D4.3 Utilisation des technologies SecurOCaml sur les cas d'études

David Maison Virgile Prevosto

Abstract

Ce document présente les résultats de l'évaluation des outils développés au sein du projet sur les bases de code du CEA Tech List et de TrustInSoft.

Introduction

Au cours du projet, différents outils, notamment ceux développés au sein de SecurOCaml, ont été évalués sur les bases de code du CEA Tech List et de TrustInSoft, afin de voir dans quelle mesure ils permettraient d'améliorer la robustesse de ces codes. Ce rapport présente les résultats de ces évaluations. Les outils concernés sont les suivants:

- Outils SecurOCaml:
  - `dead-code-analyzer`
  - `generics`
  - `ocp-analyzer`
  - `ocp-lint`
  - `OCaml`
- Outils externes
  - `crowbar`
  - `ocamllint`

Analyseur de code mort - `dead_code_analyzer`

Présentation de l'outil

`dead_code_analyzer` est un analyseur de code mort, i.e. de fonctions exportées par un module toplevel, mais jamais utilisées dans l'application. Son intérêt principal est donc de repérer des fonctions obsolètes et les supprimer pour d'une part faciliter la maintenance du code et d'autre part éviter l'accumulation de portions de code pas ou mal testées, et dont l'utilisation pourrait donc avoir des

conséquences imprévues. Plus généralement, il s'agit d'obtenir une meilleure maîtrise de la base de code de l'application.

Expérimentations

Une compilation de la version de développement de `dead_code_analyzer`, avec la version 4.05.0+trunk du compilateur OCaml, n'a pas permis de passer les tests (`make check`) de l'outil: aucun code mort n'était rapporté. La situation est la même avec le paquet `opam` de l'outil en utilisant la version `4.04.0` du compilateur OCaml. En revanche, le paquet `opam` compilé avec la version `4.03.0` d'OCaml indique bien les éléments de code non utilisés des tests de `dead_code_analyzer`. Les expérimentations sur la base de code de Frama-C ont donc été menées sur la base de cette configuration, Frama-C étant compatible avec tout compilateur OCaml à partir de la version `4.02.3`.

L'outil a été utilisé avec la ligne de commande suivante:

```
dead_code_analyzer --internal -A src
```

Dans cette configuration, il met environ 20 secondes sur un ordinateur portable pour analyser la base de code de Frama-C, ce qui est parfaitement raisonnable. Les résultats obtenus sont les suivants:

- 1400 valeurs globales non utilisées
- 500 méthodes non utilisées
- 150 constructeurs ou champs d'enregistrement non utilisés

Ces informations seront a priori très utiles pour piloter des opérations de refactoring de la base de code de Frama-C. Cependant, quelques petits détails ajoutent un certain bruit, qui diminue la pertinence des résultats.

Tout d'abord, un certain nombre de fonctions de Frama-C sont exportées pour permettre leur utilisation par des plug-ins développés en dehors de la plateforme. Il ne semble pas possible de préciser que certaines fonctions doivent être considérées comme des points d'entrée (et donc ne pas être rapportées comme inutilisées). Comme certaines d'entre elles sont utilisées dans le module où elles sont définies en plus d'être exportées, l'option `--internal` a été utilisée, mais sans changement apparent dans la sortie.

Par ailleurs, lorsque la signature d'un module utilise `include`, et qu'une des valeurs contenue dans la signature incluse n'est pas utilisée, la localisation est correspond à la signature incluse. Le nom du symbole étant celui correspondant au module complet, il est possible de retrouver ce dernier, mais au prix d'une indirection supplémentaire.

Enfin, un certain nombre de méthodes déclarées comme non utilisées se trouvent dans l'interface graphique de Frama-C, et sont définies dans des classes servant à construire des objets `lablgtk`. Ces méthodes sont donc en fait utilisées par la bibliothèque sur laquelle s'appuie Frama-C, et il semble difficile de les masquer.

Conclusion

Hormis les problèmes avec les versions les plus récentes d'OCaml, `dead_code_analyzer` est un outil efficace et utile. La sortie est claire et pointe facilement (sauf pour le point ci-dessus) vers les endroits adéquats. Toutefois, une fonctionnalité utile serait de fournir un résumé (nombre de symbole non-utilisés pour chaque catégorie) de l'analyse effectuée.

Vérificateur de données structurées : Generic

Introduction

Generic est une bibliothèque de programmation générique pour OCaml avec des fonctionnalités de sérialisation et désérialisation typées qui vérifient à l'exécution si une donnée sérialisée a bien le type que celui qui lui est associée lors du typage. Generic est disponible à l'adresse suivante : https://github.com/balez/generic .

Un programme OCaml perd toute notion de type une fois compilé, ainsi la désérialisation de données est une fonctionnalité dangereuse reposant sur la confiance des données lues : si les données lues sont corrompues, alors le programme peut s'interrompre brutalement ("segmentation fault"), ou pire...

La sérialisation des données peut s'avérer pourtant très utile :

- pour sauvegarder et charger des données dans des fichiers sur le disque dur.
- pour que deux programmes OCaml puissent communiquer à travers le réseau plus facilement sans utiliser des structures de données tel que JSON (la gestion de la lecture de données JSON est fastidieuse quand les structures de données utilisées sont complexes).

Le cas d'étude industriel, TrustInSoft Analyzer, utilise la sérialisation pour les deux cas cités. Le noyau de TrustInSoft Analyzer, tout comme sa version open-source Frama-C, utilisent déjà un mécanisme de sérialisation typé comme présenté dans le résumé à l'adresse : https://hal.inria.fr/hal-01091947/document .

Ce mécanisme est similaire à celui utiliser dans la bibliothèque Generic avec en plus le support pour la sérialisation et désérialisation de données utilisant du hashconsing. Generic n'ayant pas le support pour de telles données, il n'est alors pas envisageable de l'utiliser à la place du mécanisme existant.

TrustInSoft Analyzer dispose aussi d'une interface graphique (GUI) écrite en OCaml puis compilée vers JavaScript. Cette GUI s'utilise avec un navigateur Web et communique avec le noyau de TrustInSoft Analyzer à travers le réseau avec le protocole HTTP. Cette communication est faite en utilisant la sérialisation disponible dans le module `Marshal` d'OCaml. Les types des données utilisés

pour cette communication sont dans des fichiers partagés entre la GUI et le noyau lors de la compilation. Ainsi nous obtenons de forte garanties au niveau de la désérialisation tant que les deux entités sont compilées en même temps. Cependant, cette astuce ne protège pas des attaques malicieuses provenant du réseau.

Le cas d'étude de la bibliothèque est de remplacer les fonctions du module `Marshal` d'OCaml par celle de la bibliothèque Generic pour la communication entre la GUI et le noyau de TrustInSoft Analyzer. L'un des challenges de ce cas d'étude est la compilation et le support de la bibliothèque en JavaScript à l'aide de `js_of_ocaml`.

Retours d'expérience

Installation de Generic

Cette bibliothèque n'a pas besoin d'une version spécifique du compilateur OCaml et donc, pas non plus besoin des types à runtime dont il en était question dans la proposition du projet SecureOCaml. Ceci simplifie grandement l'utilisation de la bibliothèque et évite toute problématique de dégradation de performance à l'exécution lié à l'introduction des types à runtime lorsqu'ils ne sont pas utilisés.

Le premier bémol de la bibliothèque est l'absence de la fonctionnalité d'installation de la bibliothèque (par exemple avec `opam pin` ou `ocamlfind install`) et l'absence d'instructions pour la compilation d'un programme avec cette bibliothèque. Un exemple de compilation est donnée avec les tests de la bibliothèque, mais une explication dans le fichier README serait la bienvenue.

Le deuxième bémol est l'absence de support de la bibliothèque en natif. Frama-C et TrustInSoft Analyzer utilisent à la fois la compilation en bytecode et en natif.

Ces deux problèmes mineurs ont pu être résolu facilement pendant le cas d'étude. Deux versions de la bibliothèque sans ces problèmes sont disponible aux l'adresses suivantes : - https://github.com/bobot/generic/tree/jbuilder , en utilisant `opam` et `jbuilder` - https://github.com/david-maison/generic , en utilisant `ocamlfind`

Utilisation de `reify` ppx

La documentation de la bibliothèque est bien construire et claire. La migration du module `Marshal` vers celui de `Generic` est en théorie simple : - à chaque sérialisation et désérialisation, il faut donner un témoin (`Genetic.Ty.t`) du type de la donnée. - un témoin `Generic.Ty.t` peut être dérivée, grâce à ppx, d'un type quelconque en ajoutant l'annotation `[@@reify]` au type.

En pratique, l'utilisation de `reify` ppx a quelques limitations.

Prenons le programme suivant:

```
type t =
| A of int
| B of Lexing.position
| String of string
[@@reify]
```

- Nous obtenons l'erreur suivante à la compilation :

  ```
  Error: This variant expression is expected to have type
           'a Generic_core.Ty.T.ty
         The constructor Int does not belong to type Generic_core.Ty.T.ty
  ```

  Cette première erreur peut être résolue en rajoutant `open Generic.Ty`, mais l'utilisation de `open` pour `reify` ne devrait pas être indispensable, surtout que cela pourrait créer des problèmes liés au shadowing.

- Ensuite, nous obtenons une nouvelle erreur de compilation :

  ```
  Error: Unbound constructor Lexing.Position
  ```

  En effet, lorsqu'un type est dérivé avec `reify`, il faut que tout autre type utilisé dans ce dernier soient aussi dérivés avec `reify` (à l'exception des types primitifs d'OCaml).

  Cependant, dans certains cas, ces types peuvent provenir d'une autre bibliothèque. La modification de ces bibliothèques pour l'utilisation de `reify` n'est parfois pas envisageable.

  La documentation du module `reify.ml` décrit deux autres annotations: `[@@no_desc]` et `[@@abstract]`. La documentation ne spécifie pas de cas d'usage typique de leur utilisation, rendant difficile de savoir s'ils peuvent être utiles pour notre problème.

  Rajouter dans la documentation des cas concrets avec des exemples sortant de l'ordinaire comme celui-ci pour l'utilisation de `reify` serait appréciable pour l'utilisateur.

  Cette erreur peut être contournée avec la déclaration suivante :

  ```
  type position = Lexing.position
  [@@reify]
  [@@abstract]
  ```

- Il y a une autre limitation bloquante avec l'utilisation de `reify` dans notre cas d'étude bien spécifique lié au "packing" d'OCaml.

  Dans notre cas d'étude, les types utilisés pour la sérialisation entre le client (la GUI) et le serveur (plus précisément un plug-in de TrustInSoft Analyzer), sont dans des fichiers en commun avec les deux parties, mais ils restent compilés séparément. Le système de plug-in du noyau de TrustInSoft Analyzer (ou celui de Frama-C) utilise le packing (avec l'option `-for-pack` du compilateur).

Ainsi, nous avons pu observer une exception non rattrapée de la bibliothèque :

```
Uncaught exception: Not_found

The full backtrace is:
Raised at file "hashtbl.ml", line 194, characters 13-28
Called from file "generic_core_desc.ml", line 372, characters 24-72
Called from file "generic_core_desc.ml", line 381, characters 16-28
Called from file "generic_fun_deepfix.ml", line 30, characters 30-48
Called from file "generic_fun_marshal.ml", line 470, characters 15-75
```

Cette exception provient de la description des types dans les témoins. Pour certains types, les noms des modules, dans lequel est situé le type, se retrouve dans la description.

Par exemple, si le type `t` du module `Foo` est sérialisé du côté client, la description de ce type contiendra la chaîne de caractères `Foo.T`. Alors que du côté serveur, avec le packing, cette même chaîne de caractères pour ce même type sera `P.Foo.T` avec `P` le nom du module utilisé avec l'option `-for-pack`.

Ce problème est bien spécialisé à l'usage du packing et de la compilation séparé de notre cas d'étude. Cependant, le retour utilisateur de la bibliothèque pour signaler ce problème n'est pas très verbeux : pour comprendre que le problème était lié à la description interne des témoins, il a fallu introduire des messages de débogage dans la bibliothèque.

En plus de l'absence d'information lors d'une exception de la bibliothèque, il ne semble pas y avoir de moyen de pouvoir contourner ce problème avec `reify`. Pour continuer le cas d'étude, nous avons du modifier la chaîne de compilation de TrustInSoft Analyzer pour compiler nos types directement dans le noyau de l'analyseur sans passer par ce système de plug-in qui utilise le packing.

L'utilisation de `reify` ppx permet donc de pouvoir préparer un programme plus facilement et rapidement avec bibliothèque Generic. Cependant, quelques problèmes persistent avec l'utilisation de cette extension ppx rendant finalement plus difficile l'intégration avec Generic pour des cas d'usage plus complexes.

La documentation de la bibliothèque ne permet pas de savoir facilement comment l'utiliser sans l'extension ppx : un exemple pour sérialiser un type variant sans cette extension permettrait peut-être de contourner les problèmes liés à `reify` plus proprement.

Utilisation de Generic avec `js_of_ocaml`

La compilation de Generic avec `js_of_ocaml` pour l'utiliser en JavaScript depuis un programme originalement écrit en OCaml a été un succès. Son utilisation a même permis de découvrir un bug dans `js_of_ocaml` pour la primitive `caml_obj_dup` du runtime (https://github.com/ocsigen/js_of_ocaml/issues/666).

Il n'y a, à première vue, aucune limitation additionnelle liée à la bibliothèque dans un environnement d'exécution JavaScript comparé à la simple utilisation du module `Marshal`.

Pour rappel, en JavaScript, il n'y a que le type `number` pour représenter à la fois les types `int` et `float` en OCaml. Nous ne pouvons donc manipuler que des entiers allant jusqu'à `2^53 - 1`. Donc si le programme en JavaScript tente de désérialiser un entier plus grand que cette valeur, l'opération échouera. Mais ces problèmes ne sont pas liés à Generic.

Retour utilisateur sur les exceptions

Après l'intégration de Generic sur les types de notre cas d'étude, une erreur est survenue à plusieurs reprises lors de la sérialisation :

```
Uncaught exception: Generic_fun_marshal.Serialize_exception("Inhabited polymorphic type.")

The full backtrace is:
Raised at file "generic_fun_marshal.ml", line 247, characters 3-60
Called from file "generic_fun_marshal.ml", line 236, characters 8-26
Called from file "generic_fun_marshal.ml", line 277, characters 59-70
Called from file "list.ml", line 122, characters 24-31
Called from file "generic_fun_marshal.ml", line 296, characters 6-138
Called from file "generic_fun_marshal.ml", line 309, characters 4-49
Called from file "generic_fun_marshal.ml", line 236, characters 8-26
Called from file "generic_fun_marshal.ml", line 277, characters 59-70
Called from file "list.ml", line 122, characters 24-31
Called from file "generic_fun_marshal.ml", line 296, characters 6-138
Called from file "generic_fun_marshal.ml", line 309, characters 4-49
Called from file "generic_fun_marshal.ml", line 236, characters 8-26
Called from file "generic_fun_marshal.ml", line 159, characters 29-38
Called from file "generic_fun_marshal.ml" (inlined), line 452, characters 4-29
Called from file "generic_fun_marshal.ml", line 465, characters 26-37
Called from file "server/request_listener.ml", line 279, characters 15-76
```

Cette exception est lancée lorsqu'une valeur est sérialisée avec pour témoin `Generic.Ty.Any`.

Notre fonction de sérialisation, ici dans `request_listener.ml`, prend en entrée

une donnée de type `'a Generic.Ty.ty` et une autre de type `'a`. Le nombre de types pouvant être sérialisé par cette fonction est supérieur à 10 et les types ne sont pas triviaux : ce sont souvent des records/variants qui contiennent d'autres types, pouvant être aussi des records/variants, et ainsi de suite.

Avec l'utilisation de `reify` ppx, il est ainsi difficile de savoir quel est le type problématique dont le témoin est `Generic.Ty.Any`. Un meilleur retour utilisateur serait utile. Par exemple lorsqu'il s'agit d'un type imbriqué dans un autre, avoir l'information sur le type englobant pourrait sauvegarder plusieurs minutes de débug au développeur.

Une option de compilation pour plus de verbosité dans la bibliothèque lors de la sérialisation/désérialisation peut aussi être une solution à ce manque d'information en cas de problème (qui ne devrait arrivé qu'en phase d'intégration de la bibliothèque).

Autres remarques

- La bibliothèque ne supporte pas les variants polymorphes avec l'utilisation de `reify` ppx. Dans notre cas d'étude, ces types sont utilisés parfois pour exprimer plus facilement des contraintes sur un sous-ensemble de valeurs. La traduction vers des variants simples a été possible mais le support des variants polymorphes serait grandement apprécié.
- Au niveau des performances, le cas d'étude ne permet pas d'observer des mesures pertinentes : les messages sérialisés pour la communication entre le client et le serveur sont nombreux mais trop petits.

  Le surcoût en mémoire liés aux témoins semble parfaitement acceptable en théorie (avec peut-être une possibilité d'amélioration sur le nom du type dans le témoin qui est actuellement une chaîne de caractères).

Conclusion

La bibliothèque Generic propose des fonctionnalités intéressantes et utiles pour ajouter une couche de confiance sur des opérations où le typage d'OCaml ne peut apporter beaucoup de garanties comme lors de la désérialisation.

L'usage de `reify` ppx rend la portabilité d'un programme utilisant `Marshal` vers Generic simple et rapide en théorie. Cependant, les quelques limitations découvertes lors de ce cas d'étude, couplés avec un manque de verbosité en cas d'erreur, rendent l'utilisation de cette extension plus fastidieuse qu'elle ne devrait l'être.

Avec quelques évolutions sur la bibliothèque, l'efficacité de `reify` devrait être un atout majeur pour inciter toute utilisation du module `Marshal` à être remplacée par Generic.

Analyseur d'exceptions : ocp-analyzer

Introduction

ocp-analyzer est un analyseur d'exceptions pour OCaml développé par OCamlPro disponible en open-source à l'adresse suivante : https://github.com/OCamlPro/ocp-analyzer . Cet outil permet de détecter les exceptions qui ne sont pas gérées correctement.

L'utilisation des exceptions rend bien plus difficile la compréhension du flot de contrôle d'un programme, augmentant ainsi le risque de bugs dans le programme. Il n'y a, actuellement, aucun outil récent maintenu pour OCaml pouvant aider les programmeurs à vérifier si les exceptions de son programme sont bien gérées ou non. Or, il y a un réel besoin de pouvoir vérifier cela : plus le programme a un nombre de lignes de code important, plus il est difficile de gérer toutes les exceptions utilisées, même avec une revue de code manuelle.

Par exemple, la plate-forme d'analyse Frama-C (ou sa version industrielle, TrustInSoft Analyzer) a plus de 200k lignes de code et utilise souvent des exceptions. Des cas réels problématiques d'exceptions non rattrapées ont déjà été trouvés. Notamment, certains de ces problèmes provenaient d'une documentation obsolète pour des fonctions qui levaient des exceptions.

Pour éviter ces problèmes à l'avenir, deux solutions s'imposent : - Avoir un analyseur d'exception. - Éviter l'usage d'exception (ou/et supprimer les usages existants de ces exceptions), comme décrit dans les recommandations du sous-langage SecurOCaml.

Retours d'expérience

ocp-analyzer n'est actuellement disponible que pour la version 4.02.3 d'OCaml. Un tel outil dépend fortement de la version du langage, par conséquent, afin qu'il soit utile pour la communauté d'OCaml, il se doit donc d'être maintenu à chaque version. Dans le cas contraire, il deviendra un projet oublié comme fut `ocamlexc`, son prédécesseur.

L'utilisation de l'outil se veut plus compliqué. La documentation explique bien les contraintes d'utilisations au niveau des `.cmt`: tous les fichiers `.cmt` doivent être donnés dans l'ordre, y compris ceux des bibliothèques (stdlib inclue).

Pour certains programmes, cette contrainte est déjà un obstacle à l'utilisation de l'outil : certaines bibliothèques ne fournissent pas les fichiers `.cmt`.

Sur un exemple d'un outil interne de TrustInSoft simple de trois fichiers, nous pouvons constater les problèmes d'utilisation et d'ergonomie suivants: - Des cas erreurs non explicites :

- `$ ocp-analyzer args.cmt printer.cmt main.cmt`
  `Error while restoring env for file args.cmt`
  `Fatal error: exception Envaux.Error(_)`

  pour un cas supposé où des fichiers `.cmt` sont manquants (ceux de la bibliothèque standard).

- `$ ocp-analyzer $OCAMLDIR/camlinternalFormatBasics.cmt $OCAMLDIR/ocaml/pervasives.cmt \`
  `  args.cmt printer.cmt main.cmt`
  `Fatal error: exception Env.Error(_)`

  pour un cas supposé où des options `-I` sont manquantes pour aller lire des `.cmi`.

- `$ ocp-analyzer -I $OCAMLDIR $OCAMLDIR/camlinternalFormatBasics.cmt $OCAMLDIR/pervasives`
  `  args.cmt printer.cmt main.ml`
  `Fatal error: exception Typetexp.Error(_, _, _)`

  pour un cas où un fichier qui n'est pas un `.cmt` est donné par erreur à l'outil.

Ces cas d'erreurs ne donne aucune information utile à l'utilisateur pour l'aider à utiliser l'outil.

```
$ ocp-analyzer -I $OCAMLDIR $OCAMLDIR/camlinternalFormatBasics.cmt $OCAMLDIR/pervasives.cmt
  args.cmt printer.cmt main.ml
Fatal error: exception Typetexp.Error(_, _, _)
```

- La sortie standard de l'outil, lorsque l'analyse est réussi, contient des messages non exploitable ou non pertinent. Dans l'exemple suivant :

```
$ ocp-analyzer -I $OCAMLDIR $OCAMLDIR/camlinternalFormatBasics.cmt $OCAMLDIR/pervasives.cmt
  args.cmt printer.cmt main.cmt
Free variable in Pervasives : CamlinternalFormatBasics! -> CamlinternalFormatBasics/-994
Free variable in Args : Arg! -> Arg/-1207
Free variable in Args : Array! -> Array/-1418
Free variable in Args : Filename! -> Filename/-1557
Free variable in Args : List! -> List/-1419
Free variable in Args : Pervasives! -> Pervasives/-1202
Free variable in Args : Printf! -> Printf/-1140
Free variable in Printer : Args! -> Args/-2070
Free variable in Printer : Format! -> Format/-1673
Free variable in Printer : List! -> List/-1674
Free variable in Main : Args! -> Args/-2077
Free variable in Main : Format! -> Format/-2098
Free variable in Main : Pervasives! -> Pervasives/-2089
Free variable in Main : Printer! -> Printer/-2094
starting the analysis
```

```
analysis done
I found something:
Var .$$/-3:
as .$$/-3: <Top>;

Uncaught exception from external call
Location:
File "pervasives.ml", line 23, characters 2-98
Backtrace:
<Start>

Uncaught exception from external call
Location:
File "pervasives.ml", line 153, characters 2-42
Backtrace:
<Start>

[ ... ]

Uncaught exception from external call
Unknown location
Backtrace:
<Start>

Uncaught exception from external call
Unknown location
Backtrace:
<Start>
```

- On a une information sur des variables libres dans des modules, mais pas d'information simple sur comment la trouver.
- `I found something: Var .$$/-3: as .$$/-3: <Top>;` n'est pas compréhensible pour un utilisateur.
- Il y a plein d'avertissements lié au module Pervasives, et aucun moyen simple de filtrer ces messages avec l'outil. Ces problèmes sont certes réels, mais certains utilisateurs ne voudront pas nécessairement y prêter attention puisqu'il s'agit de la bibliothèque standard du langage. Un filtrage, même désactivé par défaut, serait le bienvenu.
- Certains avertissements n'ont aucune information de localisation, et sont donc inexploitables.
- L'outil génère aussi deux fichiers: un `.xlml` et un `.dot`. D'après la documentation, ils sont générés à des fins de déboggage et ne sont pas à destination des utilisateurs. Il n'y a donc aucun autre moyen de visualiser le résultat de l'analyse, ce qui peut s'avérer utile pour des programmes de

grande taille.

En modifiant le programme testé, nous avons introduit un cas où une exception peut être levée sans être rattrapée. En exécutant le programme modifié, l'exécution est bel et bien stoppée par cette exception non rattrapée, mais l'outil n'a pas su dire plus de chose que précédemment : est-ce que l'avertissement sur cette exception est comprise dans l'un des messages avec `Unknown exception from external call Unknown location` même s'il ne s'agit pas d'une fonction externe ?

Conclusion

Dans un cas d'utilisation d'un programme simple de 3 fichiers n'utilisant que la bibliothèque standard, l'utilisation de `ocp-analyzer` n'est pas immédiate et le résultat de l'analyse ne permet pas de savoir clairement s'il y a des exceptions qui ne sont pas gérées correctement.

Les contraintes sur à la présence des fichiers `.cmt` des bibliothèques utilisées à donner dans l'ordre de link sont très problématiques et limite très vite l'utilisation de `ocp-analyzer` dans des cas d'usages industriels. D'après les développeurs, l'outil n'est pas non plus en mesure de passer à l'échelle sur des programmes de la taille de Frama-C.

Le besoin d'un tel analyseur est réel et `ocp-analyzer` ne rempli pas ce besoin à ce jour. `ocp-analyzer` reste néanmoins une preuve de concept, plus avancé que les autres projets existants, mettant en avant la difficulté de conception d'un détecteur statique d'exception pour OCaml.

Analyseur de style - `ocp-lint`

Présentation de l'outil

`ocp-lint` est un analyseur de style pour des programmes OCaml. Lancé sur un projet OCaml, il en analyse les fichiers pour détecter différentes catégories d'alarmes allant de la syntaxe pure (mauvaise indentation) à des informations de typage (restriction du polymorphisme dans l'interface `.mli` d'un module par rapport à son implantation `.ml`). Ces catégories peuvent être activées ou désactivées soit en ligne de commande soit à l'aide d'un fichier de configuration. La sortie principale est textuelle, mais il existe un outil expérimental de visualisation HTML. L'outil peut également stocker ses résultats dans une base de données pour une consultation ultérieure.

Expérimentations

`ocp-lint` a été compilé avec la version 4.03.0 d'OCaml. L'utilisation d'opam pour installer les dépendances permet une installation très simple de l'outil.

Après une première utilisation sur un projet, un fichier de configuration `.ocplint` est créé, qui permet d'activer ou de désactiver sélectivement certaines catégories. Ce fichier est commenté, ce qui permet d'identifier facilement le but de la plupart des catégories d'alarme, même si certaines pourraient avoir une documentation plus détaillée. La sortie de `ocp-lint --list`, qui donne la liste de tous les warnings susceptibles d'être émis par l'outil, est également suffisamment claire dans la plupart des cas, mais une documentation plus exhaustive et dans un format un peu plus riche que du texte brut serait cependant appréciable. Deux warnings restent cependant extrêmement mystérieux: "Semantic patch", dont la signification peut se comprendre en reprenant la liste détaillée des warning obtenue avec `--pwarning`, et "Get Features" dont le sens reste obscur même avec ces messages détaillés. De même, il est également regrettable que les deux listes présentent des différences certaines, qui rendent leur comparaison plus compliquée. Une suggestion d'amélioration a été rapportée en ce sens.

L'utilisation de la base de données ne semble pas entraîner un gain de temps significatif lors des analyses suivantes du projet: `ocp-lint` reparse systématiquement les fichiers pour déterminer si des changements ont eu lieu. En tout état de cause, le temps d'analyse (90 secondes sur l'ensemble de la base de code de Frama-C à partir d'un ordinateur portable standard) reste acceptable.

Par défaut, l'outil fournit uniquement le nombre d'alarmes détectées, divisé en catégories et sous-catégories. Il faut relancer l'analyse avec `--pwarning` pour avoir la liste et la localisation des alarmes. Il est aussi possible de considérer certaines alarmes comme des erreurs, et d'en faire la liste avec `--perror`. Une troisième option `--pall` pourrait laisser croire par son nom et sa documentation qu'elle combine les deux, mais elle affiche en fait d'éventuelles informations supplémentaires. Une demande d'évolution du nom a été suggérée sur le dépôt github de l'outil. Par ailleurs, il faut se reporter au fichier `.ocplint` pour avoir la signification exacte de chaque warning individuel.

3 fichiers de la distribution courante de Frama-C provoquent une erreur d'`ocp-lint` lui-même. Le problème a été rapporté sur le dépôt github de l'outil.

Dans la configuration par défaut, 75000 warnings sont émis sur la base de code de Frama-C dont 30000 pour la seule catégorie `check_tuple`, qui demande d'utiliser un enregistrement plutôt qu'un n-uplet pour améliorer la lisibilité du code. Cette catégorie n'a pas été retenue dans la configuration finale utilisée, car elle n'est pas adaptée à toutes les situations et demande en outre une version de OCaml strictement supérieure à la plus vieille version supportée par Frama-C à ce jour.

D'autres catégories semblent également un peu surprenantes au premier abord. C'est le cas par exemple, des warnings suggérant d'utiliser des enregistrements

plutôt que des classes, ou de celui qui demande de mettre en majuscules les noms de types de modules, convention qui n'existe même plus dans la bibliothèque standard du langage. De même, le warning signalant des applications polymorphes de fonctions semble générer énormément de bruit en se déclenchant sur de nombreuses utilisations parfaitement légitimes.

Conclusion

Au final, une fois désactivées les catégories considérées comme non pertinentes pour Frama-C, 25000 warnings sont émis, dont une moitié concerne les espaces, tabulations et lignes trop longues, pour lesquelles un refactoring était déjà prévu. Par ailleurs, près de 10000 warnings concernent des identificateurs pas assez qualifiés, une catégorie où les faux positifs sont assez nombreux. Il reste néanmoins un bon nombre de warnings très pertinents, qui font d'`ocp-lint` un outil particulièrement intéressant pour améliorer la qualité du code de Frama-C. En outre, différentes propositions de nouvelles règles ont été faites sur le dépôt github afin de rendre `ocp-lint` encore plus utile.

Évolutions d'OCaml

Outre les outils autour d'OCaml, le compilateur lui-même a évolué durant le projet SecurOCaml, et un certain nombre de nouvelles features vont dans le sens d'un code plus robuste. C'est en particulier le cas de la distinction entre `string` (chaînes de caractères immutables) et `bytes` (tableau de caractères dont le contenu peut être modifié). Si pour des raisons de compatibilité les `bytes` ont d'abord été introduits comme des alias de `string`, Frama-C a rapidement migré vers le mode strict, dans lequel il n'est pas possible de modifier le contenu d'une chaîne. De même, la compilation de Frama-C active tous les warnings du compilateur, et les traite comme des erreurs, à l'exception d'un petit nombre d'entre eux pour des raisons documentées dans `share/Makefile.common` au dessus des variables définissant les listes des warnings à ignorer ou à ne pas considérer comme une erreur.

Outils extérieurs au projet

Outre les développements effectués dans le cadre du projet SecureOCaml, certains outils externes ont également été considérés dans l'optique d'améliorer la fiabilité du code. Ils sont brièvement décrits dans la suite de cette section.

Crowbar

crowbar (https://github.com/stedolan/crowbar) est une bibliothèque permettant de générer aléatoirement à l'aide d'afl (http://lcamtuf.coredump.cx/afl/) des entrées pour réaliser des tests unitaires. L'utilisateur dispose d'une interface assez puissante pour écrire des générateurs dédiés aux types de données et aux fonctionnalités qu'il entend tester, ainsi que pour indiquer les conditions de réussite et d'échec d'une exécution de la fonction sous test. Cette flexibilité est très importante pour Frama-C, dont les structure de données reposent sur des invariants complexes qu'une génération purement aléatoire aurait très peu de chance de respecter. Plusieurs essais concluants ont été réalisés, et l'utilisation de crowbar dans les tests d'intégration de Frama-C devrait se développer régulièrement à l'avenir.

OCamllint

ocamllint (https://github.com/cryptosense/ocamllint) est un vérificateur de style pour OCaml développé par Cryptosense. Contrairement à `ocp-lint`, il ne s'agit pas d'un outil indépendant, mais d'un `ppx` destiné à être chargé dynamiquement par le compilateur OCaml au moyen de l'option `-package ocamllint.ppx`. La documentation n'offre pas de liste des règles qui sont vérifiées par l'outil, même si les noms des constructeurs du type `Ocamllint_warning.t` offre une certaine intuition, et la configuration passe impérativement par l'écriture d'un greffon OCamllint à charger en plus, opération plus lourde que la simple d'édition du fichier `.ocplint`.

Une inspection des warnings émis et de la liste des constructeurs du type `Ocamllint_warning.t` semble montrer que la plupart des warnings utiles dans le cadre du développement de Frama-C sont déjà couverts par `ocp-lint`, à l'exception de quelques motifs très particuliers et qui, s'ils ne sont pas élégants, sont peu dangereux. `ocamllint` peut cependant être intéressant pour améliorer marginalement la base de code de Frama-C. Toutefois, `ocp-lint` semble plus pertinent, quitte à implanter les quelques règles d'`ocamllint` utiles et non encore supportées par l'outil