Démonstration de Steel, une logique de séparation concurrente pour prouver des programmes F* (démonstration)

Aymeric Fromherz, Antonin Reitz

Démonstration de Steel, une logique de séparation concurrente pour prouver des programmes F⋆

Aymeric Fromherz[1] and Antonin Reitz[2]

[1] Inria Paris
aymeric.fromherz@inria.fr
[2] Inria Paris
antonin.reitz@inria.fr

Abstract

Steel est un framework pour développer et prouver des programmes concurrents écrits en F⋆, un langage de programmation avec types dépendants ainsi qu'un assistant de preuve. Inspiré par Iris, Steel repose sur une logique de séparation concurrente imprédicative qui inclut un modèle mémoire fondé sur des monoïdes commutatifs partiels et permet l'utilisation d'invariants alloués dynamiquement pour raisonner sur des interactions concurrentes sans recourir à des verrous ("locks"). Afin d'offrir une vérification semi-automatique, Steel sépare les obligations de preuves générées en deux parties : une procédure de décision partielle, implémentée à l'aide du moteur de tactiques de F⋆, raisonne sur la logique de séparation tandis qu'un solveur SMT permet de décharger automatiquement des obligations de preuves exprimées en logique du premier ordre, par exemple liées à de l'arithmétique. Dans cette démonstration, nous présentons plusieurs bibliothèques vérifiées qui illustrent l'expressivité et la programmabilité de Steel.

Introduction Steel [10, 1] est une logique de séparation concurrente pour raisonner à propos de programmes F⋆ [9], dévelopée en collaboration étroite avec Microsoft Research. Steel est inspiré par et partage plusieurs similitudes avec Iris [4, 2, 3]: notre logique de séparation concurrente est imprédicative, supporte l'allocation dynamique d'invariants pour modéliser des interactions concurrentes sans verrous ("locks"), et propose un modèle mémoire fondé sur des monoïdes commutatifs partiels, qui permettent d'encoder différents idiomes tels que des permissions fractionnelles ou des références qui évoluent monotoniquement selon un préordre défini par l'utilisatrice. Comparé à Iris, Steel repose sur un plongement de surface ("shallow embedding") d'une logique de séparation en F⋆. Cela permet à Steel d'être applicable directement à F⋆ dans sa globalité, incluant par exemple ses types dépendents et à refinements.

Un Exemple: Swap Pour présenter les spécificités de Steel, nous utilisons un exemple classique, la fonction swap en Figure 1, qui échange le contenu de deux références r1 et r2.

Pour spécifier une fonction Steel, nous utilisons un effet [8], Steel, indexé par plusieurs éléments. Le premier indice, ici unit, correspond au type de retour de la fonction swap. Les deux indices suivants, ici ptr r1 ∗ ptr r2 et λ_ → ptr r1 ∗ ptr r2, correspondent à une précondition et une postcondition, exprimés à l'aide de notre logique de séparation. Pour swap, cette spécification repose sur le prédicat ptr r, qui symbolise que la référence r est actuellement une référence valide, i.e., elle a été correctement allouée en mémoire, et n'a pas encore été libérée. Cette spécification requiert donc initialement que r1 et r2 soient deux références valides, mais également disjointes, tel que signifié par l'utilisation de la conjonction séparatrice ∗ de logique de séparation. La postcondition est similaire, avec une différence: elle peut dépendre de la valeur de retour, et est donc représentée comme une fonction. Pour swap, cela n'est pas le cas, et la valeur de retour est donc ignorée (utilisant la syntaxe λ_ →), mais cette dépendence permet par exemple de spécifier par λr → ptr r une fonction d'allocation qui retournerait une nouvelle référence r.

```
let swap (r1 r2:ref int) : Steel unit
    (ptr r1 ∗ ptr r2) (λ _ → ptr r1 ∗ ptr r2)
    (requires λ_ → ⊤)
    (ensures λs _ s' → s'.[r1] = s.[r2] ∧ s'.[r2] = s.[r1])
  = let x1 = read r1 in
    let x2 = read r2 in
    write r2 x1;
    write r1 x2
```

Figure 1: Un exemple simple de programme Steel: swap. Après exécution, les valeurs stockées en mémoire dans les deux références sont échangées.

Une des particularités de Steel est la présence des deux derniers indices, ici requires $\lambda\_ \rightarrow \top$ et ensures λs _ s' $\rightarrow$ s'.[r1] = s.[r2] $\wedge$ s'.[r2] = s.[r1] Ces deux derniers indices sont des prédicats opérant sur des *sélecteurs*, qui ne dépendent que des fragments de la mémoire correspondant à la spécification utilisant la logique de séparation. Cette restriction est nécessaire pour assurer que ces prédicats interagissent correctement avec notre logique de séparation, et en particulier avec la *frame rule* [7] qui permet un raisonnement modulaire. Le sélecteur d'une référence s.[r] correspond à la valeur qu'elle contient. Pour swap, nous pouvons donc spécifier la correction fonctionnelle de la fonction en utilisant ces prédicats: la valeur s.[r1] de r1 dans l'état initial s correspond à la valeur s'.[r2] de r2 dans l'état final s', et vice-versa.

Automatisation de Steel L'utilisation de logique de séparation et de prédicats opérants sur des sélecteurs permet une séparation des obligations de preuve en deux catégories. En Steel, par construction, toutes les obligations de preuves liées à la logique de séparation sont exprimées comme des équivalences modulo réécritures associatives et commutatives de l'opérateur $*$. Pour décharger ces obligations, nous avons défini une procédure de décision partielle qui utilise l'unificateur d'ordre supérieur de F* pour effectuer une unification modulo réécritures. Cette procédure de décision est implémentée comme une tactique F*, ce qui garantit sa validité par rapport à F* même. Cette procédure est partielle, et requiert parfois une application manuelle de lemmes, par exemple pour dérouler un prédicat récursif, ou pour introduire ou éliminer un quantificateur. Les obligations de preuves liées aux sélecteurs sont directement déchargées par un solveur SMT, utilisant le support natif de F* pour la vérification par SMT. Pour swap, la répartition entre prédicats de logique de séparation et prédicats de sélecteurs— qui spécifient respectivement la sûreté mémoire et la correction fonctionnelle— est naturelle, et permet une vérification entièrement automatique de ce programme.

Conclusion Au-delà de cet exemple simple, Steel contient aussi deux effets, SteelAtomic, et SteelGhost. Le premier permet de représenter l'atomicité de programmes, nécessaire pour raisonner sur des invariants partagés entre plusieurs exécutions concurrentes. Le second permet d'opérer sur un état fantôme, souvent utile lors de preuves complexes. À l'aide de ces fonctionnalités, Steel a été utilisé pour vérifier une large variété de programmes [1], tels que des arbres binaires de recherche automatiquement équilibrés, un compteur monotone concurrent initialement proposé par Owicki-Gries [6], une file concurrente à deux verrous proposée par Michael et Scott [5], ou une plateforme pour encoder des sessions à types dépendents entre deux parties. Dans cette démonstration, nous proposons de présenter plusieurs bibliothèques vérifiées en Steel, qui illustrent l'expressivité et la programmabilité de l'outil.

References

[1] Aymeric Fromherz, Aseem Rastogi, Nikhil Swamy, Sydney Gibson, Guido Martínez, Denis Merigoux, and Tahina Ramananandro. Steel: Proof-oriented programming in a dependently typed concurrent separation logic. 2021.

[2] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. Higher-order ghost state. 2016.

[3] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28, 2018.

[4] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. 2015.

[5] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. 1996.

[6] Susan Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–285, 1976.

[7] Matthew J. Parkinson and Alexander J. Summers. The relationship between separation logic and implicit dynamic frames. *Logical Methods in Computer Science*, 2012.

[8] Aseem Rastogi, Guido Martínez, Aymeric Fromherz, Tahina Ramananandro, and Nikhil Swamy. Programming and proving with indexed effects, 2021. In Submission.

[9] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. 2016.

[10] Nikhil Swamy, Aseem Rastogi, Aymeric Fromherz, Denis Merigoux, Danel Ahman, and Guido Martínez. SteelCore: An extensible concurrent separation logic for effectful dependently typed programs. 2020.