TP6 Vérification d'un programme C avec Isabelle/HOL et Autocorres[1]

1 Plan

La modélisation et la vérification d'un micronoyau comme seL4[2] dans un assistant de preuve comme Isabelle[3] est un vaste sujet plein de détails techniques. Nous n'avons aucune chance d'y arriver dans deux heures de TP ! Mais en vérifiant un petit programme C, nous allons aborder quelques questions de base et aussi avoir un petit aperçu de la modélisation et la vérification avec Isabelle. Les cours de Jérôme Feret sur la sémantique axiomatique (logique de Hoare) et de Xavier Rival sur l'assistant de preuve Coq, que vous avez suivi, vont vous servir dans ce TP.

2 Préparation

L'assistant de preuve Isabelle et l'outil *AutoCorres*[4] [2] ont déjà été téléchargé sur les ordinateurs fixes (dans le dossier `/‘hostname‘`). Si vous voulez travailler sur votre ordinateur portable, vous pouvez télécharger Isabelle et Autocorres des sites web cités ci-dessus. Notez seulement que nous utilisons une ancienne version d'Isabelle (Isabelle2016 et non Isabelle2016-1) par souci de compatibilité avec AutoCorres 1.2. Les fichiers du TP sont disponibles sur la page du cours :

`http://www.di.ens.fr/~pouzet/cours/systeme/tp06/tp-06.tgz`

Après avoir décompressé les fichiers du TP, mettez-vous dans le dossier tp06, et lancez Isabelle avec la ligne de commande :

```
/‘hostname‘/Isabelle2016/bin/isabelle jedit -d /‘hostname‘/autocorres-1.2
                                              -l AutoCorres tp-06/intro.thy
```

La première fois qu'on lance Isabelle, il faut attendre quelques minutes pour qu'elle puisse construire ses théories de base. L'interface utilisateur est l'éditeur de texte *jEdit* qui interagit avec un processus Isabelle grâce au plug-in PIDE (« Prover IDE »). Il n'y a pas de « rideau bleu » comme dans ProofGeneral, l'état d'une théorie est mis à jour automatiquement et en continu. La réponse d'Isabelle à une commande est affichée dans l'onglet *Output* (cocher le cas *Proof state* pour voir les états des lignes de preuve) quand vous mettez le curseur à côté des lignes correspondantes. Beaucoup d'information sur Isabelle et jEdit est disponible dans l'onglet *Documentation*. Pour découvrir le sens d'un élément logique (y compris les symboles), vous pouvez cliquer sur lui en tenant la touche Ctrl. Si vous trouvez que le texte est trop grand, vous pourriez changer sa taille avec le menu *Isabelle/Preferences...* et puis l'option *jEdit/Text Area/Text Font* (il faut cliquer *Apply* avant de fermer la boîte de dialogue).

1. Sujet basé sur un exercice proposé dans le cours de vérification de Gerwin Klein à UNSW/Data61.
2. `http://ts.data61.csiro.au/projects/seL4/`
3. `http://isabelle.in.tum.de/website-Isabelle2016/index.html`
4. `http://ts.data61.csiro.au/projects/TS/autocorres/`

3 Vérification dans le modèle monadique

Nous allons vérifier la fonctionnalité du programme C donné dans le fichier `tp06a.c`. Ouvrez donc le squelette `tp06a.thy` dans jEdit.

La première ligne après les importations, `install_C_file "tp06a.c"`, lance un analyseur syntaxique qui transforme le texte du programme C en un terme d'un langage séquentiel générique appelé *SIMPL* [5]. Vous pouvez voir le terme créé avec la commande

```
thm tp06a_global_addresses.is_prime_body_def
```

Le langage SIMPL est fourni avec des modèles sémantiques opérationnels (« grand pas » et « petit pas ») qui définissent la transformation des états par des programmes, une logique de Hoare pour raisonner sur ces transformations, une tactique *vcg* (« Verification Condition Generator »), les théorèmes de correction, et bien d'autres choses encore. Cette transformation sert à encoder une sémantique d'un sous-ensemble du langage C dans un modèle doté de définitions et d'outils auxiliaires pour déclarer et vérifier des propriétés.

Ces mêmes outils sont utilisés pour créer le modèle C de seL4 :

```
install_C_file "kernel.c_pp"
```

C'est-à-dire, le code source après traitement par le préprocesseur. Ce modèle a deux principaux inconvénients 1/ le langage est effectivement C et donc très impératif et de bas niveau et 2/ il faut raisonner avec de mots 32 bits. Nous allons donc commencer avec le modèle monadique, mais la deuxième partie (optionnelle) du TD traite ce modèle pour ceux que cela intéresse.

Dans le projet l4.verified, les chercheurs ont écrit dans Haskell un modèle exécutable, mais plus abstrait, qu'ils ont importé dans Isabelle comme pour le modèle C. Cela produit un terme monadique avec des nombres naturels (parfois) et des structures de données plus abstraites que celles du modèle C; par exemple, des listes fonctionnelles plutôt que des listes chaînées. Ils ont défini une logique de Hoare pour ce modèle et un outil appelé *wp* (pour « weakest precondition ») qui sert la même fonction que le vcg [1]. Ils ont ensuite démontré une relation de raffinement (« ccorres ») entre ces deux modèles de seL4. Les outils et la relation permettent de raisonner sur le modèle monadique puis de transférer ces triples de Hoare vers le modèle SIMPL.

Cependant, ils ont trouvé la démonstration de la relation de raffinement assez mécanique et ils ont donc créé l'outil *AutoCorres* pour transformer automatiquement un modèle SIMPL vers un modèle monadique. C'est ce qui se fait dans la ligne

```
autocorres[ts_rules = nondet, unsigned_word_abs = is_prime is_prime] "tp06a.c"
```

qui donne un modèle/un terme que l'on peut voir avec `thm tp06a.is_prime'_def` et un théorème de raffinement (`thm tp06a.is_prime'_ac_corres`). Ce modèle manipule des nombres naturels qui va faciliter le premier exercice (nous serons obligés, cependant, de montrer que les variables n'excèdent jamais la limite `UINT_MAX`).

3.1 is_prime

Le fichier `tp06a.c` contient une fonction, `is_prime`, qui est censée renvoyer 1 si et seulement si l'entrée `n` est un nombre premier, voir la figure 1. Nous allons démontrer que cette fonction est correcte et qu'elle termine.

Commencez votre travail en regardant tout en bas du fichier au lemme `is_prime_correct`. Il est notre but ultime, les autres lemmes servent seulement à diviser le problème en plus petits morceaux.

```
/*
 * Determiner si le nombre donne 'n' est premier.
 *
 * Nous renvoyons 0 si 'n' est compose, ou pas-zero si 'n' est premier.
 */
unsigned int is_prime(unsigned int n)
{
    /* Les nombres plus petits que 2 ne sont pas premiers. */
    if (n < 2)
        return 0;

    /* Trouver le premier non insignifiant facteur de 'n'. */
    unsigned int i = 2;
    while (n % i != 0) {
        i++;
    }

    /* Si le premier facteur est 'n' lui-meme, 'n' est premier. */
    return (i == n);
}
```

FIGURE 1 – La fonction `is_prime`.

3.1.1 Invariance

Question 1. Définissez l'invariant `is_prime_inv` qui déclare ce qui est toujours vrai pendant l'exécution de la fonction `is_prime`. C'est-à-dire, avant chaque itération de la boucle et ainsi que quand la boucle termine. Isabelle est dotée d'une relation `m dvd n` ('m divides n'), mais l'exercice sera plus facile si vous utilisez `n mod m ~= 0` (comme c'est programmé dans la figure 1).

Conseil: Il est difficile au début de savoir ce qu'il faut mettre dans l'invariant. Tentez quelque chose: vous allez peaufiner votre définition en abordant les questions suivantes. Il faut une définition qui capture toute l'information que vous avez sur les deux variables de programme `i` et `n`, qui est aussi fort pour démontrer la postcondition de la question 4.

Question 2. Démontrez le lemme `is_prime_precond_implies_inv` qui énonce que l'invariant est vrai la première fois que le programme entre dans la boucle.

Conseil: Toutes les preuves sont possible avec une `unfolding` des définitions, un `apply clarsimp`, et un coup de `sledgehammer`. Plutôt que de tenter les manipulations logiques compliquées (qui n'ont pas été enseignés), faites confiance aux méthodes automatiques, lisez les obligations de preuve, réfléchissez(!) et adaptez votre invariant pour que les preuves passent.

Question 3. Démontrez le lemme `is_prime_body_obeys_inv` qui énonce que l'invariant est vrai entre itérations de la boucle.

Question 4. Démontrez le lemme `is_prime_inv_implies_postcondition` qui énonce que l'invariant implique la postcondition quand la boucle termine.

3.1.2 Terminaison

Question 5. Définissez une fonction `is_prime_measure` qui rend un nombre naturel qui décroît à chaque itération de la boucle. On s'en servira pour démontrer que la boucle termine.

Question 6. Démontrez que la fonction `is_prime_measure` décroît à chaque itération de la boucle.

3.1.3 Correction

Question 7. En utilisant les lemmes précédents et la tactique `wp`, démontrer que la fonction `is_prime` est correcte.

4 Option 1: Vérification dans le modèle C

En utilisant le squelette `tp06b.thy`, refaites la vérification de la section précédente. Le défi ici est de traiter directement les mots de 32-bits (`word32` en Isabelle). Mais, comme il est plus facile à travailler avec les nombres naturels, réfléchissez à la possibilité d'avoir deux niveaux de définitions et de lemmes : ceux de base en termes de nombres naturels (comme pour la section précédente), et les autres en termes du type `word32`.

Quelques conseils : la fonction `unat` transforme un `word` vers un `nat`; la fonction `of_nat` transforme en sens inverse; les preuves seront plus faciles si vous explicitez la taille des variables en ajoutant des annotations de type `::word32`. Vous utiliserez la tactique `vcg` au lieu du `wp`.

5 Option 2: Vérification de la multiplication par itération

En utilisant `tp06a.c` et `tp06a.thy` comme des squelettes, écrivez un petit programme en C qui calcule le produit de deux entiers naturel (donc entier non signé: `unsigned int`) en utilisant seulement l'opérateur d'addition et une boucle. Démontrez que ce programme est correct avec Isabelle et *AutoCorres*. Vous pourriez soit vous servir d'une précondition pour éviter le dépassement d'un entier soit rendre le résultat dans un plus grand entier (`unsigned long`).

6 Pour aller plus loin

La fonction `is_prime` est très naïve. Il y a des algorithmes beaucoup plus efficaces, mais leurs preuves de correction demandent un raisonnement et des mathématiques plus sophistiqués. Comment améliorer la fonction? Quelles obligations de preuve auriez-vous pour démontrer la correction de votre version?

Pour la vérification de programmes C, nous avons vu qu'une partie du problème, il reste beaucoup d'autres caractéristiques à traiter

- variables globales,
- appels à d'autres fonctions,
- pointeurs et le tas,
- opérations bit à bit,
- etcétéra.

Pour ne retenir que quelques références, il y a la thèse de Norrish [4] (qui a adapté son travail pour la vérification de seL4), l'article de Tuch et autres sur la modélisation réaliste de mémoire et de pointeurs [6], et l'article de Leroy sur le compilateur CompCert [3].

De même, les assistants de preuves sont des outils sophistiqués avec un fort mélange de théorie et de pratique. Pour aller plus loin avec Isabelle, il y a

- l'introduction : `http://isabelle.in.tum.de/dist/Isabelle/doc/tutorial.pdf`, et
- le livre récent *Concrete Semantics* : `http://www21.in.tum.de/~nipkow/Concrete-Semantics/`.

Sans parler d'autres outils comme Coq, PVS, HOL4 (et ses frères), Agda et ACL2.

Références

[1] D. Cock, G. Klein, and T. Sewell. Secure microkernels, state monads and scalable refinement. In O. A. Mohamed, C. A. Muñoz, and S. Tahar, editors, *Proc. 21st Int. Conf. on Theorem Proving in Higher Order Logics (TPHOLs 2008)*, volume 5170 of *Lecture Notes in Comp. Sci.*, pages 167–182, Montreal, Canada, Aug. 2008. Springer. `http://ssrg.nicta.com.au/publications/papers/Cock_KS_08.pdf`.

[2] D. Greenaway, J. Lim, J. Andronick, and G. Klein. Don't sweat the small stuff: Formal verification of c code without the pain. In *Proc. 2014 ACM SIGPLAN Conf. on on Programming Language Design and Implementation*, page 45, Edinburgh, UK, June 2014. ACM Press. `http://www.nicta.com.au/pub?id=7629`.

[3] X. Leroy. A formally verified compiler back-end. *J. Automated Reasoning*, 43(4):363–446, Dec. 2009. `http://gallium.inria.fr/~xleroy/publi/compcert-backend.pdf`.

[4] M. Norrish. C formalised in HOL. Technical Report 453, University of Cambridge, Cambridge, UK, Dec. 1998. `http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-453.pdf`.

[5] N. Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2006. `http://mediatum.ub.tum.de/doc/601799/document.pdf`.

[6] H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In *Proc. 34th ACM SIGPLAN-SIGACT Symp. Principles Of Programming Languages (POPL 2007)*, pages 97–108, Nice, France, Jan. 2007. ACM. `http://ssrg.nicta.com.au/publications/papers/Tuch_KN_07.pdf`.