jumel

http://graal.ens-lyon.fr/jumel

Délivrable de fin de parcours

Paul-Elliot Anglès d'Auriac, Simon Castellan,
Rémi De Joannis De Verclos, Lucas Hosseini,
Jean-Florent Raymond, Guillaume Sergent

12 janvier 2012

Résumé

jumel est un un compilateur pour un dialecte de ML auquel sont adjointes les principales primitives du π-calcul, donc du passage de messages. Le but est de compiler ce langage dans un langage propre de Lπ (un π-calcul restreint) et ensuite d'exécuter ce langage par une machine virtuelle capable de distribuer le code.

1 Prévisions du dernier délivrable

D'après le rendu mi-parcours, voilà les objectifs que nous devions remplir :

1. analyses statiques :
   - système de types évolué $\rightarrow$ non fait ;
   - distribution du code $\rightarrow$ un début d'implémentation a été fait et marche assez bien sur des exemples raisonnables ;
   - optimisations $\rightarrow$ de petites optimisations permettant de considérablement diminuer la redondance des programmes générés par la compilation ont été faites.
2. compilateur :
   - fin des traductions $\pi + ML$ vers représentation intermédiaire $\rightarrow$ fait ;
   - traduction représentation intermédiaire vers $L\pi$ $\rightarrow$ la représentation intermédiaire a été supprimée.
3. machine virtuelle :
   - gestion plus efficace des écoutes $\rightarrow$ fait ;
   - *thread-pool* : gestion des *threads* de manière globale $\rightarrow$ fait ;
   - *garbage collector* $\rightarrow$ un GC mono-machine a été codé mais le problème a été largement sous-estimé.

4. infrastructures :
   – poursuite de la documentation sur le wiki et dans le code → fait ;
   – fichiers de configuration → fait.

1.1 Analyses statiques

Suppression de la représentation intermédiaire La représentation intermédiaire étant vraiment redondante par rapport au Lπ. Il a donc été choisi de la supprimer et d'étendre le Lπ, en autorisant les identificateurs abstraits. Par exemple la conversion en De Bruijn est alors simplement de traduire les identifiants d'un arbre Lπ sous forme de chaînes, vers des identifiants entiers.

Distribution de code La distribution de code est actuellement implémentée au niveau du Lπ. Elle se base sur les annotations de l'utilisateur qui permet à l'utilisateur d'indiquer quel *thread* doit tourner sur quel nœud. Le but de la distribution de code est de compléter ces annotations, en rajoutant du code s'il le faut.

Un exemple où cela est nécessaire est le cas d'une écoute sur un canal a de laquelle dépendent deux *threads* qui s'exécutent sur deux machines différentes. Alors il faudra mettre en place un canal qui se chargera d'envoyer la valeur reçue à la machine sur laquelle n'est pas stocké a.

Optimisations Nous avons implémenté quelques optimisations de base qui permettent d'avoir du code plus clair. La vraie optimisation intéressante est la propagation des communications immédiates : quand on a un thread de la forme $a(x).P||\overline{a}\langle v\rangle$ on peut remplacer tout de suite par $a[v/x]$. Cela permet de compacter le code ; en effet la traduction génère pour $\overline{c}\langle 1+1\rangle$ du code qui ressemble à $a(x).b(y).\overline{c}\langle x+y\rangle||\overline{a}\langle 1\rangle||\overline{b}\langle 1\rangle$. Ces optimisations permettent donc que les expressions arithmétiques apparaissent de la même manière dans la source et dans et le code Lπ

1.2 Compilateur

Traductions Nous avons terminé les traductions de jumel vers Lπ dans le compilateur, notamment la compilation des sommes. De plus, beaucoup de tests ont été fait pour vérifier la correction de ses traductions.

Bibliothèques On a ajouté la possibilité d'inclure des fichiers source dans jumel en utilisant le préprocesseur C. Plus exactement, on a rendu le compilateur compatible avec les annotations de `cpp` ce qui permet d'avoir un positionnement correct des erreurs. Dans le dépôt, on a implémenté une bibliothèque pour gérer les listes et les chaînes (qui sont des listes).

1.3 Machine virtuelle

Thread-pool La gestion des threads a été grandement améliorée. On distingue désormais deux types de threads :

- les "services", *i.e.* les threads exécutant une réception répliquée, qui ne sont pas destinés à terminer ;
- les "tâches", *i.e.* les threads exécutant des instructions, et qui sont quant à eux destinés à terminer.

Pour gérer tout cela, le *thread manager* implémente un système de *spawning* pour lancer des services, et un système de *thread pool* pour gérer les tâches. Ainsi, dès que l'éxecution concurrente d'un bloc d'instructions est nécessaire, il suffit d'ajouter la tâche adéquate à la liste des tâches en attente, et le *thread-pool* s'occupera de l'exécuter dès que possible.

Cette organisation permet à la fois de contrôler (en partie) le nombre de threads actifs en même temps, et de détecter la terminaison.

Détection de la terminaison Concernant la détection de la terminaison, l'avancée est de taille. Nous sommes parvenus, à l'aide de *mutices* sur le *thread manager* ainsi que sur le *channel manager*, à garantir la propriété "exécution en cours = au moins une tâche en cours ou en attente, ou un paquet en attente dans le channel manager". Ainsi, pour détecter la terminaison locale, il suffit de tester cette propriété. Cependant, certaines constructions, comme par exemple les références, ne respectent intrinsèquement pas ce principe de stabilité, et il a fallu modifier la manière dont nous gardons trace des paquets en attente pour garantir la terminaison avec les références. A l'heure actuelle, le schéma de traduction de la construction somme ne permet pas de détecter la terminaison.

Garbage collector De plus, un *garbage collector* a été implémenté, bien que son intégration au projet n'a pu se faire, en raison d'une large sous-estimation des dépendances lors d'un fonctionnement en réseau. L'approche adoptée est connue sous le nom de *mark and sweep*, et consiste à maintenir un ensemble de racines, qui sont des pointeurs sur les données directement visibles à un instant donné, et à effectuer des cycles au cours desquels le graphe d'accessibilité est parcouru, les sommets accessibles marqués, puis les sommets non marqués à l'issue de cette phase libérés.

Interface Nous avons ajouté à la machine virtuelle une interface ncurses optionnelle. Cela permet d'observer les échanges de messages, d'afficher les valeurs reçues et les statistiques concernant les données échangés.

Canaux virtuels Pour permettre aux gens de rajouter des primitives aux langages, nous avons ajouté à la machine virtuelle un système de canaux virtuels : quand le code fait une lecture ou une écriture dessus, une fonction C est appelée spécifique au canal. Par exemple, on a implémenté un canal

`write_string` qui lorsqu'on écrit dessus une chaîne (c'est-à-dire une liste d'entiers), qui affiche cette chaîne sur le terminal de la machine virtuelle.

Bogues Lors du développement de la machine virtuelle, nous avons rencontré de nombreuses difficultés. En effet, la nature non-déterministe rend le déboggage très compliqué, et la concurrence elle même introduit son lot de dangers (deadlocks, livelocks, ...). Ainsi, un temps conséquent à été consacré à la correction de bogues.

2 Capacités actuelles

À cette date, jumel est capable de réaliser les actions suivantes :

1. compiler un programme jumel vers du code Lπ avec :
   - un typage simple ;
   - des optimisations ;
   - un découpage du code produit en vue de la distribution.
2. exécuter un programme Lπ avec :
   - du parallèlisme ;
   - des échanges de messages entre des machines distantes ;
   - un *garbage collector* ;
   - un *log* des communication effectuées ;
   - des statistiques concernant les communications ;
   - une interface ncurses.

3 Tester jumel : détails pratiques

Le plus simple est de suivre la page *quickstart* du wiki `http://graal.ens-lyon.fr/jumel/doku.php?id=quickstart`, qui contient toutes les instructions pour compiler, tester et apprécier jumel avec suavité.

4 Prochains objectifs

Bien que le projet intégré soit terminé, le développement de jumel n'est bien entendu pas complet. La version actuelle est à considérer comme un prototype, dont la création nous a permis de mieux saisir les difficultés inhérentes à chaque partie, ainsi que certaines maladresses de conception difficiles à prévoir. Les objectifs en ligne de mire sont les suivants :

- typage avancé ;
- statistiques plus détaillées ;
- meilleure distribution de code ;
- *garbage collector* distribué ;
- optimisations avancées ;

– internationalisation complète du wiki ;
– écriture de gros programmes en jumel (tests) ;
– écriture de documentation.