Structures de contrôle, troisième cours

Chassez le contrôle … : la programmation déclarative

Xavier Leroy

2024-02-08

Collège de France, chaire de sciences du logiciel
`xavier.leroy@college-de-france.fr`

La programmation déclarative

Un mouvement qui apparaît dans les années 1960 (LISP, APL) et prend de l'ampleur dès les années 1970 (Prolog) et 1980 (langages purement fonctionnels, langages réactifs).

Idée : les programmes doivent décrire ce qu'il faut calculer bien plus que comment mener le calcul.

D'où des langages qui exposent beaucoup moins les programmeurs à la mémoire (état mutable) et au contrôle (enchaînement des calculs).

L'espoir est de facilite non seulement la programmation, mais aussi l'exécution en parallèle.

Les feuilles de calcul : expressions avec partage

Un langage d'expressions avec partage

Des expressions arithmétiques avec des variables :

Expressions :

$$
\begin{array}{lll}
e ::= & 0 \mid 1.2 \mid 3.1415 \mid \dots & \text{constantes} \\
& \mid x \mid y \mid z \mid \dots & \text{variables} \\
& \mid f(e_1, \dots, e_n) & \text{opérations } (+, -, \times, /, \Sigma, \text{etc})
\end{array}
$$

Des ensembles d'équations *variable* $=$ *expression* :

Programmes :

$$p ::= \{x_1 = e_1; \dots ; x_n = e_n\}$$

Les feuilles de calcul (*spreadsheets*)

	A	B	C
1	Date	Income	Expenses
2	2005-12-17	235 €	128
3	2005-12-18	311 €	124
4	2005-12-19	457 €	466
5	2005-12-20	232 €	132
6	2005-12-21	122 €	134
7	2005-12-22	128 €	223
8	2005-12-23	432 €	218
9	2005-12-24	256 €	121
10		2.173 €	1.546
11			
12	Avg. Profit	=AVERAGE(D2:D9)	

ComputerHope.com

Une visualisation d'un ensemble d'équations :

$$B2 = 235 \quad C2 = 128 \quad \dots \quad B9 = 256 \quad C9 = 121$$

$$D2 = B2 - C2 \quad \dots \quad D9 = B9 - C9$$

$$B10 = SUM(B2, \dots, B9) \qquad C10 = SUM(C2, \dots, C9) \qquad B12 = AVG(D2, \dots, D9)$$

La condition d'acyclicité

Pour garantir qu'on peut toujours évaluer les programmes, on interdit les équations $x = e$ où e dépend directement ou indirectement de x :

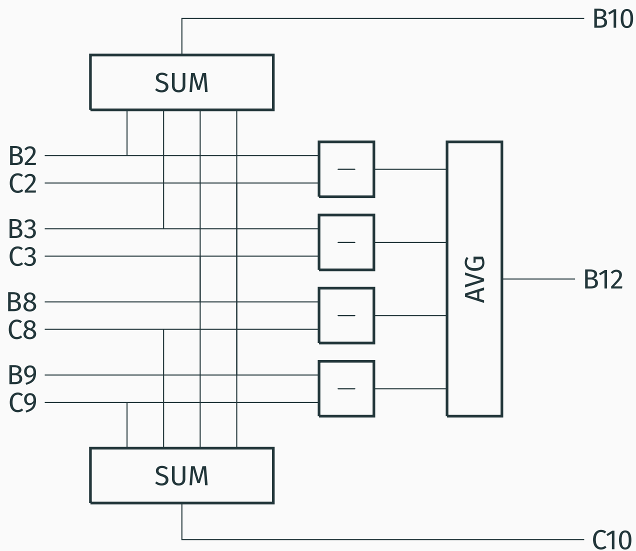
$$\{x = x^2 - 1\} \text{ ✘} \qquad \{x = y + 1;\ y = x - 1\} \text{ ✘}$$

Cette condition est vraie si et seulement si on peut écrire le programme sous forme d'une liste ordonnée

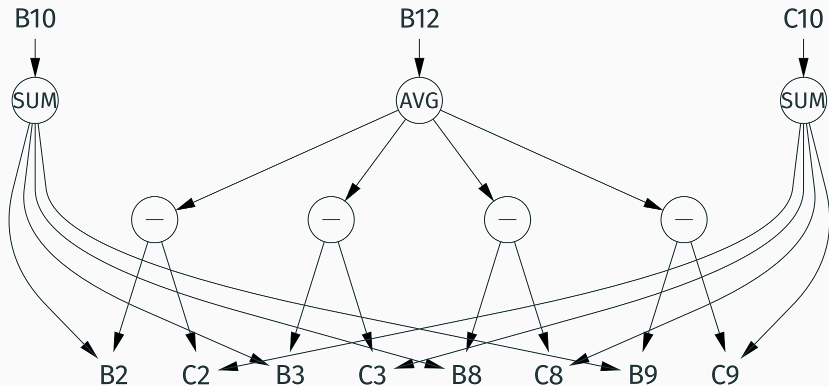
$$x_1 = e_1; \ldots; x_n = e_n$$

où e_i ne mentionne que les variables x_j avec $j < i$.

Autre présentation : les circuits combinatoires



Autre présentation : le DAG de dépendances



Règles d'évaluation

On réduit le programme en appliquant les règles suivantes n'importe où dans les membres droits d'équations :

$$x \to e \qquad \text{si } x = e \text{ est une équation}$$

$$f(v_1, \ldots, v_n) \to v \qquad \text{si } v = f^*(v_1, \ldots, v_n)$$

(v dénote une valeur numérique et f^* la sémantique de l'opérateur f, p.ex $+^*$ est l'addition en virgule flottante.)

On s'arrête lorsque le programme est en forme normale $x_1 = v_1; \ldots x_n = v_n$.

Exemple :

$$\{x = 1; y = x + x\} \to \{x = 1; y = 1 + x\}$$
$$\to \{x = 1; y = 1 + 1\} \to \{x = 1; y = 2\}$$

Séquences de réductions

On peut appliquer les règles de réduction dans des ordres différents :

$$\{x = 1 + 1; y = x + 2\} \nearrow \{x = 2; y = x + 2\} \longrightarrow \cdots$$

$$\{x = 1 + 1; y = x + 2\} \searrow \{x = 1 + 1; y = (1 + 1) + 2\} \longrightarrow \cdots$$

Toutes les séquences de réductions terminent sur la même forme normale (propriété de confluence).

Stratégies d'évaluation

Certaines séquences de réductions coûtent plus que d'autres!

Exemple : $\{x_0 = 1; x_1 = x_0 + x_0; \ldots; x_n = x_{n-1} + x_{n-1}\}$.

Stratégie «appel par nom» : on substitue avant d'évaluer

$$\{x_0 = 1; x_1 = 1 + 1; x_2 = (1 + 1) + (1 + 1); x_3 = x_2 + x_2; \ldots\}$$

États intermédiaires de taille $\mathcal{O}(2^n)$.

Stratégie «appel par valeur» : on évalue avant de substituer

$$\{x_0 = 1; x_1 = 2; x_2 = 4; x_3 = x_2 + x_2; \ldots\}$$

États intermédiaires de taille $\mathcal{O}(n)$.

Une stratégie optimale

Substituer des variables uniquement par des valeurs :

$$x \to v \qquad \text{si } x = v \text{ est une équation}$$

Si le programme est représenté par une liste ordonnée selon les dépendances, cette stratégie s'implémente par une fonction d'évaluation :

$$\begin{aligned} \texttt{eval}(\varepsilon) &= \varepsilon \\ \texttt{eval}(x = e; p) &= (x = v; \texttt{eval}(p[x \leftarrow v])) \quad \text{où } \; v = \texttt{eval}(e) \end{aligned}$$

Ajout d'opérateurs non stricts

Une expression conditionnelle n'a pas besoin d'évaluer tous ses arguments : elle est ≪non stricte≫.

$$\texttt{if0}(0, e, e') \to e$$
$$\texttt{if0}(v, e, e') \to e' \quad \text{si } v \neq 0$$

La stratégie ≪par valeur≫ peut alors faire des calculs inutiles.

Exemple :

$$\{x = e;\ \ y = E;\ \ z = \texttt{if0}(x - x, x, y)\}$$

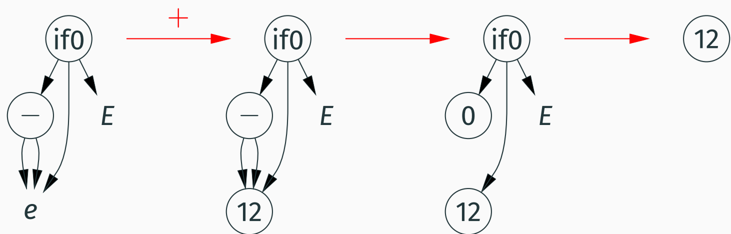
Pas besoin d'évaluer la grosse expression E pour obtenir z.

$\to$ Passage à une stratégie ≪par nécessité≫ / ≪paresseuse≫.

La stratégie paresseuse

Une forme d'appel par nom avec mémoisation : si $x = e$,
e est évaluée la première fois où la valeur de x est requise,
et sa valeur est réutilisée les fois suivantes.

S'exprime facilement sur le DAG de dépendances, par réécriture de graphe, en remplaçant les sommets calculés par leur valeur.



Programmation réactive

Calculer avec des flux de valeurs

Un flux (*stream*) : une suite de valeurs $v(0), v(1), \ldots, v(t), \ldots$ indexée par un temps discret t.

Les opérations arithmétiques s'étendent «point par point» aux flux, p.ex.

	$t=0$	$t=1$	$t=2$	$t=3$	$t=4$	$t=5$
x	x_0	x_1	x_2	x_3	x_4	x_5
1	1	1	1	1	1	1
$x+1$	x_0+1	x_1+1	x_2+1	x_3+1	x_4+1	x_5+1

Opérateurs temporels

Permettent de consulter la valeur précédente d'un flux.

v `fby` e (lire : *followed by*)

vaut la constante v au temps 0 et $e(t)$ au temps $t+1$;

$\approx$ le constructeur `cons` des listes infinies.

Exemple :

	$t=0$	$t=1$	$t=2$	$t=3$	$t=4$	$t=5$
x	1	2	3	4	5	6
`0 fby` x	0	1	2	3	4	5
`0 fby 0 fby` x	0	0	1	2	3	4

Opérateurs temporels

Opérateurs dérivés :

`pre`(e) $\equiv$ `None fby` e

vaut $e(t)$ au temps $t+1$; indéfini au temps 0.

e_1 `->` e_2 $\equiv$ `if (true fby false) then` e_1 `else pre`(e_2)

vaut $e_1(0)$ au temps 0 et $e_2(t)$ au temps $t+1$.

Exemple :

	$t=0$	$t=1$	$t=2$	$t=3$	$t=4$	$t=5$
x	0	-1	-2	-3	-4	-5
y	1	2	3	4	5	6
`pre`(y)	`None`	1	2	3	4	5
`true fby false`	`true`	`false`	`false`	`false`	`false`	`false`
x `->` y	0	1	2	3	4	5

Équations entre flux

(Lustre (P. Caspi, N. Halbwachs, 1985), Scade, Simulink.)

Un programme réactif est un ensemble d'équations entre flux $\{x_1 = e_1; \ldots; x_n = e_n\}$.

Exemple : $\{x = 0 \texttt{ fby } 1 + x\}$

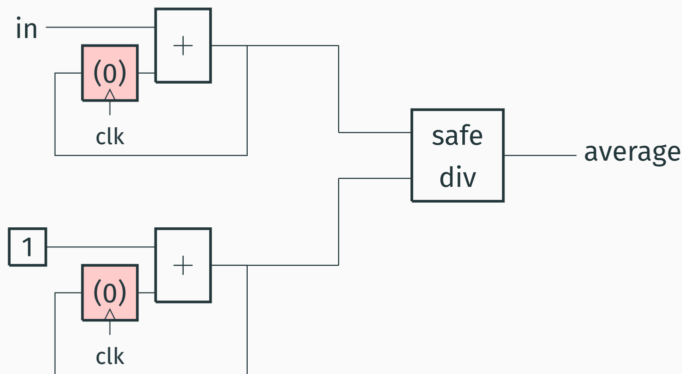
	$t=0$	$t=1$	$t=2$	$t=3$	$t=4$	$t=5$
x	0	1	2	3	4	5
$1+x$	1	2	3	4	5	6
$0 \texttt{ fby } 1+x$	0	1	2	3	4	5

Exemple : $\{sum = 0 \texttt{ fby } in + sum\}$ où *in* est un flux d'entrée.

	$t=0$	$t=1$	$t=2$	$t=3$	$t=4$
in	i_0	i_1	i_2	i_3	i_4
sum	0	i_0	i_0+i_1	$i_0+i_1+i_2$	$i_0+i_1+i_2+i_3$

Lien avec les circuits séquentiels synchrones

L'opérateur `fby` joue le rôle d'une mémoire (bascule synchrone).



En Simulink et en Scade, ces diagrammes de blocs sont une notation graphique pour des systèmes d'équations entre flux.

La condition de causalité

Dans une équation $x = e$, le membre droit e ne doit pas dépendre (directement ou indirectement) de la valeur instantanée de x, mais peut dépendre de valeurs antérieures de x.

Autrement dit : tout cycle de dépendances doit passer au moins une fois par un opérateur temporel.

Exemples :

$\{x = x + 1\}$ ✘ $\{x = y;\ y = x\}$ ✘

$\{x = 0 \texttt{ fby } x + 1\}$ ✔ $\{x = 0 \texttt{ fby } y;\ y = x + 1\}$ ✔

Sémantique dénotationnelle

La sémantique dénotationnelle du programme $\{\ldots;\ x_i = e_i;\ \ldots\}$
est la solution des équations $x_i = e_i$
(une affectation d'un flux à chaque variable x_i)
pourvu que cette solution existe et soit unique.

Exemples :

$\{x = x + 1\}$	pas de solution
$\{x = y; y = x\}$	plusieurs solutions
$\{x = 0\ \texttt{fby}\ x + 1\}$	solution unique $x = 0, 1, 2, 3, 4, \ldots$

On peut montrer que tout programme causal a une solution unique par des techniques métriques (théorème de point fixe de Banach-Tarski).

Sémantique opérationnelle

(P. Caspi, D. Pilaud, N. Halbwachs, J. A. Plaice, *LUSTRE : A declarative language for programming synchronous systems*, POPL 1987.)

Étant donné un programme $P = \{x_i = e_i\}$, on l'évalue au temps 0 en ≪projetant≫ ses équations

$$P(0) \stackrel{def}{=} \{x_i(0) = now(e_i)\}$$

où *now* : expression de flux → expression numérique est définie par

$$now(v) = v \qquad now(x) = x(0)$$

$$now(v \; \texttt{fby} \; e) = v \quad now(f(\dots, e_i, \dots)) = f(\dots, now(e_i), \dots)$$

Le programme $P(0)$ est une ≪feuille de calcul≫ acyclique → on l'évalue pour déterminer $x_1(0), \dots, x_n(0)$.

Sémantique opérationnelle

On forme le programme résiduel

$$P' \stackrel{def}{=} \{x_i = later(e_i)\}$$

où *later* : expression de flux → expression de flux est définie par

$$\begin{aligned}
later(v) &= v \\
later(x) &= x \\
later(v \ \texttt{fby} \ e) &= v' \ \texttt{fby} \ e \quad \text{où } v' \text{ est la valeur de } e \text{ au temps } 0 \\
later(f(\ldots, e_i, \ldots)) &= f(\ldots, later(e_i), \ldots)
\end{aligned}$$

On itère l'exécution avec P', puis P'', puis …
→ des valeurs $x_i(t)$ pour $i = 1, \ldots, n$ et $t \in \mathbb{N}$
qui sont solutions des équations de flux $x_i = e_i$.

Normalisation des programmes

Tout programme qui respecte la condition de causalité peut être mis sous la forme normale suivante
(en ajoutant des variables et des équations si nécessaire) :

$$
\begin{aligned}
m_1 &= v_1 \ \texttt{fby} \ f_1 && \text{(les mémoires)} \\
&\vdots \\
m_k &= v_k \ \texttt{fby} \ f_k \\
x_1 &= e_1 && \text{(les fils)} \\
&\vdots \\
x_n &= e_n
\end{aligned}
$$

Les expressions e_i et f_j sont instantanées (pas de `fby`).

e_i ne dépend que de $m_1, \ldots, m_k, e_1, \ldots e_{i-1}$ (≪vers l'arrière≫).

f_j ne dépend que de $m_j, \ldots, m_k, e_1, \ldots, e_n$ (≪vers l'avant≫).

Génération de code impératif

À partir de la forme normalisée, il est facile de produire un code impératif qui implémente le programme.

```
// initialisation des mémoires
m1 = v1; ...; mk = vk;
while (true) {
    // acquisition des entrées
    in = read_input();
    // calcul des fils
    x1 = e1; ...; xn = en;
    // production des sorties
    write_output(xn);
    // mise à jour des mémoires
    m1 = f1; ...; mk = fk
}
```

Programmation fonctionnelle

Un langage applicatif (fonctions «de 2e classe»)

On repart des feuilles de calcul (équations $var = expr$) et on ajoute la possibilité de mettre des paramètres aux variables.

Exemple : la suite de Fibonacci.

$$\begin{aligned} F\,n &= \texttt{if}\ n = 0\ \texttt{then}\ 0\ \texttt{else} \\ &\ \texttt{if}\ n = 1\ \texttt{then}\ 1\ \texttt{else} \\ &\ F\,(n-1) + F\,(n-2) \\ x &= F\,20 \end{aligned}$$

Ou, avec un algorithme itératif,

$$\begin{aligned} G\,n\,a\,b &= \texttt{if}\ n = 0\ \texttt{then}\ a\ \texttt{else}\ G\,(n-1)\,b\,(a+b) \\ F\,n &= G\,n\,0\,1 \\ x &= F\,20 \end{aligned}$$

Un langage applicatif

Expressions :

e	$::= 0 \mid 1.2 \mid \ldots$	constantes
	$\mid x$	variables
	$\mid op(e_1, \ldots, e_n)$	opérations prédéfinies
	$\mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$	conditionnelle
	$\mid F\ e_1\ \cdots\ e_n$	application de fonction

Définitions :

def	$::= x = e$	définition de variable
	$\mid F\ x_1\ \cdots\ x_n = e$	définition de fonction

Programmes :

$$P ::= def; \ldots; def$$

Un langage Turing-complet

Ce langage applicatif contient les fonctions récursives partielles de Kleene. Il est donc Turing-complet.

Si on prédéfinit les opérations sur les bandes, il est facile de coder une machine de Turing donnée : chaque état devient une fonction qui prend la bande en paramètre.

$$
\begin{aligned}
S_1\, t &= \texttt{if read}(t) = \mathbf{A} \texttt{ then } S_2\,(\texttt{left}(\texttt{write}(\mathbf{C}, t)) \texttt{ else} \\
&\quad\; \texttt{if read}(t) = \mathbf{B} \texttt{ then } S_3\,(\texttt{right}(\texttt{write}(\mathbf{A}, t)) \texttt{ else} \ldots \\
S_2\, t &= \ldots \\
S_3\, t &= \ldots
\end{aligned}
$$

Structures de contrôle dans un langage applicatif

La conditionnelle est prédéfinie dans le langage.
C'est une expression $\texttt{if}\ \ cond\ \texttt{then}\ e_1\ \texttt{else}\ e_2$
et non pas une commande comme en Algol.

Les boucles sont des fonctions récursives «terminales», p.ex.

$$G\ n\ a\ b = \texttt{if}\ \ n = 0\ \texttt{then}\ a\ \texttt{else}\ G(n-1)\ b\ (a+b)$$

est la boucle $\texttt{while}(n \neq 0)\ \{n = n - 1; (a, b) = (b, a + b); \}$

Le «goto» est présent sous forme d'appels de fonctions en position terminale, p.ex. (NB : CFG non réductible !)

$$\begin{aligned} F\ n &= \texttt{if}\ \ n < 0\ \texttt{then}\ Odd\ (-n)\ \texttt{else}\ Even\ n \\ Even\ n &= \texttt{if}\ \ n = 0\ \texttt{then}\ \texttt{true}\ \texttt{else}\ Odd\ (n-1) \\ Odd\ n &= \texttt{if}\ \ n = 0\ \texttt{then}\ \texttt{false}\ \texttt{else}\ Even\ (n-1) \end{aligned}$$

Un langage fonctionnel (fonctions comme valeurs de 1re classe)

Ajout de l'expression $\lambda x.e$, ≪la fonction qui à x associe e≫.
Plus besoin de distinguer nom de fonction et nom de variable.

Expressions :

$e ::= 0 \mid 1.2 \mid \ldots$	constantes
$\mid x$	variables
$\mid op$	opérations prédéfinies
$\mid \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3$	conditionnelle
$\mid \lambda x.e$	abstraction de fonction
$\mid e_1\ e_2$	application de fonction

Programmes :

$$P ::= x_1 = e_1; \ldots; x_n = e_n$$

Notation : $f\ x_1\ \ldots\ x_n = e$ veut dire $f = \lambda x_1 \ldots \lambda x_n.\ e$

Fonctions d'ordre supérieur

Offrent de nouvelles manières de composer et réutiliser les sous-programmes. Exemples :

$$
\begin{aligned}
\mathit{Compose}\ f\ g &= \lambda x.\, f(g\ x) \\
\mathit{Exp}\ f\ n &= \texttt{if}\ \ n = 0\ \ \texttt{then}\ \ \lambda x.x\ \ \texttt{else} \\
&\quad \mathit{Compose}\ f\ (\mathit{Exp}\ f\ (n-1)) \\
\mathit{Iter}\ \mathit{stop}\ \mathit{step}\ x &= \texttt{if}\ \ \mathit{stop}\ x\ \ \texttt{then}\ \ x\ \ \texttt{else}\ \ \mathit{Iter}\ \mathit{stop}\ \mathit{step}\ (\mathit{step}\ x) \\
\mathit{Map}\ f\ \ell &= \texttt{if}\ \ \ell = \texttt{nil}\ \ \texttt{then}\ \ \texttt{nil}\ \ \texttt{else} \\
&\quad \texttt{cons}\ (f\ (\texttt{head}\ \ \ell))\ (\mathit{Map}\ f\ (\texttt{tail}\ \ \ell)) \\
\mathit{Foldl}\ f\ a\ \ell &= \texttt{if}\ \ \ell = \texttt{nil}\ \ \texttt{then}\ \ a\ \ \texttt{else} \\
&\quad \mathit{Foldl}\ f\ (f\ a\ (\texttt{head}\ \ \ell))\ (\texttt{tail}\ \ \ell)
\end{aligned}
$$

Réduction au lambda-calcul avec constantes

On peut réduire notre langage fonctionnel au lambda-calcul non typé avec constantes

$$e ::= cst \mid x \mid \lambda x.\, e \mid e_1\, e_2$$

à l'aide de quelques codages :

- Définition non récursive :

  $$\texttt{let } x = e_1 \texttt{ in } e_2 \leadsto (\lambda x. e_2)\, e_1$$

- Définition simplement récursive :

  $$\texttt{let rec } f = e_1 \texttt{ in } e_2 \leadsto \texttt{let } f = \mathit{Fix}(\lambda f. e_1) \texttt{ in } e_2$$

  où *Fix* est un combinateur de point fixe tel que $\lambda f.\, (\lambda x.\, f\, (x\, x))\, (\lambda x.\, f\, (x\, x))$.

- Définitions mutuellement récursives $\leadsto$

  un `let rec` + plusieurs `let`.

Règles d'évaluation

Une application de fonction se réduit en le corps de la fonction où le paramètre formel est remplacé par l'argument effectif. (β-réduction dans le λ-calcul ; *copy rule* en Algol.)

$$(\lambda x.\ e)\ e' \rightarrow e\{x \leftarrow e'\}$$

Plus des règles spécifiques aux constantes et aux opérateurs :

$$\texttt{if true then } e_1 \texttt{ else } e_2 \rightarrow e_1$$

$$\texttt{if false then } e_1 \texttt{ else } e_2 \rightarrow e_2$$

$$op\ cst_1\ \ldots\ cst_n \rightarrow cst \qquad \text{si } cst = op^*(cst_1, \ldots, cst_n)$$

Sensibilité à l'ordre des réductions

Propriété de confluence : toutes les séquences de réductions qui terminent le font sur la même forme normale.

Mais certaines séquences de réductions peuvent diverger alors que d'autres terminent, plus ou moins rapidement.

Exemple : si $\Omega \xrightarrow{+} \Omega$, (p.ex. `let rec` $\Omega = \Omega$, i.e. $\Omega = \mathit{Fix}(\lambda f.f)$),

$$(\lambda x.\, 0)\, \Omega \xrightarrow{+} (\lambda x.\, 0)\, \Omega \xrightarrow{+} (\lambda x.\, 0)\, \Omega \xrightarrow{+} \cdots$$

$$(\lambda x.\, 0)\, \Omega \rightarrow 0$$

Les stratégies de réduction classiques

Appel par nom : l'argument d'une fonction est substitué non évalué dans le corps de la fonction.

Évite souvent la divergence, mais duplique beaucoup de calculs :

$$(\lambda x.\, 0)\ \Omega \to 0$$

$$\begin{aligned}(\lambda x.\, x + x)\ (\mathit{Fib}\ 20) &\to \mathit{Fib}\ 20 + \mathit{Fib}\ 20 \\ &\xrightarrow{+} 6765 + \mathit{Fib}\ 20 \xrightarrow{+} 6765 + 6765 \to 13530\end{aligned}$$

Les stratégies de réduction classiques

Appel par nom : l'argument d'une fonction est substitué non évalué dans le corps de la fonction.

Appel par valeur : l'argument d'une fonction est réduit en une valeur avant d'être substitué dans le corps de la fonction.

Économe en calculs, mais risque de divergence :

$$(\lambda x.\ 0)\ \Omega \xrightarrow{+} (\lambda x.\ 0)\ \Omega \xrightarrow{+} (\lambda x.\ 0)\ \Omega \xrightarrow{+} \ldots$$

$$(\lambda x.\ x + x)\ (\mathit{Fib}\ 20) \xrightarrow{+} (\lambda x.\ x + x)\ 6765 \rightarrow 6765 + 6765 \rightarrow 13530$$

Les stratégies de réduction classiques

Appel par nom : l'argument d'une fonction est substitué non évalué dans le corps de la fonction.

Appel par valeur : l'argument d'une fonction est réduit en une valeur avant d'être substitué dans le corps de la fonction.

Appel par nécessité («évaluation paresseuse») :
comme l'appel par nom, mais avec mémoisation des évaluations ou encore réduction de graphes au lieu de réduction de termes.

Évite la divergence sans dupliquer les calculs :

$$(\lambda x.\, 0)\ \Omega \to 0$$

$$(\lambda x.\, x + x)\ (\mathit{Fib}\ 20) \to \mathit{Fib}\ 20 + \mathit{Fib}\ 20 \xrightarrow{+} 6765 + 6765 \to 13530$$

partage (les deux occurrences de $\mathit{Fib}\ 20$) — partage (les deux occurrences de 6765)

Encoder l’appel par nom

Même dans un langage en appel par valeur, on peut forcer une sémantique d’appel par nom en passant les arguments e sous forme de suspensions $\lambda z.e$ (avec z non libre dans e).

(Réduction ≪faible≫ : on ne réduit pas e dans $\lambda z.e$ avant application).

Une transformation de programmes systématique :

$$\mathcal{N}(x) = x\;(\,)$$
$$\mathcal{N}(\lambda x.e) = \lambda x.\,\mathcal{N}(e)$$
$$\mathcal{N}(e_1\;e_2) = \mathcal{N}(e_1)\;(\lambda z.\,\mathcal{N}(e_2))$$

On obtient l’appel par nécessité en utilisant des suspensions mémoisées (`lazy` e en OCaml) au lieu de suspensions $\lambda z.\,e$.

Encoder l'appel par valeur

En utilisant le style à passage de continuations.

$\longrightarrow$ 4e cours

Programmation logique

La programmation logique

Un programme = un ensemble de règles définissant des prédicats.

Une exécution du programme = trouver les valeurs des variables X_i pour lesquelles un prédicat $p(X_1, \ldots, X_n)$ est vrai.

Liens avec la démonstration automatique :
problèmes de satisfiabilité ; algorithme de résolution (Robinson).

Liens avec l'évaluation des requêtes dans les bases de données relationnelles.

Les clauses de Horn

Un ensemble d'affirmations et d'implications de la forme

$$(\forall \vec{x},)\ p \qquad \text{ou} \qquad (\forall \vec{x},)\ q_1 \wedge \cdots \wedge q_n \Rightarrow p$$

où p, q_i sont des littéraux et les variables x_i apparaissant dans les clauses sont implicitement quantifiées universellement.

Ex : $\quad even(0) \qquad \text{ou} \qquad (\forall n,)\ odd(n-1) \Rightarrow even(n).$

Notation Prolog :

```
p .
p :- q1, ..., qn .
```

Autre présentation : axiomes et règles en déduction naturelle.

$$\frac{}{p} \qquad\qquad \frac{q_1 \quad \cdots \quad q_n}{p}$$

Datalog

Littéraux = prédicats sur des variables (majuscules) et des constantes (minuscules).

```
parent(tom, sally).
parent(erica, sally).
parent(tom, bart).
parent(martha, tom).

sibling(X, Y) :- parent(Z, X), parent(Z, Y), X != Y.

ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
```

Littéraux = prédicats sur des termes formés à partir de variables et de constantes par application de constructeurs (p.ex. le constructeur de listes `[X|Y]`, lire «`X` cons `Y`»).

Exemple : permutations de listes.

```
permut([], []).
permut([X|Xs], Ys) :- permut(Xs, Zs), insert(X, Zs, Ys).

insert(X, Ys, [X|Ys]).
insert(X, [Y|Ys], [Y|Zs]) :- insert(X, Ys, Zs).
```

Utilisation pour l'analyse syntaxique

```
s(S0,S) :- np(S0,S1), vp(S1,S).
np(S0,S) :- det(S0,S1), n(S1,S).
vp(S0,S) :- tv(S0,S1), np(S1,S).
vp(S0,S) :- v(S0,S).
det([the|S],S).
det([a|S],S).
det([every|S],S).
n([man|S],S).
n([woman|S],S).
n([park|S],S).
tv([loves|S],S).
tv([likes|S],S).
v([walks|S],S).
```

$$
\begin{aligned}
S &\rightarrow NP\ VP \\
NP &\rightarrow Det\ N \\
VP &\rightarrow TV\ NP \\
VP &\rightarrow V \\
Det &\rightarrow \text{the} \mid \text{a} \mid \text{every} \\
N &\rightarrow \text{man} \mid \text{woman} \mid \text{park} \\
TV &\rightarrow \text{loves} \mid \text{likes} \\
V &\rightarrow \text{walks}
\end{aligned}
$$

Exécution d'un programme Prolog

Par raisonnement en arrière à partir de la requête,
en appliquant la règle de résolution SLD (R. Kowalski) :

> pour montrer le but $g_1 \wedge \cdots \wedge g_n$:
> choisir une clause p `:-` $q_1, \ldots, q_m$
> et un unificateur θ tel que $\theta(p) = \theta(g_i)$;
> montrer $\theta(g_1 \wedge \cdots \wedge g_{i-1} \wedge q_1 \wedge \cdots \wedge q_m \wedge g_{i+1} \wedge \cdots \wedge g_n)$.

Généralement : recherche en profondeur d'abord, avec retour en arrière sur échecs.

Exemples d'exécution

Programme :

```
append([], X, X).
append([H|T], X, [H|U]) :- append(T, X, U).
```

Requêtes :

```
?- append([1,2], [3,4,5], [1,X,3,4,Y]).
% X = 2, Y = 5.
?- append([1,2], [3,4,5], X).
% X = [1, 2, 3, 4, 5].
?- append([1,2], X, [1,2,3,4,5]).
% X = [3, 4, 5]
?- append(X, [3,4,5], [1,2,3,4,5]).
% X = [1, 2]
?- append(X, Y, [1,2])
% X = [], Y = [1, 2] ; X = [1], Y = [2] ; X = [1, 2], Y = []
```

Sensibilité à la stratégie de résolution

La plupart des systèmes Prolog implémentent un chaînage arrière avec recherche en profondeur d'abord avec retour en arrière, sans mémoisation.

Cela peut trivialement diverger :

```
p(X) :- p(X).
q(X) :- q(f(X)).
```

Ou moins trivialement :

```
rstar(X, X).
rstar(X, Y) :- rstar(X, Z), r(Z, Y).
```

(Produit le premier résultat et diverge ensuite.)

Sensibilité à la stratégie de résolution

Le retour en arrière (*backtracking*) entraîne souvent des calculs inutiles car ne produisant pas de solutions supplémentaires.

```
list_mem(X, [X|_]).
list_mem(X, [_|Ys]) :- list_mem(X, Ys).

list_add(X, L, L) :- list_mem(X, L).
list_add(X, L, [X|L]) :- not(list_mem(X, L)).
```

Il n'y a qu'une solution à $\texttt{list_add}(0, [\overbrace{0, \ldots, 0}^{n \text{ zéros}}], \texttt{X})$
mais *n* manières différentes de l'obtenir !
$\rightarrow$ Calculer toutes les solutions prend un temps $\mathcal{O}(n^2)$.

L’opérateur de coupure (*cut*)

L’opérateur ≪ `!` ≫ (coupure, *cut*) permet de contrôler le retour en arrière.

```
list_add(X, L, L) :- list_mem(X, L), ! .
list_add(X, L, [X|L]) :- not(list_mem(X, L)).
```

L’exécution de ≪ `!` ≫ supprime les alternatives courante pour `list_add`, c.à.d. ≪réessayer `list_mem(X, L)`≫ et ≪essayer la 2e clause de `list_add`≫.

La coupure peut aussi servir à exprimer la négation :

```
list_add(X, L, L) :- list_mem(X, L), ! .
list_add(X, L, [X|L]).
```

La coupure : le «goto» des langages logiques ?

Un mécanisme de bas niveau ; sémantique formelle peu claire.

Change non seulement l'efficacité mais aussi la sémantique des programmes ! (*green cuts* vs. *red cuts*). Exemple :

```
p(a) :- ! .
p(b).
```

vs.

```
p(b).
p(a) :- ! .
```

Alternatives :

- Opérateurs spéciaux, p.ex. `firstof`(p).
- Structures de contrôle supplémentaires, p.ex. la conditionnelle `(`p `->` q `;` r`)`.
- Méta-langages pour contrôler la stratégie de résolution.

Point d’étape

La programmation déclarative

Un équilibre délicat à trouver entre

> décrire davantage ce qu'il faut calculer,
> moins comment le calculer

et

> maîtriser la terminaison et la complexité
> (en temps, en espace) des programmes.

Des structures de contrôle restent nécessaires,
non plus pour contrôler l'enchaînement exact des opérations,
mais pour contrôler la stratégie générale d'exécution.

Trois familles de langages déclaratifs

Langages réactifs :

- Déclaratifs ET performants !
- Au prix d'une expressivité très restreinte.

Langages fonctionnels :

- Il est nécessaire de fixer la stratégie de réduction.
- Appel par valeur : un modèle de coût intuitif.
- Appel par nécessité : meilleures propriétés théoriques, mais modèle de coûts difficile à maîtriser.

Langages logiques :

- Il est nécessaire de contrôler la stratégie de résolution.
- Par quels moyens ? pas de consensus.

Bibliographie

Bibliographie

La programmation déclarative vue sous l'angle ≪fonctionnel≫ :

- H. Abelson, G. J. Sussman. *Structure and Interpretation of Computer Programs*, MIT Press, 1996. Chapitre 1.

La programmation déclarative vue sous l'angle ≪logique≫ :

- P. Van Roy, S. Haridi. *Concepts, Techniques, and Models of Computer Programming*, MIT Press, 2004. Chapitres 1, 2, 3 et 9.

Une introduction au langage Lustre :

- N. Halbwachs, P. Raymond. *A tutorial of Lustre*, 2007. https://www-verimag.imag.fr/DIST-TOOLS/SYNCHRONE/lustre-v4/distrib/lustre_tutorial.pdf