Langage de programmation

Georges Dupéron

17 mai 2010

Table des matières

Première partie

Objectif

1 Sujet : Etude d'un paradigme de programmation : les langages graphiques à Dataflow

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.

2 But du projet

Le but de ce projet sera donc dans un premier temps de définir un langage de programmation visuel utilisant le paradigme du dataflow[1], et n'ayant pas de primitives fixes. Dans un second temps, nous implémenterons un EDI[2] permettant de créer des programmes dans ce langage, et de les interpréter.

[1] `http://en.wikipedia.org/wiki/Dataflow`
[2] Environnement de Développement Intégré

Deuxième partie

Définition du langage

3 Étude de l'existant

3.1 FORTH

[3] Le langage FORTH est basé sur le principe de l'expansion de macros : lorsqu'on «appelle» une macro, elle s'expanse, ses sous-macros s'expansent, et ainsi de suite, jusqu'à ce que des macros de bas niveau lisent des valeurs sur la pile (pop), et en écrivent d'autres à la place.

Mon expérience personnelle, lorsque j'ai essayé par le passé d'implémenter des algorithmes un peu complexes en TEX(le langage sur lequel est construit LATEX), m'a montré que cette approche a un défaut fondamental : il est impossible de délimiter le rayon d'action d'une macro. Étant donné qu'elle travaille sur la pile (ou dans le cas de TEX, sur les jetons qui suivent l'appel de macro), et qu'elle peut enlever ou ajouter autant d'éléments qu'elle veut sur la pile, il y a un risque assez grand pour qu'une macro dont on ne connaît pas bien le fonctionnement soit mal utilisée, et abîme la portion de pile qui appartient à d'autre macros.

Vu que les macros peuvent être imbriquées autant qu'on veut, il est possible qu'une macro située très bas dans l'arbre fasse ce genre d'erreur. Le débogage nécessite alors de parcourir tout l'arbre d'appels de macros à la recherche de l'erreur.

Une représentation graphique permet d'indiquer explicitement quelles sont les données fournies à une fonction ou une macro, et il devient alors possible de mettre en place des mécanismes de contrôle de l'utilisation de la pile, au moins pour le débogage. Sans cette approche, il n'y a aucun moyen de savoir quels sont les données passées en paramètre et quelles données doivent rester sur la pile, du point de vue de la macro appellante.

Dans le cas de FORTH, il n'y a pas de primitives dans le langage. Cela signifie que si on regarde récursivement le code des macros, il n'y a pas un moment où on va tomber sur une macro qui n'a pas de définition, mais est implémentée par le compilateur ou l'interpréteur (définition dans un autre langage, et probablement inaccessible pour peu qu'on n'ait pas le code source du compilateur). À la place, aux feuilles de l'arbre d'expansion des macros, on trouvera des instructions du langage machine.

Ce comportement est fortement souhaitable de la part d'un langage généraliste, du point de vue éducatif : les curieux peuvent farfouiller jusqu'aux racines du langage et ainsi comprendre comment fonctionne les programmes, à tous les niveaux. Certains programmeurs pensent que c'est important de comprendre en profondeur comment fonctionne la machine et les programmes qui l'utilisent[1].

[3] Les bibliothèques universitaires de Montpellier n'ont aucun ouvrage concernant forth. Il faudrait peut-être suggérer quelques achats aux bibliothécaires.

Cela confère aussi au langage une grande portabilité : Forth a été porté sur plus d'une vingtaine de processeurs[?]. En définissant un ensemble de fonctions de haut niveau que chaque implémentation doit fournir, on peut s'assurer qu'un programme fonctionnant sur une implémentation du langage fonctionnera aussi sur une autre.

3.2 Dataflow

Le paradigme du dataflow[2] (flux de données) est connu des concepteurs de langages de programmation depuis longtemps. Il a été utilisé avec succès dans certains domaines, principalement des domaines intéressant les non-programmeurs.

3.2.1 Graphisme

Le logiciel Quartz Composer sous MacOS permet la création d'images vectorielles animées et interactives sous MacOS de manière graphique : On applique des filtres graphiques, représentés par des boîtes, aux résultats d'autres filtres, en connectant les boîtes entre elles. Dans World Machine, ces filtres sont des actions physiques (érosion, soulèvement) et permettent de générer des cartes de hauteur (heightmaps), qui sont utilisées pour modéliser des terrains en 3D.

3.2.2 Musique

L'interface de certains logiciels de musique s'inspirent de l'architecture des synthétiseurs modulaires[3] (ces grosses boîtes avec pleins de prises jack qu'on relie avec des câbles : chaque prise jack est une entrée ou une sortie d'un module, les câbles sont les connexions). Un des premiers synthétiseurs virtuels, Max/MSP, utilisait cette analogie. D'autres logiciels similaires lui ont succédé : PureData (qui est aussi un langage de programmation généraliste), Alsa Modular Synth, ...

3.2.3 Mesures scientifiques

LabView permet aux scientifiques de procéder à des traitements sur les signaux et les données acquises depuis un ordinateur, grâce à un langage de programmation graphique utilisant le paradigme du dataflow.

3.2.4 Traitement de signaux

Force est de constater que tous les exemples cités ci-dessus sont des cas particuliers de traitement de signal (image, son, signaux provenant d'appareils de mesure). Le paradigme du dataflow devrait être tout aussi efficace pour construire des programmes conventionnels : les signaux d'entrée sont les évènements provoqués par l'utilisateur (clic de souris, appui sur le clavier), ceux de sortie sont les retours (écran, haut-parleurs, ...).

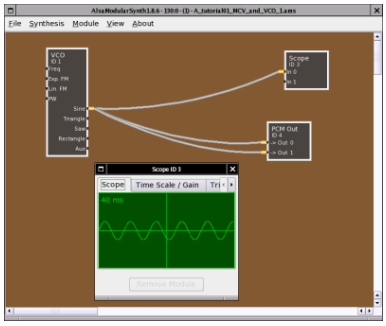

FIG. 1: Alsa Modular Synth[4]

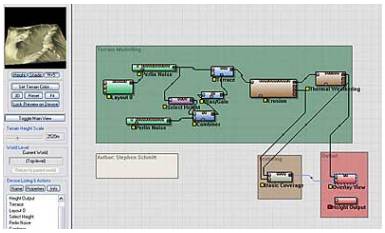

FIG. 2: World Machine[5]

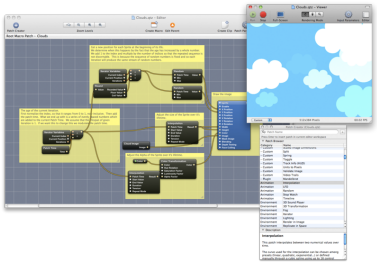

FIG. 3: Quartz Composer[6]

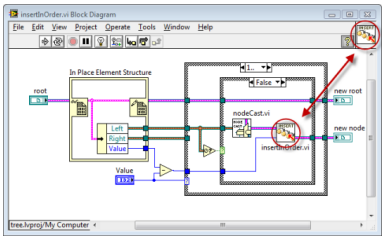

FIG. 4: LabView[7]

FIG. 5: Captues d'écrans de quelques langages de programmation utilisant le paradigme du dataflow.

3.3 langages visuels

3.3.1 Acceptation des langages visuels par les programmeurs

Les langages visuels n'ont eu que peu de succès auprès de la communauté des programmeurs, et les raisons de ce rejet méritent d'être étudiées[4].

Récemment, un chercheur a mis en place un sondage auprès des programmeurs qui devrait à terme permettre de savoir quels langages correspondent le mieux à quelles affirmations, selon les programmeurs[8]. Ces affirmations sont du type «Ce langage est facile à utiliser» ou «Ce langage à une bonne communauté». Les affirmations qui obtiennent les moins bons scores pour un langage donné indiquent en général les défauts de ce langage. J'ai donc contacté l'auteur de ce sondage pour lui demander d'ajouter des langages de programmation visuels pendant que le sondage est encore ouvert, afin de pouvoir étudier par la suite ces résultats.

[4]Peut-être que c'est simplement que les langages textuels, c'est pour les programmeurs avec du poil sur le torse, et les langages graphiques sont pour les mauviettes...

3.3.2 Historique

On a toutefois de noubreux cas d'utilisation de représentations visuelles dans la programmation : Les langages suivant le paradigme du dataflow, dont nous avons parlé ci-dessus, mais aussi l'utilisation d'un formalisme graphique pour la représentation principale des diagrammes UML[5]. Les organigrammes de programmation (control flow diagram) sont aussi utilisés depuis longtemps.

D'autres représentations visuelles sont utilisées dans certains systèmes. La figure 6 montre un graphe d'héritage des classes en Lisp avec l'environnement McCLIM. On trouve aussi des graphes d'appels de fonctions, qui relient deux fonctions si l'une appelle (ou peut appeller) l'autre.

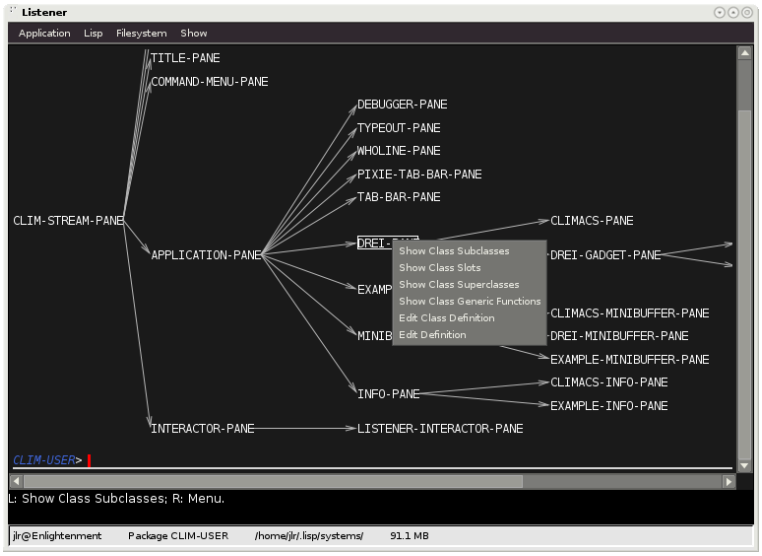

FIG. 6: Graphe d'héritage des classes en Lisp avec McCLIM

3.4 Langages spécifiques à un domaine

Il est fréquent que des programmeurs créent des «langages spécifiques à un domaine» (DSL, Domain Specific Language) pour répondre au besoin d'exprimer facilement des instructions et notions très fortement liées à un certain domaine, sans être encombrés par la syntaxe rigide, le vocabulaire limité et le manque d'expressivité de leur langage habituel. Les fichiers de configuration sont en général écrits dans un DSL par exemple. Mais les DSL ont de grosses limitations : bien qu'ils soient adaptés à l'expression d'idées dans ce domaine, ces langages ne possèdent pas l'expressivité nécessaire à des opérations plus complexes : souvent, ces DSL ne sont complets au sens de turing et n'ont pas accès aux bibliothèques de fonctions des autres langages.

[5] Il existe aussi une représentation textuelle d'UML[9]

C'est pour ces raisons que les DSL sont de plus en plus laissés un peu de côté : La configuration de GRUB 2 est toujours dans un DSL, mais elle est générée dynamiquement par des scripts shell[10], la configuration du gestionnaire de fenêtres awesome est écrite en Lua[11], beaucoup de logiciels utilisent python comme langage de script, plutôt que de créer un langage de script spécifique à l'application.

D'un autre côté, le problème des langages de programmation généralistes est qu'ils permettent d'augmenter le vocabulaire disponible, par l'ajout de fonctions et la création de variables, mais ils ne permettent pas de modifier la grammaire utilisé (la structure du code). En réalité, de nombreux langages permettent cela par le biais de macros, mais cette approche est assez limitée : l'écriture de macros est en général fastidieuse – beaucoup plus difficile que l'écriture de fonctions – et l'utilisation de macros est néanmoins soumise à un grand nombre de contraintes syntaxiques du langage hôte (parenthèses, accolades, ...). De plus le système de macros de certains langages, comme le C, n'a pas la puissance d'un vrai langage : pas de récursivité des macros, pas de structures conditionnelles à l'intérieur d'une macro.

Il y a donc des avantages dans les deux approches, mais aucune ne permet de satisfaire complètement les besoins de facilité d'expression et de puissance du langage. Nous proposons une solution alternative : permettre l'écriture d'un programme sous n'importe quelle forme visuelle (grammaire libre), dans un langage graphique orienté dataflow. Nous imposons quelques contraintes, afin de garder une certaine cohérence :

- Les sous-programmes sont représentés par des éléments graphiques (blocs) ;
- Les paramètres d'entrées de ces sous-programmes sont des entités visuelles qu'on peut connecter entre elles (ports) ;
- Les valeurs de ces paramètres sont fournies à travers des liens qui relient les ports ;

Ces règles laissent toutefois une très grande liberté, les blocs peuvent être rectangulaires, ronds ou de n'importe quelle forme. Ils pourraient par exemple afficher les calculs mathémathiques qu'ils contiennent non pas sous l'apparence d'instructions mais d'une formule mathémathique. On peut aussi faire en sorte que dans un outil de conception d'interfaces utilisateur, les composants graphiques (fenêtre, bouton, etc.) ne soient pas de simples images, mais *soient* les fonctions ou objets correspondants. Dans le logiciel LabView, les «blocs» peuvent revêtir différentes apparences.

Il est même possible que certains blocs aient une apparence invisible. Par exemple les opérations de conversion de types (cast) pourraient être masquées, et ne s'afficher que sur le lien entre la source du transtypage et sa destination.

Avec ces règles, les ports n'ont pas besoin d'être affichés en permanence. Ils peuvent par exemple être dans une boîte de dialogue de configuration dui apparaît lorsqu'on double-clique sur le bloc correspondant (World Machine utilise cette approche, et laisse aussi les ports visibles en mode «réduit»).

Les liens peuvent être annotés, regrouppés en bus, etc. Comme on peut l'imaginer, avec un minimum d'effort de cohérence, ces libertés sont suceptibles de rendre les programmes plus lisibles (non-affichage des éléments qui ne sont

pas importants à la comprhéension du programme) et plus faciles à écrire (les instructions (blocs) spécialisées ont une interface d'utilisation adaptée).

3.5 Environnement de développement intégré

> Clearly good design is as important for visual languages as for textual ones. Furthermore, the effectiveness of a visual language indeed any precise language for specifying structures, depends to a large extent on the quality of the editor, browser, interpreter, debugger and other tools supplied by the language implementation
>
> Christopher C. Risley and Trevor J. Smedley [12]

Les environnements de développement intégré jouent un rôle crucial pour les langages de programmation visuels. En effet, sans la possibilité de construire un programme par une séquence (obscure) de caractères, la convivialité du langage est directement liée aux outils que propose l'EDI. Ces EDI n'ont en général que peu en commun avec les ceux destinés aux langages textuels, et se présentent plus comme des logiciels de dessin vectoriel que comme des éditeurs de texte.

Certaines questions sont récurrentes dans la conception d'EDI pour des langages visuels. Une des plus importante est la gestion de l'espace à l'écran : les primitives graphiques sont décorées de bordures, et d'autres éléments visuels, qui occupent beaucoup d'espace à l'écran, réduisant ainsi la quantité d'informations affichées. Ce problème est plus communément connu sous le nom de «Deutsch Limit»[13] :

> Well, this is all fine and well, but the problem with visual programming languages is that you can't have more than 50 visual primitives on the screen at the same time. How are you going to write an operating system ?

Cependant, dans le cas d'un affichage textuel, le cerveau humain n'est pas capable d'ingurgiter en temps réel toute l'information à l'écran[6]. L'affichage sous forme graphique permet au contraire de clarifier et de nettoyer les informations à l'écran, on peut donc supposer que le cerveau est capable d'analyser plus rapidement du «code» graphique que son équivalent textuel (un dessin vaut mille mots, dit-on).

De plus les programmeurs passent beaucoup de temps à chercher tel ou tel morceau de code ou fonction. L'EDI Code Bubbles[14] apporte une solution à ce problème pour les langages textuels, en fournissant une interface dans laquelle le programmeur manipule un gand nombre de fragments de code assez court, et a la possibilité d'en ouvrir facilement de nouveaux de manière contextuelle (ouvrir la définition de cette fonction, ouvrir la documentation de celle-là). Nous avons repris ce principe pour les langages visuels : Dans la définition d'un bloc, il est possible d'ouvrir sur place ou à côté les définitions des sous-blocs.

[6] Sur mon écran, on peut loger plus de 60 lignes de texte verticalement, et ce sur 2 colonnes. Allez donc analyser 120 lignes de C d'un seul coup.

Une autre solution consiste à utiliser une interface «fish-eye» : les objets au centre de l'écran ont une taille normale, et sont de plus en plus petitsà mesure qu'ils approchent du bord. Une variante est l'interface «zoom», utilisée notamment dans certains jeux vidéos[15] pour permettre au joueur de focaliser son attention sur les ennemis les plus proches, ou bien avoir une vue d'ensemble, et dans le logiciel de prises de notes Project Cecily[16].

Plusieurs recherches ont été menées concernant l'utilisation de l'espace à l'écran par les langages visuels, une bibliographie référencant plusieurs papiers sur ce sujet est disponible à [17].

3.6 Manque d'expressivité des langages de programmation existants

L'utilisation fréquente de «design patterns» ou partons de conception dans les langages de programmation existants peut être considérée comme un indicateur de défauts dans ces langages[18] : s'il y a besoin d'utitliser le design pattern x, c'est que le langage ne permet pas d'utiliser les concepts vehiculés par x de manière plus élégante, c'est donc un manque d'expressivité sur ce point. En général, la manière la plus élégante d'utiliser un concept, c'est quand il est de «première classe».

La solution que nous proposons pour un autre problème d'expressivité, celui des Langages Spécifiques à un Domaine (Section 3.4, page 7), qui était de permettre aux blocs de revêtir n'importe quelle apparence, est aussi une solution aux problèmes soulignés par les Design Patterns. En effet, pour peu que le langage propose de l'introspection et que les éléments de base du langage soient (réellement) de première classe, à savoir les blocs, les ports et les connexions, il est possible de transformer un grand nombre de concepts en concepts de (pseudo-)première classe.

Par exemple, si l'on souhaite implémenter le concept des singletons, on rajoutera par introspection à chaque bloc utilisant ce concept un faux port d'entrée qui représentera directement le singleton. Ou bien on encapsulera tous les blocs représentant des variables qui doivent être uniques, avec une structure qui s'assurera que tout accès à ces blocs se fera sur une unique instance.

3.7 Réutilisabilité

3.7.1 Programmation par composants

Le paradigme du dataflow partage de nombreux points communs avec le paradigme de la programmation orientée composants. Cette approche vise à augmenter la ré-utilisabilité du code en le structurant sous forme de composants qui communiquent entre eux. Dans le paradigme du dataflow, les composants sont clairement visibles : ce sont les blocs, et ils communiquent entre eux à travers leurs ports, par le biais de connexions.

3.7.2 Commandes Unix

Ce principe de composants isolés communiquant au travers d'interfaces prédéfinies se retrouve aussi dans l'architecture Unix. Un des principes de la philosophie unix est «Do one thing, and do it well», ainsi sous Unix, plein de petis programmes différents font chacun une action bien précise, simple. Des actions plus complexes sont réalisées en connectant la sortie de certains programmes à l'entrée d'autres, en général avec un «pipe» (tuyeau).

Ce principe a même été poussé jusqu'à faire un navigateur modulaire, suivant la philosophie Unix, dont la partie chargée d'afficher les pages web est complètement dissociée de la partie qui gère les autres fonctionnalités (historique, boutons de navigation, ...).

Cependant, cette approche est très limitée, car le seul type de données que des programmes peuvent échanger au travers de pipes ou de sockets, c'est un flux d'octets augmenté d'un marqueur de fin (EOF). Il n'y a pas de communication hors-bande, pas de structuration des données. Cela pose des problèmes d'encapsulation (au sens des protocoles de communication : comment envoyer plusieurs flux en parallèle?), d'échappement, et généralement de sérialisation des données – alors qu'on transfère des données d'un programme à l'autre, sur le même système.

Le paradigme du dataflow résoudw ces problèmes en permettant de faire transiter à travers les connexions entre blocs des flux typés, plutôt que de simples octets. De plus, chaque bloc possède plusieurs ports de sortie, ce qui permet de sélectionner la donnée voulue. Sous Unix, pour récupérer la date de modification d'un fichier, il faut découper la sortie de la commande `ls -l` avec d'autres outils, comme `cut`. En dataflow, il suffit de faire une connexion uniquement avec le port de sortie intitulé «taille du fichier».

3.7.3 Conflits de nommage

Les conflits de nommage sont une grosse source de souci dans les langages de programmation textuels conventionnels. Certains utilisent des espaces de nommage pour réduire les risques de conflits, d'autres utilisent les clôtures pour limiter la portée des noms (c'est ce qui est couremment utilisé en JavaScript, par exemple).

Nous proposons une approche différente, qui consiste à stocker avec chaque fonction définie un identifiant unique, comme si le code était stocké dans une base de données. Les appels de fonction font alors référence (de manière cachée) à l'identifiant unique, et non pas au «nom usuel» de la fonction. Cela rend le langage insensible aux conflits de nommage, et robuste aux changements des noms de variables ou de fonction. D'autres personnes ont proposé de stocker du code Java dans une base de données[19].

4 Formalisation du langage

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

4.1 Programme en dataflow

Examinons un programme simple exprimé dans le paradigme du dataflow :

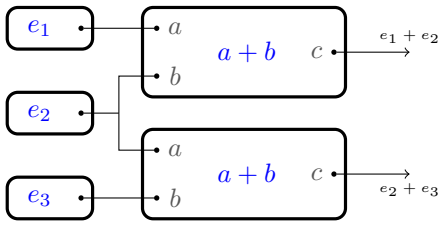

FIG. 7: Un programme simple en dataflow.

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

4.2 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 [20] :

> « 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).

4.3 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) :

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

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

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

4.4 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 mathémathique

Le lambda-calcul et la machine de Turing sont équivalents[7,8], 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, 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.

4.5 Graphe

Du point de vue de notre langage, on peut dire qu'un programme est un graphe dont les noeuds sont étiquetés (les noms ou identifiants des blocs), et dont les arcs sont étiquetés à chaque extrémité (les noms des deux ports auxquels le lien est connecté). Les programmes sont donc des graphes généralisés.

Troisième partie
Implémentation

5 Spécification fonctionnelle

Références

[1] Peter Seibel. *Coders at Work : Reflections on the Craft of Programming.* APress, 2009. `http://www.codersatwork.com/`.

[2] Peter Van Roy. Programming paradigms for dummies : What every programmer should know. In G. Assayag and A. Gerzso, editors, *New Compu-*

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

[8] Bien qu'ils ne semblent pas être totalement équivalents[21] :

> 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).

tational Paradigms for Computer Music. IRCAM/Delatour, France, 2009. http://www.info.ucl.ac.be/~pvr/VanRoyChapter.pdf.

[3] Jacques Bon. Petite histoire du synthétiseur. http://cafcom.free.fr/ams/ams1.html, section intitulée «Débuts de l'informatique musicale».

[4] Jacques Bon. Alsa modular synth. http://cafcom.free.fr/ams/ams2.html.

[5] Stephen Schmitt. Fonctionnalités de world machine, http://www.world-machine.com/features.html.

[6] Capture d'écran sur la page wikipédia «Quartz Composer», http://en.wikipedia.org/wiki/Quartz_Composer.

[7] Creating recursive VIs. (Logiciel de mesure LabView) http://zone.ni.com/devzone/cda/tut/p/id/9387.

[8] The right tool. http://therighttool.hammerprinciple.com/ est un sondage cherchant à déterminer quelles affirmations correspondent le mieux à quels langages de programmation.

[9] Object Management Group. *OMG Human-Usable Textual Notation (HUTN) Specification, version 1.0*, August 2004. Document formal/2004-08-01, http://www.omg.org/cgi-bin/doc?formal/2004-08-01.

[10] *grub.cfg : le fichier de menu de Grub 2.* http://grub.enbug.org/grub.cfg.fr.

[11] Awesome. http://awesome.naquadah.org.

[12] Christopher C. Risley and Trevor J. Smedley. Visualization of compile time errors in a java compatible visual language. In *VL*, pages 22–29, 1998. [9]

[13] Deutsch limit. http://en.wikipedia.org/wiki/Deutsch_Limite.

[14] Code bubbles : Rethinking the user interface paradigm of integrated development environments. http://www.cs.brown.edu/people/acb/codebubbles_site.htm.

[15] Mutant storm reloaded. Utilise une interface «zoom» http://pompomgames.com/mutantstormreloaded.htm.

[16] Project cecily. issu de [24] http://www.osmosoft.com/cecily/.

[17] Effective use of screen real estate. (bibliographie, partie de [22]) http ://web.engr.oregonstate.edu/ burnett/vpl.html#V6C2.

[18] Software patterns as a symptom of failure? : If you have to use them, maybe your programming language is just not powerful enough ? http://broadcast.oreilly.com/2010/02/software-patterns-as-a-symptom.html.

[19] Source code in database. http://mindprod.com/project/scid.html.

[9] Ce papier n'est pas en accès libre. Par conséquent, je ne peux pas vérifier sont contenu. Protestons contre les papiers payants.

[20] Citation d'un ami de Joe Armstrong. Cité dans [1], page 217. Version électronique : http://is.gd/caFhh.

[21] Koray Can. 10^{e}commentaire sur http://is.gd/cb7Ok.

[22] Visual programming languages research papers. (bibliographie) http://web.engr.oregonstate.edu/~burnett/vpl.html.

[23] Visual programming languages bibliography : A branch of the visual language research bibliography. (bibliographie) http://www-ist.massey.ac.nz/plyons/776/vpl%20papers.html.

[24] Tiddlywiki. http://tiddlywiki.com/.

[25] Uzbl. Un navigateur suivant la philosophie Unix. http://www.uzbl.org/.