Définition du sous-langage SecurOCaml

OCamlPro, SafeRiver

1 Motivation et objectifs

Nous voulons ici définir un langage qui soit un sous-ensemble du langage OCaml, restreint de manière à être plus facile à auditer pour validation de critères de sécurité, et plus facile à analyser pour des outils automatisés.

Ce document est complémentaire des recommandations émises lors de l'étude LaFoSec et mises à jour dans ce même projet. Alors que les recommandations sont applicables pour tout programme devant être utilisé dans un contexte de sécurité, ce document décrit des contraintes visant les programmes avec de fortes contraintes de fiabilité, et un effort particulier est fait pour faciliter le travail des auditeurs sur ces programmes.

2 Lignes directives

2.1 Déterminisme

Une partie du langage OCaml est volontairement laissée non-spécifiée, en particulier l'ordre d'évaluation des sous-expressions d'une même expression. Les cas les plus courants sont l'ordre d'évaluation des arguments d'une fonction à plusieurs arguments, des différents éléments d'un couple ou d'une valeur de type enregistrement (*record*).

OCaml n'étant pas un langage purement fonctionnel, un changement d'ordre d'évaluation peut correspondre à un changement observable de comportement du programme, ce qui nuit au travail d'audition (manuelle comme automatisée).

2.2 Flux prévisible

Un des grands inconvénients pour l'analyse des langages fonctionnels d'ordre supérieur tels que OCaml, est qu'il est souvent difficile, que ce soit pour un outil logiciel ou pour un être humain, de construire précisément le graphe de flux de contrôle.

Dans ce document, nous mettrons des contraintes sur les valeurs fonctionnelles qui peuvent être manipulées par le programme, tout en essayant de préserver les avantages liés à la programmation fonctionnelle d'ordre supérieur.

2.3 Fonctionnalités fiables

Le langage OCaml a beaucoup évolué depuis sa création, et de nombreuses fonctionnalités ont été rajoutées pour étendre les possibilités disponibles aux développeurs.

Cependant, toutes ces fonctionnalités ne sont pas égales en termes de fiabilité : les plus récentes sont moins bien testées, et même parmi les plus anciennes certaines sont connues pour être une source régulière de problèmes dans le compilateur à cause de la complexité du code associé.

3 Présentation détaillée

3.1 Options de compilation

Ce langage étant un sous-ensemble de OCaml, il est conçu pour être utilisé avec les compilateurs OCaml de la distribution officielle. Nous mettons ceci dit des restrictions sur les paramètres de compilation utilisés :

— Le compilateur permet d'activer ou de désactiver un certain nombre d'avertissements, voire de transformer ces avertissements en erreurs. Dans le cadre de ce sous-langage, nous supposons que les programmes seront compilés en activant les avertissement suivants en tant qu'erreurs : `3-6 8-12 14 16 20 21 23-50 60`

— Les options sur le typage sont les plus conservatives : option `-rectypes` désactivée, option `-principal` activée.

— Les pré-processeurs ne sont pas considérés comme faisant partie du langage. Dans le cas où des options `-pp` ou `-ppx` sont nécessaires, utiliser les fonctionnalités du compilateur pour produire un fichier `.ml` à partir des fichiers originaux. Ce sont ces fichiers produits qui seront considérés par les outils d'analyse et les auditeurs.

3.2 Restrictions syntaxiques sur le langage

3.2.1 Classes et objets

Les objets en OCaml font partie du langage depuis plus de 20 ans, et sont utilisés dans de nombreux logiciels de diverses envergures et champs d'application.

Malgré cela, nous avons choisi de complètement retirer les objets et classes du sous-langage défini dans ce document. Nous le justifions par les deux raisons suivantes :

— Les objets, et particulièrement les classes et l'héritage, permettent d'écrire des programmes avec des flux de contrôle difficiles à analyser à cause de la résolution dynamique des méthodes. Cela compromet la prévisibilité du flux de contrôle, et les autres fonctionnalités liées à la programmation objet peuvent être émulées soit par des types enregistrements (pour les fonctionnalités les plus simples), soit par des modules de première classe.

— Les fonctionnalités liées aux objets correspondent à des fragments de code à la fois spécifiques aux objets, et relativement compliqués (à la fois au niveau du typage et de la génération de code). Sans remettre en cause la sûreté du code, nous considérons que le coût de traiter correctement les objets dans les outils automatiques d'analyse justifie de ne pas l'accepter dans le langage.

3.2.2 Fonctions externes

Les fonctions externes (définies par le mot-clé *external*), permettent d'appeler du code en C, ou d'accéder à des fonctionnalités spécifiques du compilateur.

Le deuxième cas est interdit dans notre sous-langage (les fonctionnalités utiles sont déjà disponibles dans la bibliothèque standard), le premier cas est autorisé sous contraintes : pour faciliter le travail des auditeurs et des logiciels d'analyse, les fonctions externes doivent apparaître uniquement dans des unités de compilation dédiées et documentées extensivement.

3.2.3 Modules récursifs

Les modules récursifs sont interdits dans le sous-langage. Si les cas simples sont bien testés et relativement compréhensibles, les détails d'implémentation rendent l'analyse compliquée dans le cas générique, et la distinction entre un cas simple et un cas compliqué n'est pas triviale.

3.2.4 Variables ignorables

Les variables dont le nom commence par un caractère underscore reçoivent un traitement spécifiques par le compilateur : elles sont ignorées par les vérifications d'utilisation effective. Pour cette raison, dans ce sous-langage les noms de variables commençant par un underscore ne peuvent pas apparaître dans les expressions du langage, mais seulement dans les *patterns*.

3.2.5 Rattrapage d'exceptions génériques

Les constructions de type `try exp1 with _ -> exp2` sont interdites. Les exceptions peuvent être utilisées pour diverses raisons (comportements rares ou innattendus, flux de contrôle particuliers), mais dans tous les cas elles doivent être rattrapées spécifiquement, de préférence le plus tôt possible.

3.3 Restrictions sémantiques

3.3.1 Définition préliminaire : Effets et pureté des expressions

Les *effets* (ou effets de bord) en OCaml correspondent aux opérations dont l'évaluation est observable par le reste du programme.

Il s'agit principalement :

— Des mises à jour de champs modifiables d'une valeur de type enregistrement, soit par l'opération `<-`, soit par l'opérateur `:=` défini par la bibliothèque standard. Les mises à jour de champs d'un tableau (*array*) ou d'une chaîne de caractères (de type `bytes` dans les versions récentes d'OCaml) entrent aussi dans cette catégorie.
— Des exceptions lancées lors de l'évaluation d'une expression.
— Des appels à des fonctions externes (déclarées avec le mot-clé `external`).

Le langage OCaml ne pose pas de restrictions sur l'utilisation de ces fonctionnalités, mais le sous-langage défini dans ce document fera référence aux expressions qui n'utilisent pas ces fonctionnalités comme étant *pures*, et à celles qui les utilisent comme *impures*.

Les lectures de champs modifiables (ou de tableaux, ou de chaînes de caractères modifiables) ne sont pas a proprement parler des effets, puisqu'ils sont invisibles pour le reste du programme, mais leur utilisation dans des contextes où seules les expressions pures sont autorisées est une potentielle source de problèmes et idéalement devrait être systématiquement justifiée.

Le cas des expressions contenant des appels de fonctions est un peu particulier : comme appeler une fonction on peut théoriquement évaluer une expression impure, l'appel de fonction sera en général considéré comme impur. Mais un des avantages de ce sous-langage étant de rendre le flux de contrôle prévisible, nous autorisons à considérer comme pur un appel de fonction où la fonction appelée est connue, et le code correspondant est lui-même pur.

Il faut également noter que toute évaluation d'une expression pouvant allouer des données sur le tas peut potentiellement déclencher des effets de bord, soit en réaction à un signal POSIX, soit en déclenchant le ramasse-miettes qui lui-même déclanche l'exécution de finaliseurs. Dans le cadre du sous-langage défini ici, nous considèrerons les allocations comme pures, car nous interdisons les finaliseurs et nous contraignons fortement les fonctions pouvant être associées aux signaux.

3.3.2 Évaluation paresseuse de valeurs impures

L'évaluation paresseuse (via le mot-clé `lazy`) d'expressions impures est interdite.

La raison est double : d'une part parce que le filtrage de motifs paresseux peut donner lieu à de l'exécution du code correspondant, et qu'il est important que le filtrage soit considéré comme pur. D'autre part, l'évaluation paresseuse introduit des flux de contrôle non triviaux, et garantir la pureté des expressions évaluées permet d'apporter des garanties utiles aux auditeurs et outils d'analyse sans se priver de la fonctionnalité complètement.

3.3.3 Filtrage de valeurs modifiables

Il est interdit de filtrer sur une valeur modifiable. Filtrer sur les champs non modifiables d'un enregistrement contenant des champs modifiables est accepté.

Pour contourner cette restriction, une possibilité est de lier le contenu des champs modifiables dans une variable, puis de filtrer sur cette variable.

Dans le langage OCaml complet, on peut construire des exemples qui donnent des résultats surprenants, voire des erreurs à l'exécution dans certains cas pour certaines versions du compilateur. Les restrictions imposées dans le sous-langage défini ici suffisent à mitiger la plupart de ces problèmes, mais imposer un peu de code en plus autour de certains filtrages est un faible prix à payer pour la sûreté gagnée.

3.3.4 Pureté des guardes lors du filtrage

Les guardes (clauses *when*) du filtrage sont restreintes aux expressions pures.

En plus des problèmes déjà mitigés par la restriction précédente, imposer des valeurs pures dans les guardes permet de garantir que l'ordre d'évaluation de deux clauses incompatibles ne change pas le résultat obtenu.

3.3.5 Désambigüation de constructeurs

Les versions récentes d'OCaml permettent d'utiliser des version non préfixées de constructeurs, sous condition qu'une version préfixée du constructeur soit connue par l'environnement de typage et qu'il n'y ait pas d'ambigüité. Pour faciliter l'analyse et l'audition de code, l'utilisation de cette fonctionnalité est interdite, et tout constructeur doit être préfixé si nécessaire.

3.3.6 Foncteurs applicatifs et effets

À l'exception des foncteurs prenant comme argument un module de signature vide, tous les foncteurs en OCaml sont considérés comme applicatifs : deux applications d'un même foncteur à un même module donne deux modules considérés comme équivalents. En pratique, cela interdit à un foncteur applicatif de générer des types lors de son application.

Dans ce sous-langage, pour faciliter le travail des auditeurs et outils d'analyse, nous considérons que deux applications d'un même foncteur à un même module doivent être fonctionnellement équivalentes, et donc nous interdisons l'utilisation d'expressions impures ou la lecture de valeurs modifiables dans la partie du foncteur correspondant à l'initialisation du module résultat.

3.3.7 Comparaison polymorphe

Le langage OCaml expose dans sa librairie standard des fonctions de comparaison polymorphes. Si leur application sur les différents types de base est acceptable, certains types de données ne sont pas compatibles avec ces fonctions (et le typage du langage ne permet pas de le vérifier à la compilation), et leur usage en temps que fonctions polymorphes est interdit.

3.3.8 Valeurs récursives

Le langage OCaml permet de définir, sous certaines conditions, des valeurs récursives. Dans le cas de valeurs non fonctionnelles, cela permet par exemple de créer des listes cycliques (infinies). Or si les fonctions récursives sont bien supportées, les définitions de valeurs récursives non fonctionnelles passent par des heuristiques du compilateur relativement compliquées, dans lesquelles des problèmes ont été trouvés (et réglés) encore très récemment.

Ce sous-langage, par prudence, n'autorise donc dans les définitions récursives que des définitions de fonctions et des expressions évaluées paresseusement.

3.3.9 Réutilisation de noms de variables

Il est autorisé en OCaml de réutiliser un nom de variable déjà existant pour définir une nouvelle variable. En cas de conflit, la définition la plus récente (ou la plus profonde en termes d'arbre syntaxique) prend priorité pour la résolution des variables.

Dans ce sous-langage, pour faciliter la tâche des auditeurs humains, la réutilisation de variables (dans un contexte où la variable d'origine est valide) est interdite.

3.3.10 Ordre d'évaluation non déterministe

Certaines expressions en OCaml sont constituées de plusieurs sous-expressions qui doivent toutes être évaluées. Dans ces cas, il est officiellement documenté que l'ordre d'évaluation des sous-expressions est non documenté. Ces expressions sont l'application de fonction, et la construction de couples, d'enregistrements ou de types somme à plusieurs arguments.

Dans ce sous-langage, nous interdisons l'utilisation d'expressions impures dans ces contextes, et nous recommandons fortement d'évaluer au préalable toute expression non triviale, en stockant le résultat dans une variable, et de n'utiliser que des variables ou autres expressions simples dans les sous-expressions.

3.3.11 Types non injectifs

Il est possible en OCaml de définir un alias de type avec des paramètres qui ne sont pas utilisés dans la définition. Un exemple type est la définition suivante :

```
type 'a t = int
```

Ces définitions, dites non injectives, ont été impliquées dans un certain nombre de problèmes dans le compilateur et sont donc interdites dans le sous-langage.

3.3.12 Types ouverts

Les types ouverts (autres que le type prédéfini des exceptions) sont interdits dans le sous-langage.

Sans être particulièrement dangereux, ils rendent le code plus difficile à comprendre et analyser, pour un bénéfice limité en terme d'expressivité.

3.3.13 Variants polymorphes

Les variants polymorphes sont interdits dans le sous-langage.

Tous comme les types ouverts, ils rendent le code plus difficile à comprendre en général, et les interdire force les développeurs à donner plus de définitions de types, ce qui aide le travail d'analyse.

3.3.14 Types algébriques généralisés (GADT)

Les types algébriques généralisés, présents dans OCaml depuis la version 4.0, sont interdits dans le sous-langage.

Les algorithmes de typage qui garantissent la correction du compilateur ont été rendus particulièrement compliqués par l'introduction des GADT, et des erreurs allant des erreurs internes au compilateur à des programmes erronés générés par le compilateur sont encore trouvées et corrigées dans les versions les plus récentes d'OCaml.

3.3.15 Execution de code à l'initialisation

Une unité de compilation peut exécuter du code lors de son initialisation. À l'exception de la dernière unité de compilation (qui va devoir exécuter le programme complet), il est fortement recommandé de ne pas exécuter de code à l'initialisation d'un module.

En effet, suivant les outils utilisés pour construire l'exécutable final, des différences peuvent apparaître sur l'ordre dans lequel les unités de compilation sont initialisées. En particulier, il est particulièrement dangereux de modifier les données présentes dans un autre module (dans le cas d'une initialisation différée, par exemple) car rien ne garantit que l'initialisation différée sera effectuée avant l'utilisation effective du module concerné.

En revanche, il existe des cas où il est nécessaire d'exécuter du code à l'initialisation du module, par exemple lorsque les fonctionnalités exposées à travers l'interface ont des implémentations différentes suivant le contexte (système d'exploitation, bytecode ou natif, ...). Il ne s'agit donc pas d'une interdiction.

3.4 Fonctionnalités de la bibiothèque standard à éviter

Les recommandations de l'étude LaFoSec s'appliquent bien sûr à ce sous-langage.

Pour résumer les points principaux, les modules `Obj` et `Marshal` permettent de contourner les garanties liées au typage, et sont à proscrire complètement. Les modules `Digest` et `Hashtbl` doivent être utilisés avec précaution dans le cadre d'application sécurisées.

4 Recommandations supplémentaires

4.1 Exceptions

Les exceptions en OCaml ne sont pas reflétées dans le type des expressions et fonctions pouvant les lancer. Pour faciliter le travail d'audition, il est donc conseillé de ne pas les utiliser pour exprimer des résultats innattendus ou invalides et de renvoyer plutôt des valeurs de type `option` ou `result` (ce dernier n'est présent dans la librairie standard que depuis OCaml 4.03, mais peut être défini manuellement pour n'importe quelle version).

Dans le cas d'une utilisation des exceptions pour exprimer un flux de contrôle (par exemple pour sortir d'une boucle immédiatement), il est conseillé d'utiliser des exceptions définies spécifiquement pour chaque utilisation, et de s'assurer qu'elles ne peuvent pas s'échapper vers le reste du programme.

Une exception peut être faite pour la primitive `assert` : son usage habituel est d'exprimer des propriétés que le compilateur (via le système de types) ne peut pas prouver, mais qui sont nécessaires à la correction du programme. Leur présence (avec éventuellement des justifications sous forme de commentaires) facilite le travail d'audition. Cela implique bien sûr que le code ne rattrape jamais les exceptions `Assert_failure`, y compris à travers des rattrapages génériques.

4.2 Ordre supérieur, appels indirects, applications partielles

Le langage OCaml permet de manipuler des fonctions comme des valeurs de première classe, ce qui donne au développeur plus de liberté dans la façon d'écrire un programme donné.

Toutes les façons d'écrire un programme ne sont pas égales pour un auditeur, et particulièrement pour les logiciels d'analyse. Pour ne pas compliquer inutilement l'analyse par des outils automatisés, les pratiques suivantes sont recommandées :

- Éviter au maximum les applications partielles de fonctions. Préférer définir une fonction (éventuellement anonyme) avec des arguments explicites et une application complète de la fonction d'origine (procédure appelée η-expansion).
- Éviter autant que possible de stocker des fonctions dans des structures de données de type couple, enregistrement, ou constructeur d'un type variant.
- Éviter les appels indirects (applications de fonctions dont la fonction n'est pas associée à une définition de fonction). Dans le cas de fonctions d'ordre supérieur simples (telles que `List.map`), la condition peut être relâchée à condition que la fonction d'ordre supérieur elle-même soit appelée directement avec des arguments définis.