Calculabilité, réécriture, lambda-calcul, automates cellulaires ... comment s'abstraire d'un langage de programmation particulier ?

Gilles Dowek

On sait définir

La notion de fonction calculable

Ensemble des fonctions calculables : plus petit sous-ensemble de $\bigcup_n(\mathbb{N}^n \to \mathbb{N})$ qui contient

- $x_1, ..., x_n \mapsto x_i$
- $x_1, ..., x_n \mapsto 0$
- $x \mapsto x + 1$

et clos par

- composition
- définition par récurrence
- minimisation

On sait définir

La notion de programme

Texte : suite de symboles pris dans un alphabet fini
Ou alors arbre étiqueté dans un alphabet fini

Auquel on associe une sémantique qui est une fonction calculable

L'égalité

Égalité de deux fonctions : extensionnelle
La fonction « tri sélection » et la fonction « tri fusion » sont égales : une seule fonction « tri »

Égalité de deux programmes : lettre à lettre
`fun x -> x + 1` et `fun y -> y + 1` différents
Possibilité de relaxer un peu la condition (mais pas énormément)

Mais alors ...

Qu'est-ce que l'algorithme « tri fusion » ?

Ni une fonction, ni un programme

I. La sémantique opérationnelle à petits pas

Diverses manières de définir la sémantique du programme 3 * 7 + 2 * 6

Sémantique dénotationnelle : récurrence sur la structure du programme

$[\![3]\!] = 3$

$[\![7]\!] = 7$

$[\![3 * 7]\!] = [\![3]\!] * [\![7]\!] = 21$

...

$[\![3 * 7 + 2 * 6]\!] = 33$

Diverses manières de définir la sémantique du programme 3 * 7 + 2 * 6

Sémantique opérationnelle à petits pas (Plotkin, 81)

Un calcul est un processus temporel : une suite de petits pas

$3 * 7 + 2 * 6 \longrightarrow 21 + 2 * 6 \longrightarrow 21 + 12 \longrightarrow 33$

Le programme `avance(10);tourne(90);`

Le programme ne renvoie pas de valeur, mais il transforme la position de la tortue / du lutin

Sémantique dénotationnelle
$[\![\texttt{avance(10); tourne(90);}]\!]$
$= (\langle n, p, \theta\rangle \mapsto \langle n + 10 * \cos\theta, p + 10 * \sin\theta, \theta + 90\rangle$
composition de $\langle n, p, \theta\rangle \mapsto \langle n + 10 * \cos\theta, p + 10 * \sin\theta, \theta\rangle$
et $\langle n, p, \theta\rangle \mapsto \langle n, p, \theta + 90\rangle$

Sémantique opérationnelle à petits pas
$\langle \texttt{avance(10); tourne(90);}, \langle n, p, \theta\rangle\rangle$
$\longrightarrow \langle \texttt{tourne(90);}, \langle n + 10 * \cos\theta, p + 10 * \sin\theta, \theta\rangle\rangle$
$\longrightarrow \langle \varnothing, \langle n + 10 * \cos\theta, p + 10 * \sin\theta, \theta + 90\rangle\rangle$

Le programme y = x + 3; x = y + 4;

Comme dans le cas du lutin : le programme transforme quelque chose (ici invisible)
Sémantique dénotationnelle
$[\![\mathrm{y} = \mathrm{x} + 3; \mathrm{x} = \mathrm{y} + 4;]\!] = (\langle \mathrm{x} = n, \mathrm{y} = p \rangle \mapsto \langle \mathrm{x} = n{+}7, \mathrm{y} = n{+}3 \rangle)$
composition de ...

Sémantique opérationnelle à petits pas
$\langle \mathrm{y} = \mathrm{x} + 3; \mathrm{x} = \mathrm{y} + 4;, \langle \mathrm{x} = n, \mathrm{y} = p \rangle \rangle$
$\longrightarrow \langle \mathrm{x} = \mathrm{y} + 4;, \langle \mathrm{x} = n, \mathrm{y} = n + 3 \rangle \rangle$
$\longrightarrow \langle \varnothing, \langle \mathrm{x} = n + 7, \mathrm{y} = n + 3 \rangle \rangle$

À chaque petit pas

Un programme se transforme (possiblement) lui-même

Il transforme (en général) aussi autre chose : l'état

Dans beaucoup de cas une dichotomie état / programme

Langages à lutin (et robotiques) :

- position du lutin (ou du robot)
- programme

Langages impératifs :

- état mémoire
- programme

Machines de Turing :

- configuration de la bande + position de la tête + état
- table de transition

Automates cellulaires :

- état des cellules
- règles de transition

La réécriture également

$$0 + y \longrightarrow y$$

$$S(x) + y \longrightarrow S(x + y)$$

$$S(S(0)) + S(S(0)) \longrightarrow S(S(0) + S(S(0))) \longrightarrow$$
$$S(S(0 + S(S(0)))) \longrightarrow S(S(S(S(0))))$$

termes

contiennent 0, S

mais aussi $+$

mais pas $0 + y \longrightarrow y$

règles

La trace d'exécution d'un programme

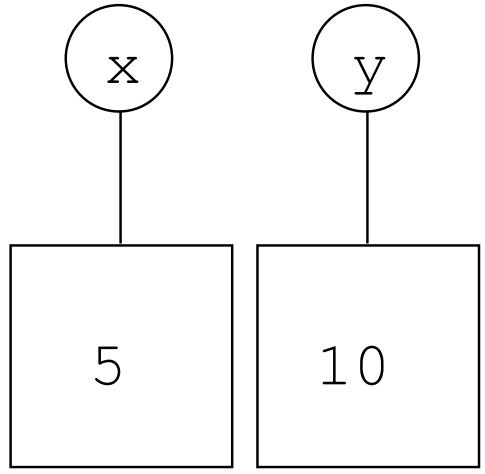
$\langle \mathtt{p}, e \rangle \longrightarrow \langle \mathtt{p}_1, e_1 \rangle \longrightarrow \langle \mathtt{p}_2, e_2 \rangle \longrightarrow \langle \mathtt{p}_3, e_3 \rangle \longrightarrow \ldots$

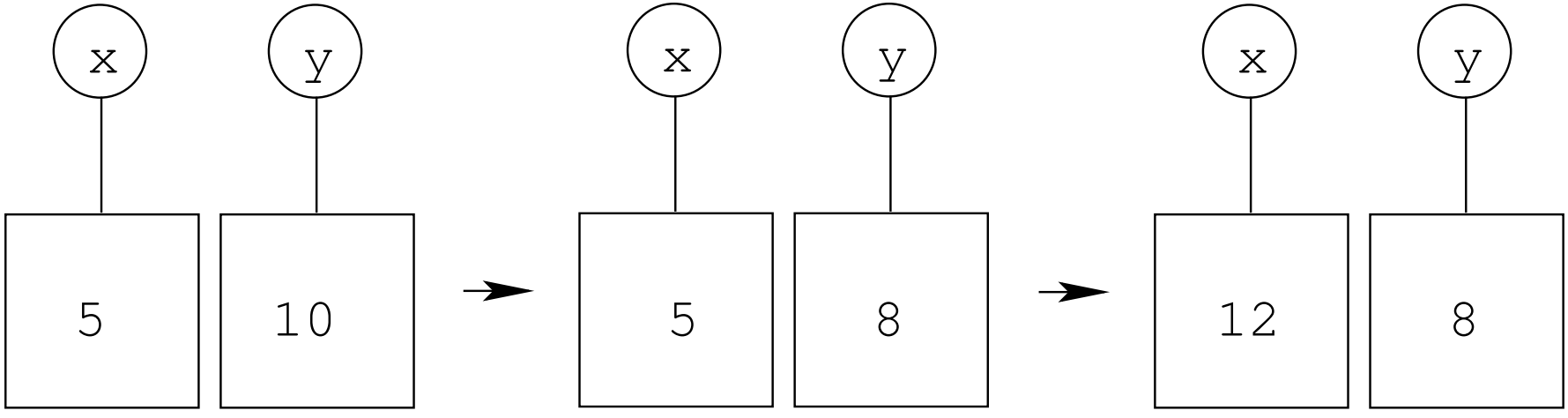
Trace d'exécution de $\mathtt{p}$ dans e : $e \longrightarrow e_1 \longrightarrow e_2 \longrightarrow e_3 \longrightarrow \ldots$

Dans le cas non déterministe : un arbre à la place d'une suite

Un exemple de trace d'exécution avec des boîtes

$y = x + 3; x = y + 4;$ sur





II. Qu'est-ce qu'un algorithme ?

Deux programmes définissent le même algorithme

si pour tout e, $\mathtt{p}$ et $\mathtt{p}'$ définissent la même trace d'exécution sur e
(Gurevich, 00)

Une définition de la notion d'algorithme

Une fonction effective qui associe une suite d'états à chaque état

Indépendant de la notion de programme (mais un programme définit un algorithme (sémantique opérationnelle à petits pas))

Si on garde uniquement le dernier état de la suite : la fonction calculée par l'algorithme

Trois regards sur un programme

Ce qu'il est : une suite de symboles (un texte)

Ce qu'il fait : une trace en fonction de l'état initial (algorithme)

Ce qu'il calcule : le dernier élément de la trace (fonction)

III. Séparer les programmes des états ?

Mais abandonner la notion d'état : un objectif en soi

Langages fonctionnels
Un programme ne fait pas quelque chose il calcule quelque chose

`(fun x -> fun y -> x * x + y) 7 1` $\longrightarrow$ `(fun y -> 7 * 7 + y) 1` $\longrightarrow$ `7 * 7 + 1` $\longrightarrow$ `49 + 1` $\longrightarrow$ `50`
est bien une suite de petits pas

Mais où est le programme et où est l'état ?

`(fun x -> fun y -> x * x + y) 7 1` $\longrightarrow$ `(fun y -> 7 * 7 + y) 1` $\longrightarrow$ `7 * 7 + 1` $\longrightarrow$ `49 + 1` $\longrightarrow$ `50`

la trace d'exécution est

$\varnothing \longrightarrow \varnothing \longrightarrow \varnothing \longrightarrow \varnothing \longrightarrow \varnothing$

même complexité $\Rightarrow$ même algorithme

`(fun x -> fun y -> x * x + y) 7 1` $\longrightarrow$ `(fun y -> 7 * 7 + y) 1` $\longrightarrow$ `7 * 7 + 1` $\longrightarrow$ `49 + 1` $\longrightarrow$ `50`

la trace d'exécution est

`(fun x -> fun y -> x * x + y) 7 1` $\longrightarrow$ `(fun y -> 7 * 7 + y) 1` $\longrightarrow$ `7 * 7 + 1` $\longrightarrow$ `49 + 1` $\longrightarrow$ `50`

même algorithme $\Rightarrow$ même programme

L'Empire contre-attaque ? (Gurevich-Moschovakis)

Toutefois : une possibilité

```
f = fun x -> fun y -> x * x + y
g = fun y -> 7 * 7 + y
```

`f 7` $\longrightarrow$ `g`

`g 1` $\longrightarrow$ `7 * 7 + 1`

`f 7 1` $\longrightarrow$ `g 1` $\longrightarrow$ `7 * 7 + 1` $\longrightarrow$ `49 + 1` $\longrightarrow$ `50`

Pas encore super : `fun y -> 7 * 7 + y` apparu au cours de la réduction : comment savoir qu'il faut un symbole `g` ?

Mais d'ailleurs : quand compile-t-on `fun y -> 7 * 7 + y` ?

Jamais

Une meilleure solution

```
f = fun x -> fun y -> x * x + y
```

f x y $\longrightarrow$ x * x + y

f 7 1 $\longrightarrow$ 7 * 7 + 1 $\longrightarrow$ 49 + 1 $\longrightarrow$ 50

Ou alors ... des fermetures

Finalement

Avec un programme fonctionnel, comme dans le cas de la réécriture

Aussi possible de définir des traces d'exécution

Et donc une équivalence entre programmes qui font la même chose

IV. Une fonction effective qui associe une suite d'états à chaque état

Deux problèmes

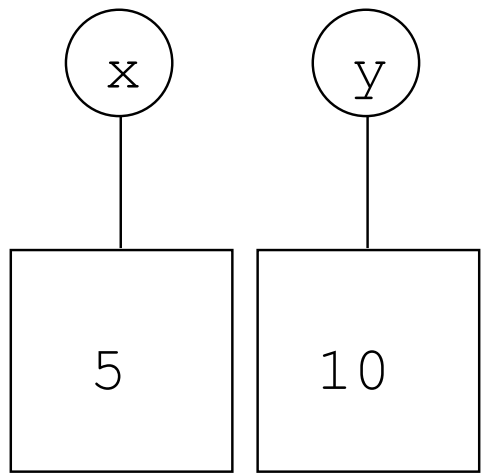
(1) Ensembles des états trop baroques : position du robot, état mémoire, bande de la machine de Turing... quoi d'autre ?

(2) Si la fonction qui à chaque état associe une trace est quelconque, rien ne garantit que la fonction calculée soit calculable

Manque petit pas : n'utilise qu'une quantité finie d'information sur la structure

Les états comme structures algébriques

Ensembles (ou familles d'ensembles) munis d'opérations
Exemple



$\mathbb{Z}$ muni de deux opérations zéro-aires $\mathtt{x}$ et $\mathtt{y}$
Exemple bande d'une machine de Turing : fonction de $\mathbb{Z}$ dans Σ

Les transitions

modifient les opérations (mais pas le domaine, ni le nombre, ni le nom des opérations)

Transitions finitaires : ensemble fini de termes : si deux structures coïncident sur ces termes, elles évoluent « de la même manière »

Ces deux conditions définissent

Des « machines à états abstraits » (Gurevich, 00) ou « structures dynamiques »

Les structures dynamiques sont à l'informatique ce que les structures sont à l'algèbre

Le langage algébrique généralise à la fois les machines de Turing, les machines de von Neumann et beaucoup d'autres modèles de calculs ...

Évite leur vocabulaire un peu baroque : bande, tête de lecture, état interne ... case mémoire ...

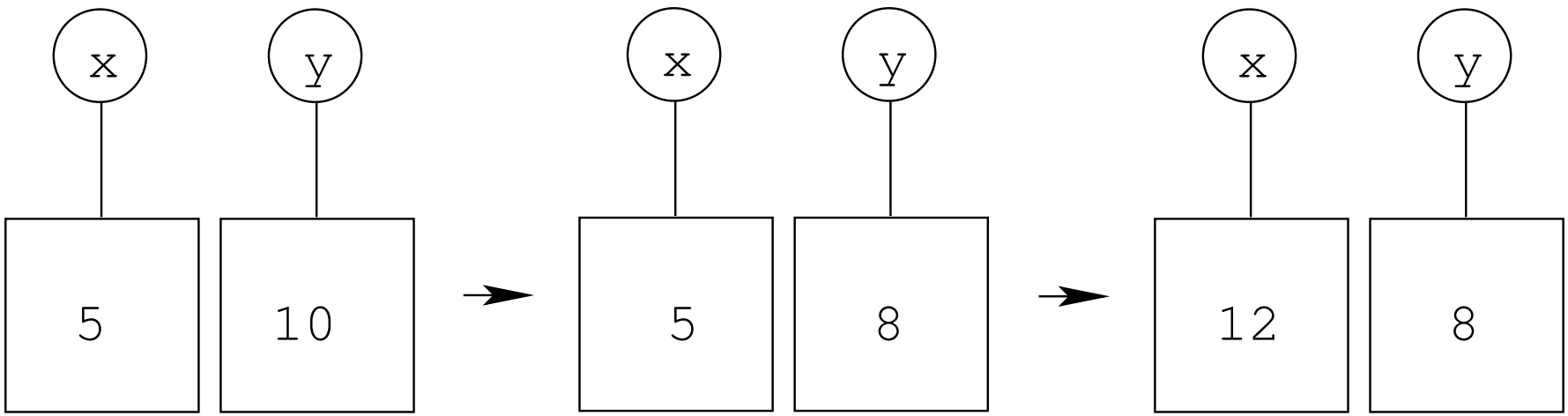
VI. Que retenir (et qu'enseigner) ?

Comprendre (la sémantique d') un langage

C'est comprendre comment un programme s'exécute petit pas par petit pas

Dans le cas des langages impératifs (Python)

Un programme fait quelque chose à un état



Comprendre un algorithme

C'est comprendre quelles traces d'exécution il engendre
(indépendamment d'un langage de programmation particulier)

Tri à bulle :
$[7, 9, 4, 13] \mapsto ([7, 9, 4, 13] \longrightarrow [7, 4, 9, 13] \longrightarrow [4, 7, 9, 13])$

Entre l'égalité des programme et l'égalité des fonctions

L'égalité des algorithmes : faire la même chose = mêmes traces