Analyses statiques pour l'observabilité de l'ordre d'évaluation en OCaml

Mots-clés :	Analyse statique, sémantique, non-interférence, dépendances, pureté
Encadrants :	Benoît Montagu, Thomas Jensen
Contacts :	`Benoit.Montagu@inria.fr`, `Thomas.Jensen@inria.fr`
Localisation :	Équipe Épicure, Irisa-Inria, Rennes

Résumé Ce sujet de stage de Master a pour but d'utiliser des techniques d'analyse statique afin de détecter des interférences dans des programmes OCaml. Un objectif pratique est d'aider les développeurs OCaml en les avertissant qu'un programme OCaml est sensible à l'ordre dans lequel sont évaluées les sous-expressions qui le composent. Ces résultats pourront être intégrés à l'analyseur statique Salto [1].

∼ ❄ ∼

L'initialisation de valeurs mutables et leur modification, l'impression de messages sur la sortie standard, le lancement d'*exceptions*, et plus généralement les *effets*, peuvent rendre *observable* l'ordre d'exécution des sous-parties d'un programme : changer l'ordre d'exécution peut conduire à des résultats différents. C'est notamment le cas du programme OCaml [1] suivant :

```
let r = ref 1 in
(fun () () -> !r) (r := !r * 2) (r := !r + 1)
```

qui renvoie la valeur `3` ou la valeur `4`, suivant que l'ordre d'évaluation des arguments d'une fonction s'effectue de la gauche vers la droite, ou de la droite vers la gauche.

L'ordre d'évaluation restant non spécifié en OCaml, les programmeurs ne doivent pas se reposer sur l'ordre d'évaluation actuellement implémenté par le compilateur. Ils sont donc invités à rendre l'ordre d'évaluation explicite en introduisant des liaisons intermédiaires de la forme `let x = ... in ...` lorsque c'est nécessaire.

Le but de ce stage est de déterminer dans quelles conditions l'ordre d'évaluation risque d'influer sur le comportement d'un programme, et de concevoir une analyse statique de programmes qui détecte ces cas, et informe le programmeur que le comportement de son programme risque d'être ambigu.

On cherchera en particulier à ne pas produire d'avertissement dans les cas bénins. Par exemple, deux sous-expressions qui lisent ou modifient des données partagées *disjointes*, ou

1. `https://salto.gitlabpages.inria.fr/`

bien deux sous-expressions *pures*, c'est-à-dire dont l'exécution n'effectuera aucun effet, peuvent s'exécuter dans un ordre ou dans un autre sans changer le comportement du programme.

Pour concevoir cette analyse, on s'intéressera notamment aux analyses de flux d'information [4, 3]. Ces analyses sont particulièrement pertinentes, car elles déterminent quels paramètres peuvent influencer quelles parties de l'état d'un programme : elles permettent de déterminer si deux parties d'un programme peuvent *interférer*, et sur quelles parties des données. On étudiera aussi du système de types et d'effets pour la dépendance de l'ordre d'évaluation [2], qui a été utilisé pour tester le compilateur OCaml en générant aléatoirement des programmes qui exhibent des comportements différents selon l'ordre choisi.

Références

[1] [Ver.log.] Xavier Leroy et al., *The OCaml system* version 4.14, 28 mars 2022. Inria. URL : https://ocaml.org/, VCS : https://github.com/ocaml/ocaml.

[2] Jan Midtgaard et al. « Effect-driven QuickChecking of compilers ». In : (août 2017), p. 1-23. DOI : 10.1145/3110259. URL : https://doi.org/10.1145/3110259.

[3] François Pottier et Vincent Simonet. « Information flow inference for ML ». In : *ACM Transactions on Programming Languages and Systems* 25.1 (jan. 2003), p. 117-158. DOI : 10.1145/596980.596983. URL : https://doi.org/10.1145/596980.596983.

[4] Dennis Volpano et Geoffrey Smith. « A type-based approach to program security ». In : *TAPSOFT '97 : Theory and Practice of Software Development*. Sous la dir. de Michel Bidoit et Max Dauchet. Berlin, Heidelberg : Springer Berlin Heidelberg, 1997, p. 607-621. ISBN : 978-3-540-68517-3.