Devoir maison
MPRI 2.3.1 2015

Etienne Lozes

Le devoir est à rendre avant le

dimanche 8 novembre 2015.

Votre devoir est à envoyer par mail à lozes@lsv.fr. Le travail en binôme est encouragé. Veillez à ne remettre qu'un seul devoir par binôme, en précisant bien vos deux noms.

Objectif

Le but de ce devoir est de vous faire programmer votre propre minifoot, une version allégée de smallfoot, en partant d'un code à trou. Plus précisément, vous avez à compléter deux fonctions :

- la fonction `vc_gen` dans le fichier `vcgen.ml` : cette fonction transforme une liste de définition de fonctions en une liste de conditions de vérification.
- la fonction `get_frame` dans le fichier `entailment.ml` : cette fonction prend deux tas symboliques étendus et renvoie la frame correspondante si elle existe.

Vous avez à rendre les éléments suivants.

1. Les fichiers `vcgen.ml` et `entailment.ml` modifiés.
2. Un fichier `positives.sf` et un fichier `negatives.sf` au format d'entrée minifoot contenant les tests que vous avez passés : le premier fichier doit contenir des fonctions qui sont prouvables, le second des fonctions qui ne sont pas prouvables. Vous devez écrire au moins 5 fonctions par fichier (éventuellement des fonctions sans code).
3. Un fichier `discussion.txt` dans lequel vous faites un bilan de votre travail.
4. Un fichier `return.txt` dans lequel vous expliquez ce qu'il faudrait changer dans le code pour étendre le langage d'entrée de minifoot de façon à avoir une instruction `return` et des fonctions renvoyant des valeurs. Attention, il y a un point un peu subtil lié à l'utilisation de la règle de frame lors des appels de fonction et des boucles while.

Pour quelqu'un travaillant seul, se limiter à 9h de programmation est conseillé. Une bonne discussion des problèmes et de l'extension avec return sera notée avec un coefficient plus important pour quelqu'un travaillant seul. La qualité des tests proposés sera aussi apréciée.

Langage d'entrée de minifoot

Le langage d'entrée de minifoot est une restriction syntaxique du langage de smallfoot. *L'archive fournie contient un fichier* `examples.sf` *et un fichier* `non-examples.sf` *qui permettent de se familiariser avec la syntaxe.*

Un programme minifoot est un ensemble de fonctions s'appellant entre elles. Contrairement à smallfoot, il n'y a ni variables globales ni resources. Le passage de paramètres "par adresse" à la smallfoot n'est pas autorisé en minifoot, on a donc une sémantique d'appel de fonction "classique". Les cellules mémoires n'ont qu'un seul sélecteur successeur, nommé `tl`, et les instructions atomiques sont restreintes à ce seul sélecteur. En résumé, la syntaxe d'un programme peut être définie par la grammaire suivante (cf aussi le fichier `defs.ml` pour cette définition et les suivantes).

$$
\begin{array}{ll}
E ::= & x \mid \text{NULL} \\
b ::= & E{=}{=}E' \mid E\,!{=}E' \\
c ::= & x = \text{new}() \mid \text{dispose}(x) \mid x = E \mid E \to \text{tl} = E' \mid x = E \to \text{tl} \\
p ::= & c \mid p; p \mid \text{while } b \text{ do } p \mid \text{if } b \text{ then } p \text{ else } p \mid f(\vec{E}) \mid f_1(\vec{E}_1)||f_2(\vec{E}_2)
\end{array}
$$

Les annotations correspondent aux tas symboliques étendus vus en cours, mais *sans le prédicat liste*. On ne peut donc parler que de cellules à l'aide du prédicat $E \mapsto E'$. La syntaxe des tas symboliques A, B et des tas symboliques étendus φ, ψ est définie par la grammaire suivante.

$$
\begin{array}{ll}
A, B ::= & E{=}{=}E' \mid E\,!{=}E' \mid \bot \mid x \mapsto y \mid \mathsf{emp} \mid A * B \\
\varphi, \psi ::= & A \mid \mathsf{if}\ b\ \mathsf{then}\ \varphi\ \mathsf{else}\ \psi
\end{array}
$$

Contrairement à ce qu'on a vu en cours, l'assertion $E{=}{=}E'$ ou $E\,!{=}E'$ est ici précise : $s, h \models E{=}{=}E'$ si $[\![E]\!](s) = [\![E']\!](s)$ et $\mathsf{dom}(h) = \emptyset$. Ainsi, la formule $x{=}{=}y$ a dans minifoot la sémantique de la formule $x = y \wedge \mathsf{emp}$ dans la logique de séparation "standard" (celle de la plupart des articles publiés, et celle vue en cours).

Les variables apparaissant dans une annotation sont soit des variables existentielles, dont le nom commence par `_`, soit des variables déclarées dans le programme et interprétées par leur valeur à ce point du programme. Plus précisément, une variable non existentielle dans une précondition de fonction ne peut être qu'un paramètre de la fonction, mais les invariants de boucle et les postcondition peuvent aussi contenir des variables locales. De même, si un paramètre de la fonction est modifié dans le corps de la fonction, c'est la valeur de ce paramètre en sortie de fonction qu'il faut prendre pour interpréter la postcondition. Les variables existentielles peuvent se penser comme des paramètres

fictifs d'une fonction qui ne sont jamais modifiés au cours de la fonction. Ainsi, si une variable existentielle `_t` apparait à la fois dans la précondition et la post-condition d'une même fonction, elle dénote une valeur qui reste inchangée au cours de la fonction.

Pour qu'une annotation A soit précise, il faut que la formule $\exists \vec{x}.A$ soit précise (cf cours), où $\vec{x}$ est l'ensemble des variables existentielles. Dans ce cas, il existe au plus une valeur possible associée à ces variables existentielles. Le langage d'entrée de minifoot se restreint aux formules précises. C'est au cours de l'inférence de *frame* qu'il est le plus naturel de tester si une annotation est précise ; il vous appartiendra donc d'afficher un message d'erreur lorsque ce sera le cas en levant l'exception `Imprecise_formula` (cf. `error.ml`).

Conditions de vérification

La première partie de ce que vous devez programmer est la génération des conditions de vérification, autrement dit la transformation d'une fonction annotée en une famille de triplets $\{A\}\ SI\ \{B\}$, où SI est un pseudo-programme sans boucle ni appel de fonction, et composé d'instructions symboliques. Dans smallfoot et dans la littérature, les instructions symboliques s'apparentent beaucoup aux instructions réelles. Pour ce devoir, on a cherché un langage assembleur le plus simple possible, composé de trois instructions symboliques atomiques

— `inhale`(φ) : "désalloue" φ, autrement dit transforme $\varphi * \psi$ en ψ
— `exhale`(φ) : l'opération inverse, transforme ψ en $\varphi * \psi$
— `rename`(σ) : transforme A en $A\sigma$

Une instruction symbolique SI est soit l'instruction `skip`, soit une des trois instructions atomiques ci-dessus, soit une séquence $SI_1; SI_2$, soit enfin un choix non-déterministe $SI_1 + SI_2$

En regardant la définition exacte dans `defs.ml`, vous verrez aussi un champ `info` : c'est un texte qui sera affiché en cas d'erreur au cours de l'exécution symbolique, et qui vous aidera à identifier d'ou provient la condition de vérification. Si vous ne savez pas quoi mettre à cet endroit quand vous générez les conditions de vérification, mettez la chaîne vide.

Structure du code source

En dehors des fichiers `vcgen.ml` et `entailment.ml` dans lesquels vous avez à travailler, le code fourni comporte les fichiers suivants

— `Makefile` : classique, taper `make` pour compiler. Vous pouvez générer un toplevel OCaml avec `make toplevel` (ceci à des fins de débogage). *Avant même de commencer à programmer, vérifier que le code fourni compile.*
— `config.ml` : les options en ligne. Vous pouvez notamment utiliser l'option `-verbose` de minifoot pour afficher des informations sur l'exécution symbolique, (en particulier pour vérifier les conditions de vérification générées) et vous pouvez rajouter vos propres options pour afficher des

informations pertinentes pour le débogage.

- `error.ml` : gestion des erreurs, c'est la que le message d'erreur en cas de levée de l'exception `Imprecise_formula` est généré.
- `defs.ml` : les définitions importantes pour vous : programmes, annotations, etc
- `symbheap.ml` : beaucoup de fonctions utiles pour manipuler les tas symboliques, en particulier faire des substitutions, calculer l'ensemble des variables, etc.
- `print.ml` les fonctions d'affichage utilisées pour les messages d'erreur (utiles aussi en cas de débogage)
- `misc.ml` quelques fonctions utilisées à droite à gauche, mais surtout la gestion des identificateurs (=variables). On peut créer une variable avec la fonction `create_ident`, tester si elle est existentielle avec la fonction `is_existential`, et créer des variables fraiches. Il y a deux façons de créer des variables fraiches :
  - soit en leur donnant un nom frais mais proche du nom d'une autre variable avec la fonction `gensym` (`gensym x`$\rightsquigarrow$ `x_i`) ; c'est surtout cette fonction que vous serez amenés à utiliser ;
  - soit en utilisant la fonction `wildcard`, qui crée une variable existentielle fraiche à chaque appel.
- `symbexe.ml` : le coeur de l'exécution symbolique, où comment les conditions de vérification sont ramenées à des problèmes d'implication logique. A lire (c'est court) pour bien comprendre comment les deux parties à programmer interagissent, et préciser au besoin la sémantique des instructions symboliques.
- `ast.ml` la transformation du résultat du parser en des objets au format spécifié dans `defs.ml`. A priori assez peu intéressant, sauf si vous voulez comprendre ce qu'on perd par rapport à smallfoot.
- `location.ml` : petit module pour gérer les positions dans le code calculées au parsing, et affichées dans les messages d'erreurs quand on a gardé ces informations. A priori vous concerne peu.
- les autres fichiers sont issus de smallfoot (à quelques petits ajustements près). Ils ont peu d'intérêt vis à vis de ce devoir.

Bon travail. Pour toute question, vous pouvez me contacter par mail, je ferai suivre à tout le groupe les questions et réponses qui me semblent pertinentes.