Conception et implémentation d'un outil et d'une librairie OCaml pour manipuler les lieurs

Proposition de stage de recherche (niveau M2)

18 décembre 2014

Encadrant

François Pottier, directeur de recherche, INRIA Paris-Rocquencourt (francois.pottier@inria.fr)

Résumé

Lorsqu'on écrit un programme qui manipule des programmes, des formules, ou autres objets symboliques, on fait fréquemment face à la question de la gestion des noms et des lieurs. Par quelle structure de données faut-il les représenter en machine ? Sous quelle forme faut-il écrire le code qui effectue les opérations élémentaires (renommage, substitution, etc.), de façon à rendre la tâche du programmeur moins répétitive et moins sujette aux erreurs ? La littérature regorge de réponses variées à ces questions. Il existe aujourd'hui quelques librairies pour différents langages de programmation, en particulier Haskell et Coq. Pour OCaml, cependant, il n'existe pas grand-chose, en dehors de Cαml [4, 5], un outil écrit par moi-même en 2005. Cet outil est aujourd'hui dépassé à de multiples points de vue. L'objectif de ce stage est de concevoir un nouvel outil, accompagné d'une nouvelle librairie, qui soient aussi simples, ouverts, extensibles que possible.

Il s'agit d'un sujet de recherche, car certaines questions restent ouvertes : par exemple, quel langage faut-il offrir à l'utilisateur pour décrire la structure des termes avec lieurs ? Quelle(s) représentation(s) en machine doit-on ou peut-on adopter ? Cependant, ce sujet comporte également une forte composante d'implémentation, car il faut développer en OCaml une librairie, un outil de génération de code, ainsi que des applications permettant de les tester.

Quelques principes

Soulignons d'emblée que le but n'est pas de faire « quelque chose qui marche, vite ». Cαml rentrait déjà dans cette catégorie par certains aspects, en particulier son système de génération de code, un peu trop primitif et peu flexible.

Un objectif essentiel, ici, est au contraire de mettre au point une architecture propre, simple, modulaire, facile à faire évoluer.

On essaiera donc de découper le projet en plusieurs composants ou couches indépendantes. On évitera, autant que possible, de « visser en dur » certaines décisions fondamentales, comme le choix de la représentation des noms et des lieurs. On essaiera d'écrire le maximum de code sous forme d'une librairie et le minimum de code sous forme d'un générateur, car ce dernier est plus difficile à comprendre et à maintenir. On pensera dès le début aux applications, afin de ne pas s'enfermer dans une impasse. On essaiera d'obtenir toujours une complexité asymptotique satisfaisante, mais on ne s'inquiètera pas trop de la performance absolue.

Le découpage du projet en différents composants n'est pas encore clair, mais on peut suggérer quelques éléments :

1. Un outil pour engendrer un visiteur (c'est-à-dire une classe offrant des opérations `map` et `fold` extensibles) à partir d'une définition de type algébrique OCaml. Ce type d'outil existe déjà (voir `camlp4`, `deriving`, `ocsigen-deriving`, `type-conv`, `ppx_deriving`, etc.). Cependant, on souhaitera peut-être le réimplémenter, afin de mieux le contrôler et de pouvoir l'étendre, si besoin, à des définitions de types avec lieurs.
2. Une librairie de bas niveau, fournissant la représentation des noms et des abstractions en machine, ainsi que les opérations élémentaires dont nous avons besoin sur les noms et les abstractions. Dans un premier temps, on pourra fixer une représentation particulière (par exemple « nominal », « locally nameless », ou « de Bruijn »). Cependant, dans un second temps, il serait souhaitable de comprendre comment on peut offrir un choix entre plusieurs représentations, offrant une interface unifiée.
3. Une librairie de haut niveau qui, en s'appuyant sur les deux composants précédents, implémente l'éventail des opérations usuelles de haut niveau : test d'égalité modulo α-équivalence ; substitution ; calcul des noms libres ; rafraîchissement progressif ; etc.
4. Une ou plusieurs applications, à faire évoluer en même temps que la librairie, afin de tester sa conception et son implémentation. On peut songer par exemple à un *type-checker* pour un langage typé simple, par exemple Système F, ou encore le noyau de Mezzo.

Pré-requis

Des bases solides en programmation en général, et en OCaml en particulier, sont indispensables.

Détails pratiques

Le stage se déroulera à l'INRIA, sur le site de Rocquencourt, sous la direction de François Pottier, de mars à juillet 2015 environ.

Quelques références bibliographiques

On pourra s'inspirer de plusieurs outils ou librairies existants, d'une part pour la conception du langage de description de termes avec lieurs, d'autre part pour l'organisation de l'outil et des librairies. La liste suivante n'est pas exhaustive :

1. Cαml [4, 5] produit du code OCaml.
2. Ott [7] est doté d'un langage de spécification puissant, mais ne produit pas de code.
3. LNgen [1] produit du code Coq.
4. Unbound [8] est également doté d'un langage de spécification intéressant. Il produit du code Haskell.
5. NomPa [6] est une librairie Agda. Elle peut nous donner des idées pour ce qui concerne la ré-utilisation et le partage de code.
6. Bound [3] est une librairie Haskell, peu efficace je crois, mais élégante.
7. Clochard, Marché et Paskevich [2] ont écrit une librairie et prouvé sa correction à l'aide de Why3.

Références

[1] Brian Aydemir and Stephanie Weirich. LNgen: Tool support for locally nameless representations. Technical Report MS-CIS-10-24, University of Pennsylvania Department of Computer and Information Science, 2010.

[2] Martin Clochard, Claude Marché, and Andrei Paskevich. Verified programs with binders. In *Programming Languages Meets Program Verification (PLPV)*, pages 29–40, 2014.

[3] Edward Kmett. Bound. Blog post, 2014.

[4] François Pottier. Cαml, 2005.

[5] François Pottier. An overview of Cαml. In *ACM Workshop on ML*, volume 148 of *Electronic Notes in Theoretical Computer Science*, pages 27–52, 2006.

[6] Nicolas Pouillard and François Pottier. A unified treatment of syntax with binders. *Journal of Functional Programming*, 22(4–5) :614–704, 2012.

[7] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strnisa. Ott: Effective tool support for the working semanticist. *Journal of Functional Programming*, 20(1) :71–122, 2010.

[8] Stephanie Weirich, Brent A. Yorgey, and Tim Sheard. Binders unbound. In *International Conference on Functional Programming (ICFP)*, pages 333–345, 2011.