Langage de programmation

Georges Dupéron

12 mai 2010

Table des matières

Table des matières **1**

1 Objectif **2**

2 Recherche **2**
2.1 Programme en dataflow 2
2.2 Caractéristiques du programme 2
2.3 Généralisation 3
2.4 Base 4

3 Notes pour la suite... **4**

Références **5**

1 Objectif

Le but de ce projet est de mettre au point un langage de programmation utilisant le paradigme du dataflow[1], n'ayant pas de primitives fixes.

2 Recherche

2.1 Programme en dataflow

Examinons un programme simple exprimé dans le paradigme du dataflow :

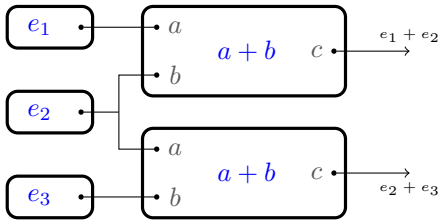

FIG. 1 – Un programme simple en dataflow.

Ce programme a trois entrées (e_1, e_2, e_3), et deux sorties ($e_1 + e_2$ et $e_2 + e_3$).

2.2 Caractéristiques du programme

La question à se poser maintenant est :

Qu'est-ce qui définit ce programme ?

Ou, comme le disait un des collègues de Joe Armstrong [1] :

> « A program is a black box. It has inputs and it has outputs. And there is a functional relationship between the inputs and the outputs. What are the inputs to your problem ? What are the outputs to your problem ? What is the functional relationship between the two ? »

Ce programme est constitué
- d'entrées,
- de sorties,
- d'un graphe étiqueté exprimant la relation entre les sorties et les entrées grâce à des fonctions (+) qu'on suppose déjà définies,
- d'une sémantique d'évaluation, qui donne une signification au graphe,
- d'une représentation graphique,

[1] `http://en.wikipedia.org/wiki/Dataflow`

– Et pour l'exécuter pour de bon, il faut une machine réelle vers laquelle on peut compiler le programme (ou un interpréteur fonctionnant sur cette machine).

2.3 Généralisation

Si on généralise ce résultat, on peut dire qu'un programme est défini par une structure de données abstraite (on n'a pas besoin de connaître la représentation en mémoire de cette structure), qui peut contenir des entrées, des sorties, un arbre (ou graphe) syntaxique, etc. Cette structure peut être représentée sous forme d'un programme dont l'entrée est la désignation d'une partie de la structure que l'on souhite accéder, et dont la sortie est la partie en question.

La représentation graphique ou textuelle du programme peut être assurée par un autre programme.

Sa sémantique d'évaluation peut être définie par une machine abstraite, dont les (méta-)entrées sont le programme, ainsi que des entrées pour le programme, et dont les (méta-)sorties sont les sorties que le programme devrait avoir. Tiens ? Entrées, sorties, une relation fonctionnelle (les sorties sont celles du programme pour ses entrées), ... Eh oui, notre machine abstraite, c'est-à-dire notre sémantique d'évaluation est bel et bien un programme elle aussi.

De même, la machine réelle vers laquelle on éspère pouvoir compiler le programme, peut être modélisée par un programme. Pendant que nous y sommes, rien ne justifie la présence de la machine réelle, car la machine abstraite pourrait très bien être la même que celle qui exécute le programme. C'est le cas par exemple si notre langage est le code machine d'un certain processeur : L'octet 0x12345678 a pour signification «diviser l'accumulateur par 2», c'est une définition de la sémantique du langage, et à la fois une définition de la machine qui exécute le programme.

La machine abstraite n'a donc pas besoin d'être si abstraite que ça, et pourrait être n'importe quelle machine, et il pourrait même y avoir plusieurs machines qui spécifient la sémantique du langage (de manière redondante, pour avoir le choix, et pour que ces définitions se vérifient mutuellement).

Et, pour continuer sur cette voie, il peut aussi y avoir plusieurs représentations syntaxiques (textuelle, avec ou sans coloration, graphique, ...).

On a donc les équations suivantes dans le cas simple (une seule machine, une seule représentation) :

$$\text{Programme} = \left\{ \begin{array}{ll} & \text{Type de données abstrait (ADT)} \\ + & \text{Machine abstraite (sémantique)} \\ + & \text{Représentation} \end{array} \right\} \tag{1}$$

$$\text{Programme} = \text{Programme} + \text{Programme} + \text{Programme} \tag{2}$$

$$\text{Programme} = 3 \times \text{Programme} \tag{3}$$

2.4 Base

Nous voilà bien avancés... Un programme est un programme. C'est donc une définition récursive. Et toute définition récursive doit avoir une base, et une règle pour générer de nouveaux éléments. Nous venons de définir la règle, cherchons les bases possibles :

- Machine de Turing
- Lambda-calcul
- Langage mathémathique

Le lambda-calcul et la machine de Turing sont équivalents[2], par contre le langage mathémathique permet d'exprimer des fonctions non-calculables, des ensembles infinis, et tout un tas de choses obscures. Comme nos machines physiques actuelles sont une version bâtarde des machines de Turing (qui n'ont pas de limite sur la quantité de mémoire disponible, contrairement aux notres), il semble sage de laisser de côté le langage mathémathique (pour l'instant, dans quelques années, peut-être qu'il sera temps de l'ajouter).

L'équivalence λ-calcul vs. Turing nous laisse le choix pour l'implémentation de notre première machine à partir de laquelle les autres seront définies, directement ou indirectement. Explorons donc la suite du problème avant de prendre une décision. À terme, le meilleur sera probablement d'implémenter les deux, comme base, et de les définir mutuellement l'une à partir de l'autre, pour avoir une vérification.

3 Notes pour la suite...

Nous allons prendre un programme en dataflow, et le déconstruire le plus possible, afin de voir quelles sont les «primitives» sémantiques nécessaires à notre langage. Bien que FORTH n'ait pas à proprement parler de primitives, il a lui aussi une sémantique (chaque mot est expansé en la suite de mots le définissant, jusqu'à arriver au code machine).

Il faut définir un bloc eager-evaluation, qui prend en paramètre un graphe, et

- Soit le réécrit (compilation)
- Soit l'évalue (interprétation)

Dans le cas où on compile, on aura une "instruction" call-bloc

call-bloc prend en paramètre des fonctions permettant de calculer ses paramètres, ainsi que les paramètres eux-mêmes.

- En évaluation paresseuse, on n'évalue les paramètres que s'ils sont nécessaires.
- En évaluation «eager», on évalue les paramètres au début de l'appel du bloc, et on stocke leur valeur pour une future utilisation (ou non).
- Pour une macro, on stocke juste les paramètres eux-mêmes (avec leur fonction d'évaluation, s'il y en a une).

[2] `http://en.wikipedia.org/wiki/Lambda_calculus#Computable_functions_and_lambda_calculus`

Références

[1] Peter Seibel. *Coders at Work : Reflections on the Craft of Programming*, page 217. APress, broché edition, 2009. Citation de Joe Armstrong.