DEA Programmation 1996

Sémantique dénotationnelle

Roberto Di Cosmo
École Normale Supérieure
75005 Paris
E-mail: `dicosmo@dmi.ens.fr`

October 11, 1996

Définitions des langages de programmation

Dans la *définition* d'un langage de programmation, on rencontre plusieurs types de difficultés, liées à des aspects assez différents, dont les plus importants sont

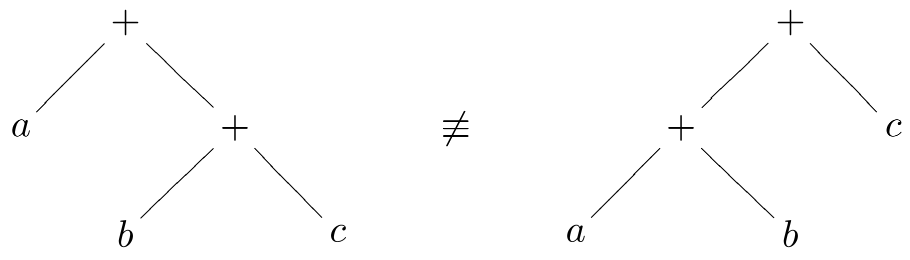
- la *syntaxe*
  l'ortographe et la grammaire des langages de programmation

- la *sémantique*
  ce que un programme fait

- la *pragmatique*
  est-ce ceci le bon langage pour écrire le programme qui m'interesse?

Syntaxe

- **syntaxe**: c'est la définition formelle de ce que c'est un *programme* dans le langage. On doit donner un moyen pour décider si une chaîne de caractères donnée fait partie du langage. On utilise normalement les instruments vus dans les cours des Langages Formels, tels les expressions régulières pour les mots-clefs et les identificateurs, et les grammaires pour les phrases du programme.
  Depuis la définition de Algol et Pascal, on utilise normalement les notations BNF et EBNF.
  On distingue entre
  - **syntaxe concrète**: c'est le programme sous forme de chaîne de caractères, ce qui peut être ambigu
  $$a + b + c =^{?} \begin{cases} a + (b + c) \\ (a + b) + c \end{cases}$$
  - **syntaxe abstraite**: c'est le programme sous forme d'arbre syntaxique, ce qui n'est plus ambigu



Problème: définir la syntaxe de façon que toutes les ambiguïtés syntaxiques soient résolubles simplement par l'analyse *syntaxique*. Ce n'est pas les cas des vecteurs et fonctions FORTRAN, comme on verra plus avant.

Sémantique

- **sémantique**: Une fois bien défini quels sont les programmes syntaxiquement correctes, on doit pouvoir dire de façon précise et univoque ce que chaque programme "fait" à l'exécution (ce qu'il calcule, ou comment il se conduit), y compris si l'exécution donne lieu à des erreurs: cela est indispensable pour
  - l'utilisateur
  - l'implementateur
  - le vérificateur
  - le concepteur du langage

  Une *sémantique* d'un langage de programmation essaye de donner une telle définition précise, à un certain niveau d'abstraction choisi.

  Il existent plusieurs styles différents de sémantique, dont on va rappeler très brièvement les plus connus.

Sémantique opérationnelle

- **sémantique opérationnelle**

Dans l'approche opérationnel, on définit une machine abstraite qui est typiquement un ensemble d'*états* et une *relation de transition* entre les états.
Sur cette machine, un programme avec ses entrées détermine un *état* initial e_0 et une suite de transitions jusqu'à un état final (si le programme termine) qui représente le résultat.
Il y a plusieurs façons, plus ou moins rusées, de présenter la relation de transition de la machine:

- la simple table de transition d'un automate
- la sémantique par *réduction* (ou *small-step semantics* très proche de la réécriture)m qui a sa forme la plus évoluée dans la *Structured Operational Semantics (SOS)* de Plotkin
- la sémantique *naturelle* (ou *big-step semantics*)
- ...

N.B.: La sémantique opérationnelle ne donne pas de signification à un programme tout seul, mais à un programme avec ses données d'entrée.

Sémantique dénotationnelle

- **sémantique dénotationnelle**

  Dans l'approche dénotationnel, on associe à chaque construction du langage de programmation certaines entités mathématiques, tels les entiers, les fonctions, les ensembles et les relations.

  On définit d'abord l'ensemble de ces objets mathématiques, appelés *dénotations*, et ensuite on associe une dénotation à chaque phrase du langage.

  On veut aussi garantir que cette association aie certains propriétés fondamentales:

  – chaque morceaux de programme a une dénotation

  – la dénotation d'une phrase composée dépende seulement de la dénotation de ses composantes (*compositionalité*)

  N.B.: dans cet approche, on donne sémantique aux programmes tous seuls, sans entrées

Sémantique axiomatique

- **sémantique axiomatique** (ou *predicate transformer*)

C'est l'approche qui découle des travaux de Floyd et Hoare sur l'utilisation de la logique pour la spécification et vérification de propriétés de programmes. On s'intéresse à savoir, étant données des assertions logiques valides avant l'exécution d'une instruction, quelles autres assertions vont être valides après (la sémantique d'un programme est ainsi un *transformateurs de prédicats*, d'où la terminologie). Hoare introduit la notation

$$\{P\}Instr\{Q\}$$

pour indiquer que si P est vrai juste avant d'exécuter $Instr$, alors Q est vrai après son exécution.

N.B.: cet approche est normalement utilisé pour la vérification ou la dérivation à partir de specifications des programmes, et il est complémentaire à une sémantique opérationnelle ou dénotationnelle

Ici, on s'intéressera plutôt a sémantique dénotationnelle.

Typage, sémantique statique

Ils existent plein de programmes syntaxiquement correctes qui donnent lieu à l'exécution à des *erreurs*, par exemple

`3/x` si x vaut 0

ou

`3+"a"`

Il serait intéressant de pouvoir détecter ces programmes à la compilation, avant l'exécution et l'erreur qui en suit. C'est en partie pour cela que l'on utilise des *systèmes de type* pour rejeter *avant l'exécution* des programmes erronés (pas tous, et pas seuls, malheureusement).

La phase de typage qui est effectuée à la compilation prend souvent le nom de *sémantique statique* (par exemple, dans la définition de SML [MTH90]).

On va brièvement rappeler ici les notions de base sur le types en programmation.

Qu'est-ce que c'est un type?

Un *type* est essentiellement une *annotation* utilisée pour classifier des objets: on le retrouve dans différents cadres:

logique mathématique introduit par Whitehead et Russel pour réparer la théorie des ensembles de Gottlöb Frege ([Fre79, Fre93, Fre03])

1902 $R = \{x | x \notin x\}$ est un ensemble paradoxale (Betrand Russel)

1910 théorie des types de Principia Mathematica: R n'est pas “bien typé” ([RW10])

informatique théorique la recherche en logique mathématique à été progressivement reprise par l'informatique théorique à partir des travaux fondamentaux de Alonzo Church

[Chu32] lambda calcul non-typé comme fondation des mathématiques basée sur le *fonctions* plutôt que les *ensembles*

[KR35] le terme paradoxale Y oblige a introduire le lambda calcul typé

Maintenant, on s'intéresse aux types *en programmation*

Types et programmes

Une notion très commune aujourd'hui:

- un mécanisme pour classifier les données: strings, int, etc.
- protection contre erreurs fréquents de programmations:
  - on associe un *type* a toute donnée
  - on donne des règles pour décider quelles opérations sont correctes.
    Si $f : A \rightarrow B$, et $o : A$, on peut appliquer f à o et obtenir $fo : B$, comme dans la règle
    $$\frac{f : A \rightarrow B \qquad o : A}{fo : B}$$

  Alors,

  ```
  "a"+3
  ```

  n'est pas *bien typé*.
- documentation minimale

  ```
  mystery: int -> int -> int
  ```
  est sûrement un opérateur binaire sur les entiers
- outils pour chercher des programmes dans les librairies

  La spécification:
  ```
  int -> int -> int
  ```
  permet de retrouver pas mal de fonctions binaire entières

Un ensemble de règles qui définit comment on associe des types aux fragments de programmes et comment vérifier la correction de ces types, est appelé un *système de types*.

Caractéristiques des systèmes de types

Quand on vérifie: la vérification de consistance peut avoir lieu à la compilation (vérification de types *statique*) ou à l'exécution (vérification de types *dynamique*).

Qui écrit les types: les types peuvent soit être inférés par le compilateur (*typage implicite* ou *inférence de type*) ou bien ils doivent être déclarés explicitement par le programmeur (*typage explicite*).

Sûreté: le système de type peut assurer qu'il n'y aura pas d'erreurs de type à l'exécution (*typage fort*) ou seulement éliminer quelque classe réduite d'erreurs (*typage faible*). Un langage qui a un système de types faible est normalement compilé avec du code en plus qui essaye de capturer les erreurs de type à l'exécution, ce qui le ralentit par rapport à un langage avec système de types fort.

Flexibilité: un système de type peut rejeter des programmes correctes, mais trop complexe pour être reconnus comme tels. Les systèmes de type ont différents niveaux de flexibilité, qui vont des systèmes *monomorphes* qui permettent l'utilisation d'un fragment de code seulement avec son type, jusqu'aux très complexes systèmes *polymorphes* qui permettent de réutiliser le même code avec des types différents.

En général, on préfère un système de types statique, fort, flexible et implicite.

Terminologie: sémantique *statique* vs. *dynamique*

Dans la literature (par exemple, dans le Manuel de Définition du langage SML), on trouve souvent employé le terme *sémantique statique* pour parler du système de typage. Dans ce cas, ce que nous appelons *sémantique* et renommé *sémantique dynamique*.

Il est important de remarquer ici que typage et sémantique (ou, si on veut, sémantique statique et sémantique dynamique), ont des objectifs bien différents:

- la sémantique (dynamique) doit donner une définition rigoureuse du comportement de tout programme légal, même si celui-ci aboutit sur un erreur
- un système de type, par contre, a pour but de structurer les données et de sélectionner parmi les programmes syntaxiquement corrects seulement ceux qui ne vont pas produire des erreurs, tout en essayant d'en accepter le plus possible

Il est très dangereux de ne pas faire correctement la distinction entre ce deux aspects: si le système de typage est trop rudimentaire, il peut être trop rigide, et on peut céder à la tentation (comme en C) de le flexibiliser de la mauvaise façon en bricolant la sémantique (par exemple, en rajoutant de pointeurs non sûrs). On n'intervient pas au bon niveau, et le résultat est un langage dont le système de types devient inutile.

Une histoire (incomplète) des *vrais* langages de programmation

- FORTRAN
- PASCAL
- ADA
- C
- C++

FORTRAN

"FORmula TRANslation language": moitié des années 50

- syntaxe:
  - même notation pour vecteurs et fonctions:

    `f(100)`

    peut tout aussi bien être la valeur du vecteur `f` à la position `100` ou la valeur de la fonction `f` sur l'entier `100`
  - comme les espaces ne sont pas significatifs, les cycles `do` ne sont pas reconnaissables par un parseur avec un look-ahead fini:

    `do 100 identificateur=0,100`

    Il faut attendre la `,` pour savoir si c'est une boucle `do` ou bien une assignation à l'identificateur `do100identificaeur`
- sémantique: l'ambiguïté de la syntaxe sur vecteurs et fonctions et la compilation séparée empêchent la compositionalité, (en compilant un module, il y a des cas où on ne sait pas si on doit compiler une fonction ou un vecteur) et en l'absence d'une définition précise, chaque compilateur fait ce qu'il peut.

  En plus, des constructions comme `COMMON` et `EQUIVALENCE`, motivées par l'efficacité, compliquent grandement la tâche de donner une sémantique formelle du langage et donnent des erreurs surprenants.
  Le passage des paramètres par adresse donne des résultats imprévus.

FORTRAN: sémantique statique

MINUS:

- trés faible système de types: real, integer et array (pas de strings!)
- puissants moyens pour casser les système de types: `COMMON` et `EQUIVALENCE`[1]

```
C       fortran program for the real fortran programmer (courtesy of Dr. M. Stiavelli)
        program bleah
        real myvalues(100)
        common/bleah1/myvalues
        call joke(3.22)
        write(*,*) 'let''s write 3.22' 3.22
        write(*,*) (myvalues(i),i=1,100,10)
        stop
        end

        subroutine joke(value)
        common/bleah1/myvalues(100)
        do i=1,100
                myvalues(i) = value
        enddo
        value = 3.755
        return
        end
```

PLUS:

- forte motivation pour chercher des meilleurs systèmes de types

[1] Cela donne des bugs surpenants: programmes qui marchent sur une machine ne marchent pas sur une autre (typiquement avec des types real ou integer de longueur différente.

PASCAL

Langage didactique introduit par Wirth ([JW72]) en 1971-73

PLUS:

- quatre types de base (*reals*, *integers*, *characters* et *booleans*),
- moyens pour définir des types utilisateur (même récursifs): vecteur (de longueur fixe), enregistrements et pointeurs *sûrs*
- typage fort: chaque identificateur doit être utilisé avec son type déclaré

MINUS:

- chaque identificateur doit être utilisé avec son type déclaré, signifie aussi:
  programmes lourdes à écrire (déclarations de type explicites)
- système de type *monomorphe* empêche la réutilisation du code

Une procédure de tri en PASCAL

- l'*algorithme* de tri ne dépend pas du type des objets à trier
- mais, en Pascal le type de chaque identificateur est figé

```
procedure sort(function comp(x: t, y: t): boolean,
               v: array [1..300] of t)
```

Voilà: on doit écrire une copie de ce code pour trier des entiers, une autre pour des réels etc., même si la seule différence est le type déclaré pour les paramètres formels de la procédure.[2]

Cela étant inacceptable, il y a deux choix:

- chercher un système de types plus flexible (ADA)
- rajouter dans le langage des moyen de casser le système de type (C)

[2]En réalité, c'est pire que cela: en Pascal le vecteurs sont statiques, et ce 300 est obligatoire dans la déclaration, donc il faut aussi une version différente pour trier 1000 entiers!

ADA

Défini sous contrat de la DOD: premier standard en 1979

PLUS:

- ADA ([Ame83]) introduit la notion de composante `generic`: un module peut être *parametré* par rapport à une quelque structure de données [3].
- on peut écrire une procédure de tri `sort` parametrée par rapport au type `t` des objets du vecteur à trier:

```
generic
   type typ is private
   with function comp(x: typ, y:typ)
procedure sort(a: array (1..MAX) of typ)
```

Finalement, un tri sur les entiers n'est qu'une instance du programme générique où `typ` est `int`:

```
procedure int_sort is new sort(int,"<")
```

N.B. ce mécanisme produit en effet une nouvelle copie du code à chaque instance, mais ce n'est plus le programmeur à faire ça.

MINUS:

- les objets génériques ne sont pas de première classe:
  (on peut le nommer, mais pas le calculer ou passer comme paramètres)

[3] Cela vient plutôt de la tradition des ADT, mais il est aussi un instrument pour introduire du polymorphisme.

Histoire de (l'horreur de) C

Dans C ([KR78]),

- pas de pointeurs *sûrs*:
  - `&` rend la position en *mémoire* de tout type de donnée
  - `*` rend le contenu de tout adresse *mémoire*
  - tout valeur de type *integer* peut être utilisée comme pointeur
- le *type cast*: un façon de mentir légalement sur les types

```
float *a;
char *b;

b = (char *) a;
```

  compile sans même un avertissement!
- on ne vérifie pas les paramètres des fonctions (même pas le numero)
- ...

CELA DONNE UNE LIBERTÉ TOTALE DE:

- casser le système de types
- casser la portée des identificateurs
- casser votre programme

Le tri en C

Il est possible d'écrire une fonction polymorphe de tri: voilà comment est défini *quicksort* dans la librairie standard Unix

```
void *qsort(void *base, int nmemb, int size, int (*compar)());
```

Paramètres:

- base: un pointeur à l'adresse de base des données en mémoire
- nmemb: le numéro d'éléments à trier
- size: la taille en *bytes* d'un élément
- compar: le nom de la fonction de comparaison, à appeler avec deux pointeurs *non-sûrs* aux éléments à comparer

Comment (mal)utiliser qsort

Voici du code C *parfaitement légal*, pris dans une page de manuel Unix:

```
static    int charcompare(i,j)
char *i, *j;
{
     return(*i - *j);
}

main()
{
     char a[10];
     int i;
     a[0] = '9';  a[1] = '8';  a[2] = '7';

     qsort(a,3,sizeof(char),charcompare);
     for (i=0; i<3; i++) printf(" %c",a[i]);
     printf("\n");
}
```

- on déclare `compar` comme fonction sans arguments
- on passe à `compar` deux arguments: les adresses mémoire des objets à comparer
- on soustrait (le code de) deux caractères dans `charcompare`!

Tout ça est non seulement *légal*, mais même recommandé !

Sortir des identificateurs C de leur portée

Ce code C *parfaitement légal* fait sortir la valeur de l'identificateur `v` de sa portée:

```
#include "stdio.h"

char *breakscope()
{
char v;
        v = 's';
return(&v);
}

main()
{
     printf("Broken scope, value is %c\n",*breakscope());
     exit(0);
}
```

- dans `breakscope`, on prend l'adresse de l'identificateur locale `v`
- on le rend comme résultat à main
- le programme marche, et imprime `Broken scope, value is s`

C'est *légal* !

Ce système de type trop rigide, affaibli par les pointeurs non sûrs, est tellement faible et il a causé tellement de problèmes, que dans les langages successifs on voit comment beaucoup plus d'attention est réservée au typage statique, même s'il s'agit de langages basés sur C, comme . . .

C++ [Str86]

PLUS:

- vérifie le numéro des paramètres d'une fonction:
  notre paramètre formel C `compar` n'est pas un paramètre formel C++
- vérifie le type des paramètres des fonctions

MINUS:

- type cast encore là: on peut toujours mentir
- pointeurs non sûrs encore là
- on peut bricoler le `qsort` C et le faire passer sous C++

MINUS: (pour les experts)

- mélange horrible entre object-oriented et abstract data types

L'ordre d'évaluation

Erreurs communs dans la définition de la sémantique informelle
Dans pas mal de définitions informelles (ou pas complètement formelles) de la sémantique des langages de programmation, on trouve des *ambiguïtés* dans la définition, souvent motivées par le souci de permettre à des éventuels compilateurs des optimisations sur le code.

Ces ambiguïtés sont, à juste titre, considérées comme *erreurs* dans une sémantique formelle (sauf dans le cas ou le nondeterminisme fait partie du langage que l'on spécifie, mais il ne s'agit pas de ce qui se passe en Algol ou en C), et cela en raison du fait que la signification d'un programme n'est plus bien définie.

Pour ne citer que deux exemples, en Algol le résultat du programme suivant (exemple de C. Gunter [Gun92])

```
begin integer a;
integer procedure f(x,y); value y,x; integer y,x;
          a := f := x+1;
integer procedure g(x); integer x;
          x := g := a+2;
a := 0; outreal(1, a + (f(g(a),a)/g(a))) end
```

puisque `f` et `g` font de effets de bord sur une même variable, dépend fortement de l'ordre d'évaluation de l'expression, que, lui, n'est pas défini dans le Algol report. (En Pascal, on remédie à ça en spécifiant l'ordre habituel de gauche à droite).

En C, l'expression

```
f(i++,i--)
```

est de même mal définie, vu que l'ordre d'évaluation des arguments d'une fonction n'est pas spécifié dans les manuel de C.

References

[Ame83] American national standard institute. *The programming language ADA: reference manual*, 1983. ANSI-MIL-STD-1815A-1983.

[Chu32] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2):346–366, 1932.

[Fre79] Gottlob Frege. Begriffsschrift, eine der arithmetischen nachgebildete formelsprache des reinen für das reinene dansken, 1879. Reprinted in [vH67], pagg. 1–82.

[Fre93] Gottlob Frege. *Grundgesetze der Arithmetik, begriffsschriftlich abgeleitet*, volume 1. Pohle, Jena, 1893.

[Fre03] Gottlob Frege. *Grundgesetze der Arithmetik, begriffsschriftlich abgeleitet*, volume 2. Pohle, Jena, 1903.

[Gun92] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques.* Foundations of Computing. MIT Press, 1992.

[JW72] Kathleen Jensen and Niklaus Wirth. *PASCAL user manual and report : ISO PASCAL standard.* Springer Verlag, 1972.

[KR35] S. C. Kleene and J.B. Rosser. The inconsistency of certain formal theories. *Annals of Mathematics*, 36(2):630–636, 1935.

[KR78] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language.* Prentice-Hall software series. Prentice Hall, 1978.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* The MIT Press, 1990.

[RW10] Bertrand Russel and Alfred North Whitehead. *Principia Mathematica.* Cambridge University Press, 1910. Cambridge, England.

[Str86] Bjarne Stroustrup. *The C++ programming language.* Addison-Wesley series in Computer Science. Addison-Wesley, 1986.

[vH67] Jean van Heijenoort, editor. *From Frege to Gödel, a source book in mathematical logic.* Harvard University Press, 1967.

Index