Heuristique d'inlining complexe

Niveau requis: M1 ou M2 lié aux langages de programmation et compilation

Durée du stage: 5 ou 6 mois prévus, adaptable selon conditions

Contexte

Dans un compilateur optimisant, l'inlining est une optimisation centrale. Non pas parce qu'elle améliore significativement les performance d'elle même, mais plutôt parce qu'elle permet aux autres optimisations de s'activer. Et ceci d'autant plus que le langage laisse l'utilisateur libre de construire des abstractions. Ceci est particulièrement le cas en OCaml qui est un langage fonctionnel permettant d'écrire des fonctions polymorphes.

Par exemple il est possible d'écrire une fonction transformant les éléments d'une liste de manière très générique.

```
let rec list_map (f : 'a -> 'b) (l : 'a list) : 'b list =
  match l with
  | [] -> []
  | h :: t -> f h :: list_map f t

val list_map : ('a -> 'b) -> 'a list -> 'b list
```

Le type de cette fonction se lit "Pour tout type a et b, pour toute fonction de type (a -> b), cette fonction consomme des listes de a et retourne des listes de b". C'est très pratique pour l'utilisateur, mais ne laisse pas beaucoup de latitude au compilateur pour l'optimiser.

Une fois en contexte, cette fonction pourrait être spécialisée pour le site d'appel et en particulier, pour la fonction `f` passée en argument.

Néamoins, le choix de la spécialisation ne doit pas être fait systématiquement. En effet spécialiser ou inliner a un coût: en temps de compilation et en taille des exécutables produits. Et un gros exécutable consomme de la bande passante mémoire à l'exécution, ce qui risque de ralentir significativement le programme.

Mais bien faire le choix d'inliner est complexe et potentiellement coûteux. Le faire parfaitement est en général indécidable. C'est pour cela que les compilateurs suivent en général des heuristiques pour évaluer ces choix.

Dans le cas de OCaml, cette heuristique est relativement simple et guidée par une analyse du bénéfice apporté par la transformation: inlininer une fonction a un coût en taille et un bénéfice en temps d'exécution, tout deux estimés. Si le bénéfice dépasse le coût alors l'inlining est appliqué. Néamoins l'exploration des cas potentiellement interessant est coûteux à la compilation. En particulier il arrive souvent qu'inliner une fonction seule ne change rien, mais faire le choix d'appliquer la transformation simultanément sur plusieurs fonction soit très bénéfique. Par exemple pour une fonction jouet `map_pair` :

```
let map_pair g h (a, b) = (f a, g b)
```

Inliner `map_pair` seul ne sert à rien en tant que tel, mais être capable d'inliner `f` et `g` est potentiellement très bénéfique. Ces choix ne peuvent être fait que simultanément, il faut donc une procédure de recherche en profondeur de cas potentiellement bénéfiques.

Pour borner le coût potentiel d'une telle procédure, il y a des restrictions sur la taille des fonctions potentiellement inlinées et le nombre d'essais avant qu'un choix d'inlining soit considéré comme bénéfique.

Celà convient assez bien à de nombreux usages, mais dans certains cas les niveaux d'abstraction sont tels que pour pouvoir reconnaître il faudrait essayer des combinaisons d'inlining très profond.

Sujet de stage

Ce type de recherche profond n'est évidemment pas faisable systématiquement, mais il est envisageable de reconnaître des situations où cette exploration a de grandes chances d'être bénéfique sans être trop coûteux. Ce stage a pour but d'évaluer et ou formuler de telles heuristiques.

Sont envisagées entre autre * Définir des langages d'annotations adaptées pour laisser l'utilisateur guider cette recherche. * Faire une analyse permettant de reconnaître quels arguments éliminent des branches de fonctions en forçant des conditionnelles. * Reconnaître des formes classique de généricité similaire aux type classes, qui bénéficient fortement de spécialisation.

Pour ce dernier exemple, la transformation à appliquer n'est pas exatement l'inlining. Les détails de cette transformation serait à explorer.

Implémentation

Un tel stage contient une large base théorique à développer, mais serait bancale sans expérimentation. Ainsi un travail d'implémentation serait attendu du stagiaire. Mais faire ce genre d'implémentations sur le code du compilateur OCaml ferait perdre beaucoup de temps au stagiaire. Pendant la durée d'un stage, il n'est pas envisageable de pouvoir le maitriser suffisement pour pouvoir expérimenter efficacement. C'est pour cela que le stagiaire expérimentera sur des langages jouets. Le résultat final attendu du stage, pour OCamlPro sera principalement la théorie et l'expérience acquise, et non le code lui même.

Environnement

Le stage se déroulera au sein de l'équipe 'flambda' chez OCamlPro qui développe des optimisations sur le langage OCaml, et en particulier l'inliner.

Connaissances requises

- Connaissances en compilation

Lieu du stage

Le stage se déroulera dans les locaux d'OCamlPro, 21, rue de Chatillon, 75014, Paris.

Contact

Vincent Laviron, Senior R&D Engineer, vincent.laviron@ocamlpro.com
Pierre Chambart, Senior R&D Engineer, pierre.chambart@ocamlpro.com

À Propos d'OCamlPro :

OCamlPro SAS est une société issue de l'INRIA, créée en avril 2011, pour promouvoir l'utilisation du langage de programmation OCaml dans le milieu industriel. Elle participe activement à des programmes de recherche et de développement visant à améliorer la sûreté et la sécurité des applications informatiques en général. Vous trouverez plus d'informations sur notre site web : http://www.ocamlpro.com/