UPMC

Master informatique 2 – STL
NI503 – CONCEPTION DE LANGAGES
Notes II

Basile STARYNKEVITCH, http://starynkevitch.net/Basile/ *
travaillant au CEA, LIST sur **gcc-melt.org**
reprenant le cours de
Pascal MANOURY, http://www.pps.univ-paris-diderot.fr/~eleph/

2013-2014

courriel : **basile@starynkevitch.net** *et* **basile.starynkevitch@cea.fr**
forum : conception-langage-upmc2013@googlegroups.com

Rappels

- de l'importance des **arbres de syntaxe abstraits** *(**A**bstract **S**yntax **T**rees)*.
- L'**injection de code** est généralement une faute professionnelle - à éviter absoluement dans tout code en production (SQL injection, oubli d'échappement - par exemple dans une chaîne passée à `system(3)`, etc ...). Voir **`$*`** vs **`$@`** dans BASH.
- Les occurrences d'appels récursifs terminaux sont *syntaxiquement* détectables.
- La pile d'appel contient des cadres d'appels (variables locales, arguments, adresse de retour).
- Le λ-calcul formalise les fonctions, et permet d'abstraire une fonction ; la notation $\lambda x.x+2$ signifie la fonction $x \to x+2$ (et x est une variable liée par le λ). Distinguer les variables liées des variables libres. Tous les λ-termes ne sont pas normalisables (par exemple $(\lambda x.xxx)(\lambda x.xxx)$ etc...). On peut considérer un λ-calcul simplement typé.
- Ocaml est un langage de programmation fonctionnel, statiquement typé, avec inférence de types (ou synthèse de type) à la Hindley-Milner.
- Les fonctions `setjmp` et `longjmp` fournissent des sauts non-locaux. Voir `setjmp(3)` ; préférer `siglongjmp(3)` avec les signaux
- Dans beaucoup de langages de programmation, la portée d'une variable (ou "scope") est généralement "lexicale". Les variables dynamiques sont rares...

*Toutes les notes de cours sont disponible sur mon site web.

— une fermeture (ou clôture, “closure”) regroupe du code et des données (les variables closes ou libres, ou “up-values”), c’est ainsi qu’elles sont semblables aux objets de la programmation orientée objet.
— Un language de programmation est **homoïconique** si son AST est une donnée facilement manipulable du language.

1 Éléments de sémantique formelle

Plusieurs approches à la sémantique d’un langage de programmation (cf APS 1, J.MALENFANT et B.S.) :
— **sémantique *dénotationnelle*** (ou “mathématique”, ou de Scott-Strachey). On construit une *dénotation*, le plus souvent définie compositionnellement, sur les ASTs. Voir aussi Théorème de Frege (1884) ;
— **sémantique *opérationnelle*** ; la sémantique opérationnelle *structurelle* est à petit pas (“small step”) car elle définit compositionnellement [1] le comportement d’un programme par pas élémentaires sur les constructions syntaxiques élémentaires. On utilise donc des règles de ré-écriture sur l’état d’une machine abstraite.
— **sémantique *axiomatique*** On décrit en logique de Hoare des assertions en formules logiques concernant les pré- et post- conditions des instructions du programme. Il existe même des outils logiciels libres pour vérifier plus ou moins automatiquement ces assertions, comme FRAMA-C et son langage d’annotation ACSL (wikipage ACSL).

1.1 Pour continuer la sémantique d’une “calculette”

Introduisons les variables dans la syntaxe abstraite

$$\begin{aligned} e &::= \ldots \text{comme avant} \\ &::= \texttt{var} \\ &::= \texttt{var} \leftarrow e \end{aligned}$$

Comme en *C* ou *Scheme* l’affectation est une expression ...

1.1.1 Sémantique des variables

Il nous faut l’ensemble Id des identifiants de variables, et un *environnement* qui est une fonction $Id \to \mathbb{R}$, Notons $Env = \mathbb{R}^{Id}$ le domaine des environnements. Il intervient dans la sémantique de nos expressions (une expression e a une valeur $[\![e]\!]_E$ dans un environnement E, donc $[\![.]\!] : Env \to A \to \mathbb{R}$, alors [2]

$$\begin{aligned} [\![\mathtt{num}_x]\!]_E &= x \\ [\![e_1 + e_2]\!]_E &= [\![e_1]\!]_E +_{\boldsymbol{R}} [\![e_2]\!]_E \\ [\![var_{id}]\!] &= E(id) \end{aligned}$$

Pour l’affectation, c’est plus compliqué : l’environnement est modifié (en un nouvel environnement [3]). On introduit alors la substitution : Si f est une fonction, alors $f[a \mapsto \chi]$ est la fonction f' telle que $f'(a) = \chi$

1. Ou *inductivement*, c.à.d. *récursivement* !
2. Par convention on note $asae_E$, mais la sémantique $[\![]\!]$ a bien deux arguments : l’arbre de syntaxe abstrait e et l’environnement E.
3. C’est un point subtil et important : si dans un ordinateur la mémoire est physiquement modifiée - par exemple la charge en électrons d’une cellule de DRAM est “pleine” pour un bit 1 et “vide” pour un bit 0, dans la modélisation on *transforme* un état en un *autre* état !

et $f'(y) = f(y) \forall y \neq a$, autrement dit $f' = f[a \to \chi] = \lambda x.\mathbf{if}\ x = a\ \mathbf{then}\ \chi\ \mathbf{else}\ f(x)$; on note parfois $f[a ::= \chi]$ ou $f[a\ \to\ \chi]$. On définit de même inductivement la substitution dans une expression (ou une formule, ou même un AST) de toutes les occurrences libres d'une variable.

1.1.2 Sémantique de l'affectation

L'affectation modifie évidemment l'état du calcul, puisque une variable est affectée. Physiquement l'état de la machine évolue (par exemple la charge -ou les courants, ou le magnétisme-). Formellement, l'environnement est transformé par la sémantique : la sémantique d'une expression dépend de l'environnement de départ, et produit d'une part une valeur, d'autre part un nouvel environnement (parfois le même).

Les effets de bord ("side effects") sont importants [4], car ils motivent l'utilisation d'un ordinateur. D'un certain point de vue, l'affectation est un effet de bord. Mais les effets de bord doivent être disciplinés et abstraits. Par exemple, l'ordre d'évaluation des arguments dans un appel est généralement non spécifié [5] par les langages (et il est souvent erroné d'en dépendre).

Avec l'affectation, la sémantique renvoit *une valeur* ***et*** *un environnement* (souvent le même qu'en entrée) : $[\![.]\!] : Env \to A \to I\!R \times Env$

$$
\begin{array}{rcl}
[\![\mathtt{num}_x]\!]_E & = & (x, E) \\
[\![\mathtt{var}_{id}]\!]_E & = & (E(id), E) \\
[\![e_1{+}e_2]\!]_E & = & (\sigma, E')\ \mathbf{where} \\
 & & \quad [\![e_1]\!]_E\ =\ (v_1, E_1) \\
 & & \quad [\![e_2]\!]_{E_1}\ =\ (v_2, E_2) \\
 & & \quad \sigma\ =\ v_1 +_{\boldsymbol{R}} v_2 \\
 & & \quad E'\ =\ E_2
\end{array}
$$

Notez qu'on a défini l'ordre d'évaluation (gauche puis droite) des arguments de + ; l'affectation est alors :

$$
\begin{array}{rcl}
[\![\mathtt{var}_{id} \leftarrow e]\!]_E & = & (\alpha, E')\ \mathbf{where} \\
 & & [\![e]\!]_E\ = (\alpha, E_1) \\
 & & E'\ = E_1[id \mapsto \alpha]
\end{array}
$$

1.2 sémantique du flôt de contrôle par opérateur de point fixe

Voir précisément le cours d'APS [6] (notamment fin du cours 2, cours 3, cours 4, cours 5).

On rappelle seulement ici quelques idées clefs.

4. Un ordinateur réel qui n'a aucun effet de bord significatif est un coûteux radiateur électrique donc n'a aucun intérêt ! Voir aussi *The Fundamental Physical Limits to Computation* par C.BENETT et R.LANDAUER, (Scientific American, july 1985) etc...

5. Les standards C, C++, Scheme précisent explicitement que l'ordre d'évaluation n'est pas spécifié !

6. Analyse des Programmes et Sémantiques, cours de J.MALENFANT que j'ai enseigné en 2012-2013 en Master 1.

1.2.1 définitions inductives des domaines de Scott

Voir la wikipage des domaines de Scott pour une introduction formelle. Je ne donne ici que quelques intuitions [7]

Considérons encore les arbres de syntaxes abstraits. Notons Op l'ensemble des 4 opérations arithmétique $Op = \{+, -, *, /\}$ et Id l'ensemble infini dénombrable des identifiants (ou noms de variables, ou simplement variables) ; les arbres peuvent être une feuille (nombre ou variable), formellement une union disjointe : $A_0 = \mathbb{R} \uplus Id$ notée aussi $A_0 = \mathbb{R} \oplus Id$; ils peuvent aussi être un nœud à deux feuilles et une opération arithmétique, donc $A'_1 = Op \times A_0 \times A_0$, puis les arbres de profondeur 0 ou 1 sont $A_1 = A_0 \uplus A'_1$, les arbres de profondeur au plus 2 sont $A'_2 = Op \times A_1 \times A_1$, et on peut ainsi (sous certaines conditions bien précises, voir CPO ("complete partial order") etc...) "passer à la limite" et donner un sens à la définition circulaire $A = \mathbb{R} \uplus Id \uplus Op \times A \times A$

D'ailleurs quand on déclare un type somme : programmatiquement, une union étiquettée ("tagged union") ; ou en théorie des types, on a une définition circulaire bien fondée ("well founded") ; voir axiome de régularité, relation bien fondée, etc...

1.2.2 modélisation de la mémoire par un "store"

On modélise une mémoire ("potentiellement infinie") par un *store* ; on se donne un ensemble infini dénombrable (souvent isomorphe aux entiers naturels $\mathbb{N}$) d'adresses $\mathbb{A}$; le store contient une finitude de cases, c.à.d. de valeurs associées à des adresses. Un store à valeurs dans $\mathbb{V}$ est donc une fonction σ à support fini [8] partant de $\mathbb{A}$ arrivant dans $\mathbb{V}$ donc : $\sigma(a) \in \mathbb{V}$. On note Σ le domaine (ou l' "ensemble") des stores.

Comme pour les environnements, une sémantique sur une mémoire doit faire évoluer la mémoire, et la fonction sémantique va donc transformer des stores en d'autres stores. La substitution d'un store sert donc à y modéliser l'affectation ("assignment").

Pour modéliser (par exemple) un *tas de mémoire* dynamique ("heap"), c.à.d. la gestion de la mémoire ("memory management"), il est utile de modéliser un *allocateur d'adresse* (un `malloc`). Il faut donc disposer d'une fonction $FreshAd : \Sigma \to \mathbb{A}$ qui à un store σ associe une adresse $\alpha = FreshAd(\sigma)$ hors du support $\ker \sigma = \{a \in \mathbb{A}, \sigma(a) \neq \bot\}$ de ce store, donc telle que $\alpha \in \mathbb{A} \ \wedge \ \forall a \in \ker \sigma, a \neq \alpha$.

Généralement, la sémantique d'un language présuppose une mémoire potentiellement infinie (donc arbitrairement grande). Pratiquement, le programmeur doit supposer que l'allocateur fonctionne (quitte à s'appuyer sur un ramasse-miettes ("garbage collector" ou glaneur de cellules) : généralement on ne peut plus faire grand chose quand l'allocateur échoue !

Un ramasse-miettes doit généralement connaître toutes les valeurs (notamment celles de variables locales des cadres d'appel). Les ramasse-miettes performants sont copieurs générationnels (ils copient par l'algorithme de Cheney les valeurs vivantes hors d'un "from-space" qui poura être libéré en vrac ; ils insistent sur les cases mémoires les plus fraîchement allouées - la jeune génération - dont la plupart des valeurs sont temporaires donc meurent rapidement). Le ramasse-miettes conservatif de Boehm est utilisable en C ou C++.

7. Il faut se méfier de nos intuitions, elles peuvent être fausses : au XIX$^{\text{ème}}$ siècle, les mathématiciens intuitaient qu'il y autant de nombres réels qu'entiers (ensemble de Cantor, diagonalisation de Cantor, séquence de Cauchy ...), que toutes les fonctions étaient continues, puis que toutes les fonctions continues étaient dérivables... au XX$^{\text{ème}}$, indécidabilité du problème de l'arrêt, théorèmes d'incomplétude de Gödel, ... On peut aussi avoir des intuitions incorrectes en sémantique formelle !

8. L'ensemble de départ, c'est à dire des adresses a telle que $\sigma(a)$ soit défini (donc $\sigma(a) \neq \bot$), est fini donc borné.

1.2.3 combinateur de point fixe

Rappel : un point fixe ("fixpoint") d'une fonction $f : \mathbb{R} \to \mathbb{R}$ (suffisamment "gentille") est un réel x tel que $f(x) = x$ et, dans des hypothèses raisonnables à préciser, en partant d'un réel quasi-arbitraire x_0 et en calculant $x_1 = f(x_0)$ puis $x_2 = f(x_1) = f(f(x_0)) = f^2(x_0)$, puis $x_3 = f(x_2) = f^3(x_0)$, la limite de $x_{n+1} = f(x_n)$ converge vers un point fixe $x = x_\infty$ de f. Quand f est gentillement dérivable, la méthode de Newton-Raphson permet d'en calculer un point fixe souvent rapidement. On calcule très bien la racine carrée (d'un nombre $a > 1$) ainsi :

$$\sqrt{a} = f_\infty(x_0) = \lim_{n\to\infty} f_n(x_0) \quad \text{avec} \quad f(x) = \frac{a}{x} \quad \text{et par exemple} \quad x_0 = \frac{a}{2}$$

On a fait un calcul itératif de point fixe ("fixed-point iteration"). Pour le calcul de la racine carrée, la convergence est très rapide : expérimentalement, dès qu'on a approximé $\sqrt{a}$ avec un chiffre après la virgule, l'étape suivante donnera 2 chiffres de précisions, celle d'après 4 chiffres, etc. En machine, flottant double précision IEEE 754, il faut partir d'un bon x_0 (par exemple stocké dans une table si $1 < a < 4$) et il suffit d'itérer quelques fois (2 à 6 fois) en pratique : la racine carrée est "câblée" dans nos processeurs !

En λ-calcul non typé ("untyped lambda calculus"), il existe un opérateur de point fixe inventé par H.CURRY : le combinateur de point fixe $\mathbf{Y}$ ("Curry's fixed-point combinator"). Il est défini par l'équation

$$\mathbf{Y} = \lambda f.\ (\lambda x.f(xx))\,(\lambda x.f(xx))$$

et vous devez vérifier que si $z = \mathbf{Y}f$ alors $fz = z$

Les points fixes sont liés aux itérations (donc aux récursions) donc l'opérateur $\mathbf{Y}$ formalise les exécutions itératives, c'est à dire les boucles.

Extrait verbatim du slide 38 du cours APS-5 [9]

$$
\begin{array}{l}
execute : Instruction \to Env \to \Sigma \to \Sigma \\
execute[\![\texttt{seq}\ i_1\ i_2]\!]\rho\sigma = execute[\![i_2]\!]\rho(execute[\![i_1]\!]\rho\sigma) \\
execute[\![v\texttt{:=}e]\!]\rho\sigma = (((updateStore\ \sigma)\ (\rho\ v))\ eval[\![e]\!]\rho\sigma) \\
execute[\![\texttt{if}\ be\ i_1\ i_2]\!]\rho\sigma = \textbf{if}\ beval[\![be]\!]\rho\sigma\ \textbf{then}\ execute[\![i_1]\!]\rho\sigma\ \textbf{else}\ execute[\![i_2]\!]\rho\sigma \\
execute[\![\texttt{while}\ be\ i]\!]\rho\sigma = (loop\ \sigma) \\
\qquad \textbf{where}\ loop\ = (\textbf{fix}\ \lambda f.\lambda\sigma.\textbf{if}\ beval[\![be]\!]\rho\sigma\ \textbf{then}\ (f\ execute[\![i]\!]\rho\sigma)\ \textbf{else}\ \sigma)
\end{array}
$$

où **fix** est l'opérateur de point fixe $\mathbf{Y}$ du λ-calcul

2 le flôt de contrôle dans quelques langages

Quelques exemples de flôt de contrôle...

9. par Jacques MALENFANT, dans le cours "Analyse des Programmes et Sémantique" que j'enseigné en 2012-2013.

2.1 sauts conditionnels

On connait tous les boucles etc, donc **goto** et **while** (ou les variantes **repeat**, **for**, ...).
Notez l'amusant Come From :

```
10 COMEFROM 40
20 INPUT "WHAT IS YOUR NAME? "; A$
30 PRINT "HELLO, "; A$
40 REM
```

Sucre syntaxique, par exemple C++11 range-based for loops

```
vector<int> vec;
vec.push_back( 10 ); vec.push_back( 20 );

for (int i : vec )          // range-based for
    cout << i;
```

2.2 les fonctions anonymes et les fonctionnelles

Tout le monde connait Scheme et Ocaml.
Notez que ça existe aussi en C++11

```
extern int a;
auto f =
   [=a](int x) { return x+a; };      // en lambda-calcul:  λ x.x + a
```

Mais C++11 suggère d'expliciter les variables closes [10]. On aurait pu coder `[=]` au lieu de `[=a]`.

2.3 les exceptions

On a déjà vu `setjmp(3)` et `longjmp(3)` en *C*.

En C++, les exceptions :

```
// exceptions
#include <iostream>
using namespace std;
int main () {
  try {
    throw 20;
  } catch (int e) {
    cout << "An exception occurred. Exception Nr. " << e << endl;
  }
  return 0; }
```

Lien subtil entre exceptions et destructeurs (unwinding the stack).
En Java, les exceptions de Java déroulent plus rapidement la pile
Les exceptions d'Ocaml ressemblent à celles de Java, mais le type des exceptions est un type somme ouvert...

10. C++11 permet de clore dans des λ des variables par valeur ou par référence !

2.4 les coroutines

Voir la wikipage sur les coroutines
Voir par exemple Io (sur `iolanguage.org`).

2.5 les générateurs

Voir la wikipage sur les générateurs
En CLU (1975) :

```
string_chars = iter (s: string) yields (char);
  index: int := 1;
  limit: int := string$size (s);
  while index <= limit do
    yield (string$fetch(s, index));
    index := index + 1;
    end;
end string_chars;

for c: char in string_chars(s) do
   ...
end;
```

2.6 Continuation Passing Style

Si on a le temps, on voit la transformation par Continuation Passing Style au tableau (sinon à regarder chez vous).

Prochain projet 2

Etudier des interprètes en logiciel libre existants `sourceforge.net`, `freecode.com`, en choisir un pour y ajouter une primitive et une structure de contrôle à votre choix.

Votre choix détaillé doit être fait pour le 01 décembre 2013 par mél *public* vers `conception-langage-upmc2013@googlegroups.com` et vous devez y expliquer l'interprète choisi (ceux étudiés mais rejetés), la (ou les) primitive(s) à faire, la structure de contrôle à ajouter...

Soyez originaux dans vos choix ! !

C'est à faire totalement en 2013, si possible avant Noël.