

Langage de programmation

Georges Dupéron

16 mai 2010

Table des matières

| | | |
|----------|---|----------|
| 1 | Objectif | 2 |
| 1.1 | Sujet | 2 |
| 1.2 | | 2 |
| 2 | Formalisation du langage | 3 |
| 2.1 | Programme en dataflow | 3 |
| 2.2 | Caractéristiques du programme | 3 |
| 2.3 | Généralisation | 4 |
| 2.4 | Base | 5 |
| 3 | Notes pour la suite... | 5 |
| 4 | Implémentation | 6 |
| A | notes | 6 |
| A.1 | Historique | 6 |
| | Références | 6 |

1 Objectif

1.1 Sujet

Il y a environ 20 ans, le langage et le système d'exploitation FORTH avaient été mis au point, avec pour but de créer un environnement totalement personnalisé pour chaque utilisateur. La particularité de FORTH était qu'il ne possédait pas de mots-clés, ou instructions figées, et que chaque utilisateur était en mesure de définir lui-même ses propres primitives, voire redéfinir ses primitives... à l'infini.

L'échec de FORTH est venu, entre autres de la nécessité d'échanger des programmes entre utilisateurs, et des conflits dus à l'homonymie (même nom, fonction différente) et à la synonymie (même fonction, noms différents). Les fonctions n'étaient pas toujours documentées, ce qui fait qu'un même programmeur ne pouvait pas faire exécuter, à un intervalle de temps relativement court, 2 fois le même programme...

Pour autant, le côté adaptatif et souple de FORTH aurait été largement plébiscité s'il n'y avait eu cette difficulté. Une manière de contourner ce genre de conflit, dû à une représentation symbolique textuelle trop fortement contrainte par la syntaxe, est de se pencher vers une programmation qui privilégie les flux de données sur les actions à réaliser, et vers des bases graphiques plutôt que textuelles, laissant donc à chaque utilisateur la liberté de définir ses actions, et préservant en revanche les flux.

L'idée de ce projet est de proposer une première architecture de compilateur ou d'interpréteur de premier niveau pour illustrer ce paradigme, et tenter d'en évaluer les propriétés. Ce projet nécessite un excellent niveau en programmation et un goût prononcé pour l'écriture de compilateurs.

1.2 ...

Le but de ce projet sera donc dans un premier temps de définir un langage de programmation visuel utilisant le paradigme du dataflow¹, et n'ayant pas de primitives fixes. Dans un second temps, nous implémenterons un EDI² permettant de créer des programmes dans ce langage, et de les interpréter.

Des recherches ont montré que dans le cadre des langages de programmation visuels, l'éditeur jouait un rôle aussi important que le langage lui-même.

DSL = bien mais pas bien. Langages existants : ajout au vocabulaire (fonctions, variables), mais pas à la grammaire (structure) (sauf macros, mais pas complètement). Nous on propose de permettre d'écrire de n'importe quelle manière (grammaire libre), tout en gardant la possibilité d'ajout au vocabulaire. Nous ne gardons que les barrières nécessaires à une certaine cohérence (des éléments graphiques connectés à d'autres). Les blocs peuvent être ronds, les traits annotés d'étiquettes, ou encore d'autres choses.

¹<http://en.wikipedia.org/wiki/Dataflow>

²Éditeur de Développement Intégré

Graphes avec noeuds étiquetés et arcs éiquetés à chaque extrémité (et sur l'arc lui-même) => graphes généralisés.

Preuve de complétude de Turing : un graphe = des noeuds connectés par des arcs, un noeud = l'arc NULL ou qqch du genre. // $\lambda(\lambda(\dots))$.

L'interface utilisateur devient alors la même chose (fenêtres = blocs avec un affichage particulier).

2 Formalisation du langage

Dans cette section, nous allons essayer de trouver quelle est la nature, l'essence d'un programme, de manière à

2.1 Programme en dataflow

Examinons un programme simple exprimé dans le paradigme du dataflow :

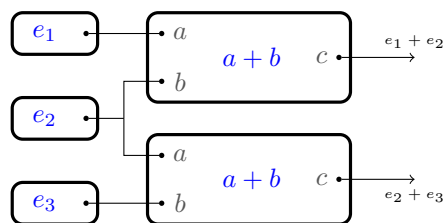


FIG. 1 – Un programme simple en dataflow.

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 [2] :

« 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,
- 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 souhaite 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 espère 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) :

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

$$\text{Programme} = \text{Programme} + \text{Programme} + \text{Programme} \quad (2)$$

$$\text{Programme} = 3 \times \text{Programme} \quad (3)$$

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 des règles pour générer de nouveaux éléments. Nous venons de définir les règles, cherchons les bases possibles :

- Machine de Turing
- Lambda-calcul
- Langage mathématique

Le lambda-calcul et la machine de Turing sont équivalents^{3,4}, par contre le langage mathématique 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 nôtres), il semble sage de laisser de côté le langage mathématique (pour l'instant, lorsque le langage aura gagné en maturité, 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.

³http://en.wikipedia.org/wiki/Lambda_calculus#Computable_functions_and_lambda_calculus

⁴Bien qu'ils ne semblent pas être totalement équivalents[1] :

However, [Lambda Calculus] is not a model of computation for we cannot calculate an upper bound on resource consumption of its reduction steps without resorting to another model of computation, such as [Turing Machines] (according to Yuri Gurevich).

- 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).

4 Implémentation

Pour l'implémentation, nous nous limiterons à un sous-ensemble purement fonctionnel du langage défini dans les sections suivantes.

A notes

- code bubbles
- dataflow
- world machine
- vidéo alan kay
- pas de conflits de nommage
- gruntnetwork.com
- Design patterns
- Domain Specific Language
- DSL
- vue «libre»
- thèse sur la programmation par l'exemple
- Preuves
- Spécification fonctionnelle

A.1 Historique

- Control flow graphs
- alsa «composer»
- <http://recherche.ircam.fr/equipes/repmus/RMPapers/CMJ98/>
- Pourquoi dataflow peu utilisé? (chest hair?, <http://therighttool.hammerprinciple.com/>, contacter l'auteur pour qu'il ajoute des VPL)

Références

- [1] Koray Can. 10^e commentaire sur <http://is.gd/cb70k>.
- [2] Citation d'un ami de Joe Armstrong. Cité dans [3], page 217. Version électronique : <http://is.gd/caFhh>.
- [3] Peter Seibel. *Coders at Work : Reflections on the Craft of Programming*. APress, 2009. <http://www.codersatwork.com/>.

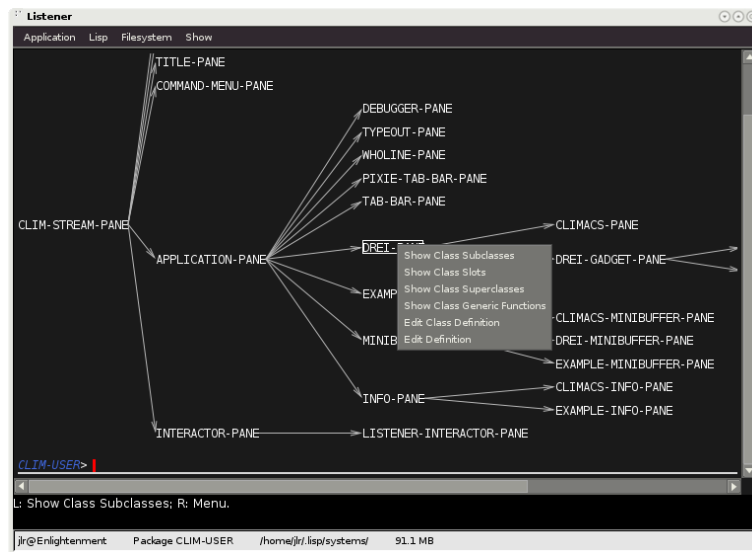


FIG. 2 – Graphe d'héritage des classes en Lisp avec McCLIM