Lazi / informatique : compilateur ouvert

Emmanuel Chantréau

2 décembre 2016

Table des matières

1 Introduction

Nous présentons ici l'intérêt et la définition des compilateurs ouverts.

2 Évolution des langages informatiques

Les premières machines ont été programmées directement en langage machine. L'évolution des langages a permis de représenter des systèmes de plus en plus éloignés du langage machines. La structure et la logique (avec les types) ont pris une place toujours plus grande. Avec les langages fonctionnels pures est apparu une étape supplémentaire dans l'évolution : alors que le langage machine ne contient que des effets de bord, une partie du traitement d'information en est complètement isolé. En prenant un peu de recul on peut voir cette évolution ainsi : d'abord de l'action pure, puis de la pensée et de l'action entremêlés et enfin séparation de la pensée et de l'action. Ce qui est normal pour nous humain (pouvoir penser sans pour autant être obligé d'agir en même temps) est tout nouveau en informatique.

Pour arriver à cette dernière étape, il a fallu répondre à la question « Comment isoler la pensée de l'action puisque le but de la pensée est de commander l'action ? » Étant donné que la pensée ne produit que des idées (que l'on peut dénommer aussi « représentation »), la solution est contrainte : il faut que la pensée produise des représentations d'actions qui seront utilisées pour générer les actions.

En Haskell les monades sont utilisées pour représenter les actions et par exemple tout programme Haskell doit produire une monade « IO t ».

Donc en résumer, nous sommes partis de langages décrivant directement le programme à des langages où le programme est le résultat de calculs dans un langage proche des mathématiques.

3 Le sens de l'histoire de la programmation

Si on veut concevoir un logiciel d'architecture, on sait qu'il suffit d'aider à concevoir des représentations de divers matériaux de diverses formes assemblés de diverses manières. Mais si on veut concevoir un compilateur toute limite dans la forme du programme sera une contrainte artificielle car un programme est, comme la pensée, un traitement de l'information : on ne peut pas préjuger de limitations naturelles.

C'est pourquoi le sens de l'histoire est de donner toujours plus d'ouverture dans les structures permettant de concevoir des logiciels. Ainsi il a fallu attendre Haskell et les monades pour pouvoir concevoir des analyseurs syntaxiques simples et puissants.

Donc si on veut concevoir un compilateur sans limite artificielles il faut fournir un langage :
— Ne limitant pas la partie « pensée structurée ». Hors le langage le moins limité que l'on connaisse est le langage mathématique (comme Lazi).
— Ne limitant pas la représentation des actions.

4 Comment ne pas limiter la représentation des actions

Le produit final d'un compilateur est une arborescence de fichiers. Donc un compilateur n'imposant pas de limite laissera un contrôle total sur l'arborescence et le contenu des fichiers.

Comment alors fournir à la fois un contrôle total et des services de haut niveau ?

Le but en programmation fonctionnelle pure est de calculer une représentation d'un programme. Le but d'un compilateur est de calculer le langage machine d'un programme. Nous voyons qu'il y a une similitude entre les deux que nous pouvons assembler en un seul : le programme calcul directement les fichiers binaires, le compilateur étant fourni sous forme de librairies aidant à traduire des abstractions de calculs en abstractions de plus bas niveau, jusqu'à atteindre le langage machine.

5 Définition d'un compilateur ouvert

Remarque : nous entendons ici par langage tout type servant à représenter plus ou moins abstraitement du code exécutable.

Un compilateur ouvert est :
— un ensemble de librairie fournissant des fonctions de traduction de différents langages et tel que :
  — Au moins un des langages est du langage machine.
  — Tout langage a pour finalité d'aboutir à une traduction en langage machine.
— Un interpréteur du code source.

L'exécutable est donc produit en interprétant le code source, le résultat étant sous forme d'un tableau associatif chemin de fichier/contenu. Ce code source a pour seul fonction d'entrée/sortie la fonction « fileContent path » qui retourne comme valeur le contenu du fichier « path » (l'équivalent du « include » ou « import »).

6 Exemples de code source

6.1 « Hello world » avec un compilateur ouvert

Le programme source pourrait être :

```
toExec "x86_64" "ELF" "test" Ol
  $Cp[
    print "Hello Word"
  ]
```

où toExec est une fonction de traduction en exécutable prenant en argument le processeur, le format de l'exécutable, le nom du fichier exécutable, des définitions (ici vide) et une représentation abstraite de l'exécutable.

6.2 Inverser une chaîne de caractères avec un compilateur ouvert

Le programme à produire lit une chaîne de caractères dans l'entrée standard et la réécrit inversée dans la sortie standard. L'intérêt par rapport à « Hello world » est qu'un calcul est réalisé en plus d'une entrée/sortie.

```
$Def defs = $Lazi[
/*
Retourne une liste : le début devient la fin
*/
$Def listReverse = $F l -> listRevFold addToList 0l l
]

toExec "x86_64" "ELF" "test" defs
$Cp[
let a = readline STDIN ,
print STDOUT . listReverse a
]
```

Le code Lazi destiné à être compilé est séparé du code Lazi servant à produire l'exécutable (cela n'empêche pas d'avoir du code commun, il suffit d'utiliser des systèmes d'import de fichiers sources).

Un élément de la liste « $Cp » représente un « bout de code », c'est l'équivalent des monades Haskell.

7 Quel intérêt ?

7.1 La modularité

Chaque fois que l'on offre un système modulaire (comme les smartphones ou les navigateurs web permettant l'installation de modules) apparaît une inimaginable diversité de modules.

Associé à un langage mathématique permettant d'exprimer totalement les contraintes sur les fonctions, il devient possible d'accepter tout module sans craindre d'effet de bord (alors que la modification des compilateurs actuels est réservé aux spécialistes du compilateur concerné).

On peut imaginer par exemple des modules permettant :
— de s'adapter aux contraintes de la programmation en temps réel pour des processeurs peu puissants,
— les calculs massivement parallèles

7.2 La pérennité

Faire évoluer le compilateur revient à fournir des librairies supplémentaires offrant de nouvelles syntaxes. Il n'y aurait pas besoin d'avoir à gérer différents compilateurs pour différents langages car un langage et un compilateur extensibles n'ont pas besoin d'être remplacés mais seulement étendus. La formation des développeurs consisterait uniquement en l'apprentissage d'abstractions et de syntaxes supplémentaires.

7.3 Facilité de développement

On pourrait imaginer la tâche monumentale mais le côté modulaire et mathématique rend les choses simples. On peut commencer par les librairies proches du langage machine et augmenter petit à petit en abstraction. On peut ainsi faire des testes dès le début.