base en effet sur la capture des accès aux variables pour pouvoir déterminer si une RC survient. Évidemment, il serait possible de formuler une analyse complète avec le framework GEN/KILL, détectant les RC mais générant aussi beaucoup de faux positifs. Cela pourrait être fait en rajoutant simplement la variable dans l'ensemble si celle-ci est accédée. Cependant, cette analyse ne nous permettrait d'en fait que de récupérer la liste des variables partagées qui sont accédées par le programme, ce qui est loin de la précision de la propriété recherchée.

C'est pourquoi, dans la suite, nous proposons une adaptation du framework GEN/KILL afin de pouvoir le rendre plus précis pour la détection des RC.

8.2.1 Environnement abstrait

Pour ce faire, nous définirons un environnement ϵ de la manière suivant :

$$\epsilon = (\phi_{RA}, \phi_{RAc}, \phi_{WA}, \phi_{WAc}, \phi_{RC}, \phi_{RCc})$$

$$o\text{ù}$$

$$\forall v \in \phi_{RA} : v \in \mathbb{P}(V)$$

$$\forall v \in \phi_{RAc} : v \in \mathbb{P}(V)$$

$$\forall v \in \phi_{WA} : v \in \mathbb{P}(V)$$

$$\forall v \in \phi_{WAc} : v \in \mathbb{P}(V)$$

$$\forall v \in \phi_{RC} : v \in \mathbb{P}(V)$$

$$\forall v \in \phi_{RCc} : v \in \mathbb{P}(V)$$

L'environnement abstrait permet de stocker dans un tuple les variables dans des ensembles ϕ représentant chacun une propriété d'accès à une variable par le thread courant ou par les threads concurrents. Ainsi, il sera possible de sauvegarder l'accès successifs à une variable au sein de ces ensembles. On définira néanmoins l'invariant suivant :

$$I \equiv \forall i, j : 0 \leq i, j \leq 5 | i \neq j \ \wedge v \in \epsilon[i] \Rightarrow v \notin \epsilon[j]$$

vérifiant qu'une varaible ne se trouve jamais dans plusieurs ensembles avec une propriété différentes au sein d'un même environnement abstrait ϵ.

8.2.2 Lattice et opération

L'environnement abstrait ϵ étant un tuple d'ensemble, nous définissons la lattice en définissant les opérations $\sqsubseteq$ et $\sqcup$ de la manière suivante :

$$\epsilon_1 \sqsubseteq \epsilon_2 \Leftrightarrow \forall i : 0 \leq i \leq 5, \forall v : v \in \epsilon_1[i] : \exists j : i \leq j \leq 5 | v \in \epsilon_2[j]$$

$$\epsilon_1 \sqcup \epsilon_2$$

$$= \epsilon_3 \cup \epsilon_4 \mid$$

$$[\forall i : 0 \leq i \leq 5, \forall v : v \in \epsilon_1[i] : (\nexists j : i < j \leq 5 : v \in \epsilon_2[j] \Leftrightarrow v \in \epsilon_3[i]]$$

$$\wedge$$

$$[\forall i : 0 \leq i \leq 5, \forall v : v \in \epsilon_1[i] : (\nexists j : i < j \leq 5 : v \in \epsilon_2[j] \Leftrightarrow v \in \epsilon_3[i]]$$

$$\top = (\varnothing, \varnothing, \varnothing, \varnothing, \varnothing, V)$$

$$\bot = (\varnothing, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing)$$

8.2.3 Stack d'environnement

La présence de thread implique la gestion de plusieurs environnements ϵ. C'est pourquoi nous définissons la stack E :

$$E = \langle \epsilon_0, ..., \epsilon_n \rangle$$

Comme pour l'analyse présentée ci-avant, l'environnement courant sera considéré comme étant le dernier environnement de la pile.

8.2.4 GEN et KILL

Pour chaque point du programme p, il nous reste à définir les opérations suivantes GEN et KILL qui vont manipuler l'état des ensembles de variable dans l'environnement courant et concurrent. Dans la suite, nous détaillons ces opérations dans le cadre de l'assignation d'une variable globale. La logique concernant la lecture peut être reprise pour définir les opérations pour les autres instructions lisant l'état d'une variable. Pour tous les autres points de programme p, on définit pour un environnement ϵ :

$$GEN(\epsilon)(p) = \bot$$

$$KILL(\epsilon)(p) = \bot$$

L'analyse mise en place peut ainsi être définie comme un ensemble d'équations qui permette la transition entre l'input et l'output à chaque point du programme p :

$$IN(p) = \bigcup_{s \in PRED[p]} OUT(s)$$

L'output d'un programme p peut être de types différents :

1. Modification de l'environnement courant :

$$OUT(p) = \langle E, (GEN(\epsilon)(p) \sqcup (\epsilon \backslash KILL(\epsilon)(p))) \rangle \; avec \; E \circ \epsilon = IN(p)$$

2. Création d'un nouvel environnement :
$$OUT(p) = IN(p) \circ \bot$$

3. Réduction du tuple d'envrionnement : on supposera un opérateur $\oplus$ permettant de résumer deux environnements en un seul avec la détection des RC.

$$OUT(p) = E \circ \epsilon' \; avec \; E \circ \epsilon \circ \epsilon_t = IN(p) \; et \; \epsilon' = \epsilon \oplus \epsilon_t$$

8.2.5 Exemples

Détaillons maintenant les opérations GEN et KILL pour l'assignation d'une variable à une constante dans un environnement $\epsilon = (\phi_{RA}, \phi_{RAc}, \phi_{WA}, \phi_{WAc}, \phi_{RC}, \phi_{RCc})$. Encore une fois, nous prenons comme hypothèse que la variable n'a pas été locked auparavant.

- Pour x=c où $c \in \mathbb{Z} \; \wedge \; x \notin \phi_{RC} \; \wedge \; x \notin \phi_{RAc} \; \wedge \; x \notin \phi_{WAc} \; \wedge \; x \notin \phi_{RCc}$

$$\begin{aligned} GEN(\epsilon)(x = c) &= (\varnothing, \varnothing, \{x\}, \varnothing, \varnothing, \varnothing) \\ KILL(\epsilon)(x = c) &= (\{x\}, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing) \quad \text{si } x \in \phi_{RA} \\ KILL(\epsilon)(x = c) &= (\varnothing, \varnothing, \{x\}, \varnothing, \varnothing, \varnothing) \quad \text{si } x \in \phi_{WA} \end{aligned}$$

- Pour x=c où $c \in \mathbb{Z} \; \wedge \; (x \in \phi_{RC} \; \vee \; x \in \phi_{RAc} \; \vee \; x \in \phi_{WAc} \; \vee \; x \in \phi_{RCc})$

$$\begin{aligned} GEN(\epsilon)(\epsilon)(x = c) &= (\varnothing, \varnothing, \varnothing, \varnothing, \{x\}, \varnothing) \\ KILL(\epsilon)(x = c) &= (\{x\}, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing) \quad \text{si } x \in \phi_{RA} \\ KILL(\epsilon)(x = c) &= (\varnothing, \varnothing, \{x\}, \varnothing, \varnothing, \varnothing) \quad \text{si } x \in \phi_{WA} \\ KILL(\epsilon)(x = c) &= (\varnothing, \varnothing, \varnothing, \varnothing, \{x\}, \varnothing) \quad \text{si } x \in \phi_{RC} \end{aligned}$$

Afin de ne pas rendre la formulation trop lourde, nous analysons pour la lecture le cas d'une instruction conditionnelle while :

- Pour *while (x==c)* où $c \in \mathbb{Z} \; \wedge \; (x \notin \phi_{RC} \; \wedge \; x \notin \phi_{WA}) \; \wedge \; (x \notin \phi_{RCc} \; \vee \; x \notin \phi_{WAc})$

$$\begin{aligned} GEN(\epsilon)(while(x == c)) &= (\{x\}, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing) \\ KILL(\epsilon)(while(x == c)) &= (\{x\}, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing) \quad \text{si } x \in \phi_{RA} \end{aligned}$$

- Pour *while (x==c)* où $c \in \mathbb{Z} \; \wedge \; (x \in \phi_{RC} \; \vee \; x \in \phi_{WA}) \; \wedge \; (x \notin \phi_{RCc} \; \vee \; x \notin \phi_{WAc})$

$$\begin{aligned} GEN(\epsilon)(while(x == c)) &= \bot \\ KILL(\epsilon)(while(x == c)) &= \bot \end{aligned}$$

- Pour *while (x==c)* où $c \in \mathbb{Z} \; \wedge \; (x \in \phi_{RCc} \; \vee \; x \in \phi_{WAc})$

$$\begin{aligned} GEN(\epsilon)(while(x == c)) &= (\varnothing, \varnothing, \varnothing, \varnothing, \{x\}, \varnothing) \\ KILL(\epsilon)(while(x == c)) &= (\varnothing, \varnothing, \{x\}, \varnothing, \varnothing, \varnothing) \quad \text{si } x \in \phi_{WA} \\ KILL(\epsilon)(while(x == c)) &= (\varnothing, \varnothing, \varnothing, \varnothing, \{x\}, \varnothing) \quad \text{si } x \in \phi_{RC} \end{aligned}$$

Ainsi, il est possible de modéliser l'analyse statique mise en place avec le framework GEN/KILL avec néanmoins la nécessité d'adapter le framework à ce cas précis en mettant en place une stack d'environnements contenant chacun une suite d'ensemble que les opérations GEN et KILL vont pouvoir modifier.

9 Analyse forward

L'analyse mise en place est une analyse **forward** : elle parcourt le flux naturel du Control Flow Graph. Le choix a été fait de mettre en place une analyse forward car cela permettait réellement de savoir à tout moment du programme si un thread s'exécutait en parallèle d'une portion de code. Faire une analyse backward aurait été plus complexe, car il est possible de réaliser à la fin de l'analyse uniquement, qu'un autre thread a pu s'exécuter en parallèle du code précédemment analyser. Il aurait fallu mettre en place des mécanismes pour reparcourir les instructions déjà analysées ou bien effectuer une analyse prenant en compte plusieurs perspectives (une valeur abstraite si un thread s'exécute en parallèle et une valeur abstraite si aucun thread ne s'exécute en parallèle). C'est pourquoi, mettre en place une analyse forward paraissait plus évident et pertinent dans ce cas-ci.

Toutefois, mettre en place une analyse forward, pour un programme mettant en place des mécanismes de concurrence, n'est pas si évident, car l'ordre d'exécution des instructions des différents threads est indéterminé. Pour éviter le problème d'avoir une analyse aléatoire et pour pouvoir conserver un flux d'exécution séquentielle, les nouveaux threads lancés sont considérés comme des appels à des fonctions. On va d'abord analyser les accès aux variables partagées au sein du nouveau thread avant de continuer l'analyse du thread l'ayant créé. Ainsi, l'analyse se fait sur un CFG séquentiel et une simple analyse forward peut être mise en place. L'exemple ci-dessous illustre comment un programme est transformé en CFG :

```
1  gVar x;
2  gVar y;
3
4  function main(){
5      thread {
6          x = 1;
7          thread {
8          if (y==1){
9              x=2;
10             }
11         else{}
12         }
13         y= x;
14     }
15 }
```

```
class com.SmallConcurrency.cfg.graph.EntryBlock
class com.SmallConcurrency.cfg.graph.GlobalVarDecl
class com.SmallConcurrency.cfg.graph.GlobalVarDecl
class com.SmallConcurrency.cfg.graph.Function
class com.SmallConcurrency.cfg.graph.Thread
class com.SmallConcurrency.cfg.graph.Assignment
class com.SmallConcurrency.cfg.graph.Thread
class com.SmallConcurrency.cfg.graph.IfElse
class com.SmallConcurrency.cfg.graph.Assignment
class com.SmallConcurrency.cfg.graph.EndIf
class com.SmallConcurrency.cfg.graph.EndThread
class com.SmallConcurrency.cfg.graph.Assignment
class com.SmallConcurrency.cfg.graph.EndThread
class com.SmallConcurrency.cfg.graph.EndFunction
class com.SmallConcurrency.cfg.graph.EndBlock
```

10 Exemple de programme d'entrée

①	gVar x ;	$\langle(\phi,\phi_c)\rangle \mapsto (\phi[x \mapsto NA)], \phi_c[x \mapsto NA)])$
②	gVar y ;	$\langle(\phi,\phi_c)\rangle \mapsto (\phi[y \mapsto NA)], \phi_c[x \mapsto NA)])$
③	gVar z ;	$\langle(\phi,\phi_c)\rangle \mapsto (\phi[z \mapsto NA)], \phi_c[x \mapsto NA)])$
④	gVar w ;	$\langle(\phi,\phi_c)\rangle \mapsto (\phi[w \mapsto NA)], \phi_c[x \mapsto NA)])$
㉕	function f() {	$\langle(\phi,\phi_c)\rangle \mapsto \langle(\phi,\phi_c)\rangle$
㉖	return x ;	$\langle(\phi,\phi_c)\rangle \mapsto \langle(\phi,\phi_c)\rangle$ car $(\phi(x) = RC)$
	}	
⑤	function main(){	$\langle(\phi,\phi_c)\rangle \mapsto \langle(\phi,\phi_c)\rangle$
⑥	x = 1 ;	$\langle(\phi,\phi_c)\rangle \mapsto (\phi[x \mapsto WA)], \phi_c)$
⑦	y = 2 ;	$\langle(\phi,\phi_c)\rangle \mapsto (\phi[y \mapsto WA)], \phi_c)$
⑧	z = 3 ;	$\langle(\phi,\phi_c)\rangle \mapsto (\phi[z \mapsto WA)], \phi_c)$
⑨	w = 4 ;	$\langle(\phi,\phi_c)\rangle \mapsto (\phi[w \mapsto WA)], \phi_c)$
⑩	thread {	$\langle(\phi,\phi_c)\rangle \mapsto \langle(\phi,\phi_c),(\phi_{t1},\phi_{t1_c})\rangle$ où $(\phi_{t1}\|\forall v \in \phi : \phi_{t1}(v) = NA) \ \wedge \ (\phi_{t1_c}\|\forall v \in \phi : \phi_{t1_c}(v) = NA)$
⑪	while(x < 10){	$\langle(\phi,\phi_c),(\phi_{t1},\phi_{t1_c})\rangle \mapsto \langle(\phi,\phi_c),(\phi_{t1}[x \mapsto RA],\phi_{t1_c})\rangle$
⑫	x = x + z ;	$\langle(\phi,\phi_c),(\phi_{t1},\phi_{t1_c})\rangle \mapsto \langle(\phi,\phi_c),(\phi_{t1}[x \mapsto WA, z \mapsto RA],\phi_{t1_c})\rangle$
	}	
⑬	thread {	$\langle(\phi,\phi_c),(\phi_{t1},\phi_{t1_c})\rangle \mapsto \langle(\phi,\phi_c),(\phi_{t1},\phi_{t1_c}),(\phi_{t2},\phi_{t2_c})\rangle$ cfr ⑩
⑭	if (y == 2)	$\langle(\phi,\phi_c),(\phi_{t1},\phi_{t1_c}),(\phi_{t2},\phi_{t2_c})\rangle \mapsto \langle(\phi,\phi_c),(\phi_{t1},\phi_{t1_c}),(\phi_{t2}[y \mapsto RA],\phi_{t2_c})\rangle$
⑮	z = y+ 1 ;	$\langle(\phi,\phi_c),(\phi_{t1},\phi_{t1_c}),(\phi_{t2},\phi_{t2_c})\rangle \mapsto \langle(\phi,\phi_c),(\phi_{t1},\phi_{t1_c}),(\phi_{t2}[z \mapsto WA, y \mapsto RA],\phi_{t2_c})\rangle$
⑯	else	$\langle(\phi,\phi_c),(\phi_{t1},\phi_{t1_c}),(\phi_{t2},\phi_{t2_c})\rangle \mapsto \langle(\phi,\phi_c),(\phi_{t1},\phi_{t1_c}),(\phi_{t2},\phi_{t2_c})\rangle$

⑰	z = y-1 ;	$\langle(\phi,\phi_c),(\phi_{t1},\phi_{t1_c}),(\phi_{t2},\phi_{t2_c})\rangle \mapsto \langle(\phi,\phi_c),(\phi_{t1},\phi_{t1_c}),(\phi_{t2},\phi_{t2_c})\rangle$
⑱	}	$\langle(\phi,\phi_c),(\phi_{t1},\phi_{t1_c}),(\phi_{t2},\phi_{t2_c})\rangle \mapsto \langle(\phi,\phi_c),(\phi_{t1},\phi_{t1_c}[y \mapsto RA, z \mapsto WA])\rangle$
⑲	if (w == 3)	$\langle(\phi,\phi_c),(\phi_{t1},\phi_{t1_c})\rangle \mapsto \langle(\phi,\phi_c),(\phi_{t1}[w \mapsto RA],\phi_{t1_c})\rangle$
⑳	x = z + 1 ;	$\langle(\phi,\phi_c),(\phi_{t1},\phi_{t1_c})\rangle \mapsto \langle(\phi,\phi_c),(\phi_{t1}[z \mapsto RC],\phi_{t1_c})\rangle$
㉑	else	$\langle(\phi,\phi_c),(\phi_{t1},\phi_{t1_c})\rangle \mapsto \langle(\phi,\phi_c),(\phi_{t1},\phi_{t1_c})\rangle$
㉒	x = z-1 ;	$\langle(\phi,\phi_c),(\phi_{t1},\phi_{t1_c})\rangle \mapsto \langle(\phi,\phi_c),(\phi_{t1},\phi_{t1_c})\rangle$ (car $\phi_{t1}(z) = RC$)
㉓	}	$\langle(\phi,\phi_c),(\phi_{t1},\phi_{t1_c})\rangle \mapsto \langle(\phi,\phi_c[x \mapsto WA, y \mapsto RA, z \mapsto RC, w \mapsto RA])\rangle$
㉔	x = f() ;	$\langle(\phi,\phi_c)\rangle \mapsto \langle(\phi[x \mapsto RC],\phi_c)$
㉗	x=5 ;	$\langle(\phi,\phi_c)\rangle \mapsto \langle(\phi,\phi_c)\rangle$ car $(\phi(x) = RC)$
}		

```
The program is semantically correct!
Starting cfg generation...
Finished cfg generation!
2024-01-09 16:04:29.237 [main] WARN  com.SmallConcurrency.staticAnalysis.StaticAnalysisVisitor - Race condition detected on variable z at line 28:
Read may happen during write in another thread
2024-01-09 16:04:29.239 [main] WARN  com.SmallConcurrency.staticAnalysis.StaticAnalysisVisitor - Race condition detected on variable z at line 30:
Read may happen during write in another thread
2024-01-09 16:04:29.239 [main] WARN  com.SmallConcurrency.staticAnalysis.StaticAnalysisVisitor - Race condition detected on variable x at line 33:
    Write may happen during read or write in another thread
2024-01-09 16:04:29.239 [main] WARN  com.SmallConcurrency.staticAnalysis.StaticAnalysisVisitor - Race condition detected on variable x at line 7:
Read may happen during write in another thread
2024-01-09 16:04:29.239 [main] WARN  com.SmallConcurrency.staticAnalysis.StaticAnalysisVisitor - Race condition detected on variable x at line 34:
    Write may happen during read or write in another thread
2024-01-09 16:04:29.239 [main] WARN  com.SmallConcurrency.main.Main - Race condition detected on variable x!
2024-01-09 16:04:29.239 [main] WARN  com.SmallConcurrency.main.Main - Race condition detected on variable z!

Process finished with exit code 0
```

11 Comment exécuter le programme

11.1 Exécution rapide

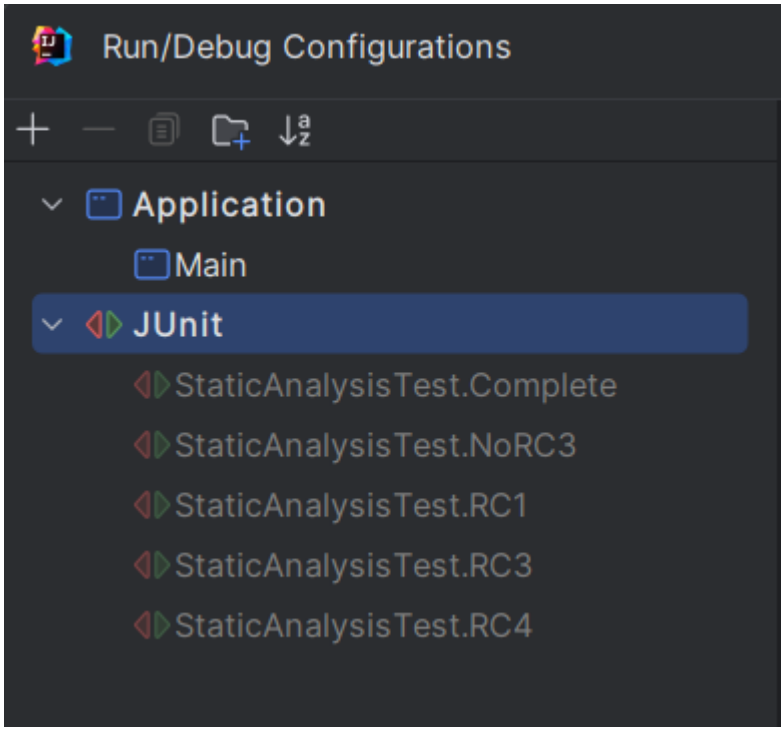
Afin de lancer une exécution rapide du code, sans devoir mettre en place un environnement de développement, une archive JAR est disponible sur le Repository. dans le dossier */bin*. POur lancer l'exécution du programme, insérer la commande suivant dans le terminal pour exécuter les 3 fichiers fournis :

```
java -jar .\smallConcurrencyAnalyzer-1.0-SNAPSHOT-jar-with-dependencies.jar ../src/main/resources/test.smallConcurrent

java -jar .\smallConcurrencyAnalyzer-1.0-SNAPSHOT-jar-with-dependencies.jar ../src/main/resources/Complete.smallConcurrent

java -jar .\smallConcurrencyAnalyzer-1.0-SNAPSHOT-jar-with-dependencies.jar ../src/main/resources/test2.smallConcurrent
```

11.2 Mise en place d'un environnement de développement

POur mettre en place un environnement de développement afin de pouvoir apporter des modifications aux schémas, il faut :

1. Ouvrir le projet dans IntelliJ
2. Exécuter la commande *mvn compile* dans le dossier racine du programme
3. Créer une nouvelle "application" partant de la classe Main du package *com.SmallConcurency.main* en lui fournissant en argument le chemin du fichier à exécuter depuis la root
4. Lancer l'application créée



En plus du code du programme, il est aussi possible d'exécuter les tests mis en place dans le fichier StaticAnalysisTest.java du package *com.SmallConcurency.main* (dans le répertoire test).

12 Conclusion générale du projet

Mettre en place une analyse statique pour détecter l'occurrence de situations de course n'est pas triviale. Plusieurs approches peuvent être abordées pour les détecter. Dans ce rapport, l'approche présentée est basée sur l'enregistrement continu de la nature des accès aux variables partagées du programme (lecture/écriture).

Le principal inconvénient de cette approche est que l'abstraction de la valeur réel ne permet pas, dans le cas d'un branchement conditionnel, de déterminer quel chemin prendre. Ainsi, les deux chemins sont explorés de manière préventive, menant à une analyse complète, mais pas sound.

L'utilisation du framework ANTLR a permis de mettre en place aisément la grammaire du langage SmallConcurrency en générant directement un parser et un visiteur de base de l'arbre abstrait d'un programme. La mise en place de formules mathématiques a pu guider le développement d'une analyse robuste et mathématiquement correct directement dans le visiteur du CFG (généré préalablement).

12.1 Amélioration future & Évaluation des résultats de l'analyse statique

Le projet ici présenté ouvre de plus amples horizons qui mériteraient qu'une recherche plus poussée soit effectuée :

1. Permettre le logging précis des lignes de code des RC entre des threads en séries (ce qui implique une modification de la logique des fonctions de flux)
2. Détailler plus précisément la fonction $\oplus$ utilisée dans le cadre du framework GEN/KILL
3. Évaluer l'introduction d'une nouvelle instruction Await (permettant d'attendre la fin d'un thread) sur les fonctions de flux et la modélisation de la solution.
4. Appliquer cette analyse à des autres langages de programmation plus complexe.

L'analyse statique permet de détecter **toutes** les situations de course qui peuvent apparaitre au sein d'un programme. Le fait qu'elle puisse par moment détecter une RC dans un branchement conditionnel jamais exécuté pourrait apparaitre comme un inconvénient de l'analyse. Toutefois, pouvoir repérer ce genre de cas n'est en fait que bénéfique pour le programmeur : si le code venait à être modifié, l'erreur aurait déjà été repérée auparavant.

Références

[1] Baledung : What is a race condition ? URL `https://www.baeldung.com/cs/race-conditions`.

[2] *INFOM227 : Syllabus : program analysis and abstract interpretation.*