Programmation paresseuse et causale des architectures orientées services

Remi.Douence@mines-nantes.fr
Ascola (Mines Nantes)

February 25, 2015

Abstract

L'évaluation paresseuse est une technique essentielle en ingénierie logicielle. Elle permet de définir des composants logiciels très généraux (donc réutilisables), puis de construire des systèmes par assemblage automatique des composants et d'obtenir une application efficace. Son but premier n'est donc pas de permettre l'écriture de programmes plus efficaces, mais d'obtenir des programmes plus simples et de réutiliser plus de code. L'évaluation paresseuse peut être offerte très naturellement dans les langages fonctionnels. Étrangement, la situation est très différente dans les langages impératifs, alors que cette technique est pourtant largement répandue (par exemple, les accès aux fichiers utilisent des tampons, les arbres de jeux sont construits incrémentalement, les pages web sont chargées incrémentalement). Dans les langages impératifs, le programmeur est responsable de la programmation manuelle de la paresse. Ce qui est fastidieux et bien sûr source de bogues. L'objectif de cette thèse est de fournir des mécanismes langages et des outils pour l'évaluation paresseuse dans un cadre impératif en général et dans un cadre orienté services en particulier.

Mot clés :

Programmation impérative, paresse, dépendance, sémantique, analyse statique et dynamique, optimisation, programmation distribuée, causalité, architecture orientée services.

Introduction

L'évaluation paresseuse est une technique essentielle en ingénierie logicielle [Hug89]. La paresse fait aujourd'hui un retour en force, et si il n'existe pas de support (c'est bien la le problème) dans les langages les plus répandus, une requête google avec "laziness" et le nom d'un langage retourne plusieurs centaines de milliers de résultats, dont de nombreux blogs de programmeurs (par exemple C++ [Mil14], Java [SPI14], Python [Nai13]) qui expliquent comment l'encoder (mais à ses risques et périls).

Contexte et problématique

La réutilisation de code est une notion fondamentale du génie logiciel. Contrairement à ce que la plupart imagine elle ne permet pas d'obtenir des programmes plus efficaces. Elle permet d'obtenir des programmes plus simples et de réutiliser plus de code [Hug89]. Par exemple, la fonction qui calcule le minimum d'une collection peut être définie comme la composition d'un tri et de l'accès au premier élément de la collection triée :

```
minimum collection = head (tri collection)
```

Cependant, cette définition est inefficace car la fonction générale `tri` effectue de nombreux calculs inutiles dans ce contexte spécialisé d'utilisation. L'évaluation paresseuse rend une telle définition efficace en effectuant uniquement les calcul requis par `head`. Des définitions simples et réutilisant beaucoup de code mais totalement inefficace en évaluation stricte deviennent raisonnables et donc simplement possibles en évaluation paresseuse.

La paresse peut être très naturellement offerte dans les langages fonctionnels (car l'absence d'effets de bord rend les dépendances explicites et donc le réordonnancement des calculs possibles). La paresse n'est pas offerte dans les langages impératifs (car les effets de bord rendent les dépendances implicites et donc le réordonnancement impossible en général). Pourtant, la programmation paresseuse est très utile dans un contexte impératif. Par exemple, un fichier n'est pas lu immédiatement mais par bloc grâce à un tampon (c'est à dire la lecture est paresseuse), un arbre de jeu va être partiellement construit et exploré en alternance, une page web va être chargée incrémentalement. On retrouve la paresse au niveau architectural aussi (par exemple batch *versus* pipe-filter, push *versus* pull [SG96]).

Problèmes et opportunités

Dans un contexte impératif, le programmeur est responsable de programmer la paresse manuellement. Ceci est fastidieux, aussi les programmes ne sont pas aussi paresseux qu'ils pourraient l'être. Ceci est dangereux, car le programmeur peut créer des bogues si un calcul retardé est évalué plus tard qu'il ne devrait l'être.

En conséquence, le programmeur définira et utilisera peu de paresse et donc peu de composants logiciels très généraux (qui de par leur généralité nécessitent de la paresse pour être performants).

Dans un contexte distribué le problème est le même (sauf que dans ce cas la notion de séquence de calculs et de dépendance entre calculs devient causale [Lam78]). Les architecture orientées services reposent sur la composition (orchestration et choréographie) de services. Mais, là encore, pour des raisons d'efficacité les services ne sont pas aussi généraux qu'ils pourraient l'être.

Si les dépendances entre bloc impératifs pouvaient être déclarées (ou inférées) un algorithme [DLL09, DT14] pourrait réaliser l'évaluation paresseuse

en retardant ou forçant l'évaluation en cascade de tels blocs.

De plus, puisque les calculs sont réifiés (afin de les retarder) l'algorithme pourrait aussi introduire des optimisations dynamiques [GLJ93]. Par exemple, quand une collection doit être triée deux fois, un tri est suffisant : les deux blocs retardés peuvent être remplacés par un seul.

Travail demandé

Objectifs

L'objectif de ce travail est de proposer un outil pour soutenir la programmation paresseuse dans un cadre orienté services. Cet outil reposera sur des annotations d'effets pour chaque service et réalisera la composition dynamique (ou statique). Il permettra aussi de considérer des optimisations dynamiques. Un tel outil augmentera la réutilisation de composants logiciels en découplant la composition (généralisation) des contraintes d'efficacité (spécialisation).

Plan de travail prévisionnel de l'étude

Le travail consiste en:

- étudier la paresse dans les langages fonctionnels.
- identifier les utilisations et les opportunités d'utilisation de la paresse dans des applications impératives / orientées services bien établies (par exemple en inspectant des applications open sources reconnues).
- évaluer les techniques de programmation paresseuse dans un contexte impératif / orienté services.
- fournir des moyens de spécifier des dépendances dans un cadre impératif / orienté services.
- fournir des moyens de retarder, forcer, déclencher en cascade des évaluations dans un cadre impératif / orienté services.
- utiliser ces moyens pour remplacer la paresse manuelle et introduire plus de paresse dans les applications précédemment étudiées.
- fournir des moyens d'exprimer et d'effectuer des optimisations de séquences retardées.
- introduire des optimisations dynamiques dans les applications précédemment étudiées.

Candidats

Compétences

Le candidat doit être passionné par les langages de programmation aussi bien d'un point de vue pratique que théorique. Il devra être capable de répondre à des questions sur [DT14] qui sert de point de départ à ce travail.

References

[DLL09] Rémi Douence, Xavier Lorca, and Nicolas Loriant. Lazy composition of representations in java. In Alexandre Bergel and Johan Fabry, editors, *Software Composition*, volume 5634 of *Lecture Notes in Computer Science*, pages 55–71. Springer, 2009.

[DT14] Rémi Douence and Nicolas Tabareau. Lazier Imperative Programming. In *Principles and Practice of Declarative Programming (PPDP)*, Canterbury, Royaume-Uni, September 2014.

[GLJ93] Andrew J. Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In *FPCA*, pages 223–232, 1993.

[Hug89] John Hughes. Why functional programming matters. *Comput. J.*, 32(2):98–107, 1989.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[Mil14] Bartosz Milewski. Getting lazy with c++. *Blog*, April 2014.

[Nai13] Vineet Naik. Python generators and being lazy. *Blog*, March 2013.

[SG96] Mary Shaw and David Garlan. *Software architecture - perspectives on an emerging discipline*. Prentice Hall, 1996.

[SPI14] SPIRiT_1984. Will java 8 have lazy evaluation? *Stackoverflow*, February 2014.