Cours 5
Mémoire–bis

Jean-Jacques.Levy@inria.fr

http://jeanjacqueslevy.net

secrétariat de l'enseignement:

Catherine Bensoussan

cb@lix.polytechnique.fr

Laboratoire d'Informatique de l'X

Aile 00, LIX

tel: 34 67

http://w3.edu.polytechnique.fr/informatique

Références

- Programming Languages, Concepts and Constructs, Ravi Sethi, 2nd edition, 1997.
  http://cm.bell-labs.com/who/ravi/teddy/
- Theories of Programming Languages, J. Reynolds, Cambridge University Press, 1998.
  http://www.cs.cmu.edu/afs/cs.cmu.edu/user/jcr/www/
- Type Systems for Programming Languages, Benjamin C. Pierce, Cours University of Pennsylvania, 2000.
  http://www.cis.upenn.edu/~bcpierce/
- Programming Languages: Theory and Practice, Robert Harper, Cours Carnegie Mellon University, Printemps 2000. http://www.cs.cmu.edu/~rwh/
- Notes du cours de DEA "Typage et programmation", Xavier Leroy, Cours de DEA, Décembre 1999,
  http://pauillac.inria.fr/~xleroy/dea

Plan

1. **Complexité de l'inférence de type de PCF**
2. **Typage des références**
3. **Variables modifiables**
4. *L-values* **et** *R-values*
5. **Assertions de correction**
6. **Correction faible, correction forte**
7. **Alias**
8. **Équivalences de type**

Complexité de l'inférence de type de PCF

Mairson a montré qu'elle pouvait être en $O(2^{2^n})$. Mais en général elle est quasi linéaire.

Exemple problématique

let $x_0 = \lambda x.(x, x)$ in

let $x_1 = \lambda y.x_0(x_0\ y)$ in

let $x_2 = \lambda y.x_1(x_1\ y)$ in

...

let $x_n = \lambda y.x_{n-1}(x_{n-1}\ y)$ in

$x_n\ (\lambda z.z)$

Exercice 1 Calculer son type et essayer de comprendre pourquoi il est compliqué.

Typage des références

Le typage polymorphe des références viole le théorème de progression.

$$\mathtt{let}\ r = \mathtt{ref}\,\mathtt{nil}\ \mathtt{in}\ r := \underline{3} :: \mathtt{nil}; (\mathtt{hd}\ !r)\ \underline{2}$$

Il faut donc se souvenir des valeurs stockées dynamiquement dans une référence pour ne pas avoir de contradictions.

Solution 1 Toutes les références sont monomorphes.

Alors $!$ ou $:=$ ne sont plus typables. Mais on peut typer $!M$ et $M := N$. On ne sait plus aussi typer les variables références polymorphes dans les tris, par exemple.

Solution 2 [Leroy] Dans $\mathtt{let}\ x = M\ \mathtt{in}\ N$, on ne généralise pas les variables de type apparaissant dans le type d'une référence allouée dans l'évaluation de M.

Dans Ocaml, $\_\alpha$ désigne une variable de type non généralisable. Par exemple, $\mathtt{ref}(\lambda x.x) : \mathtt{ref}(\_\alpha \to \_\alpha)$.

Variables modifiables

Termes

M, N, P	$::=$	$\ldots$	**voir cours précédents**
	$\mid$	$\texttt{letvar}\, x \leftarrow M \texttt{ in } N$	**création**
	$\mid$	x	**valeur**
	$\mid$	$x \leftarrow M$	**modification de valeur**

Exemples

$\texttt{letvar}\, c \leftarrow \underline{0} \texttt{ in let } x = c \leftarrow c + 1 \texttt{ in } c$

$\texttt{letvar}\, c \leftarrow \underline{0} \texttt{ in let } f = (\lambda x.\texttt{let } y = c \leftarrow c + 1 \texttt{ in } x) \texttt{ in } f(\underline{4}) + f(\underline{5}) + c$

$\texttt{letvar}\, c \leftarrow \underline{0} \texttt{ in let } f = (\lambda x.\texttt{let } y = c \texttt{ in let } z = c \leftarrow x \texttt{ in } y) \texttt{ in } f(\underline{2}) + f(\underline{3})$

$\texttt{letvar}\, c \leftarrow \underline{0} \texttt{ in } c \leftarrow c + 1; c$

$\texttt{letvar}\, c \leftarrow \underline{0} \texttt{ in let } f = (\lambda x.c \leftarrow c + 1; x) \texttt{ in } f(\underline{4}) + f(\underline{5}) + c$

$\texttt{letvar}\, c \leftarrow \underline{0} \texttt{ in let } f = (\lambda x.\texttt{let } y = c \texttt{ in } c \leftarrow x; y) \texttt{ in } f(\underline{2}) + f(\underline{3})$

Règles de réductions

alloc_v	$\langle \texttt{letvar}\, x \leftarrow V \;\texttt{in}\; N,\, s\rangle \rightarrow \langle N[x \backslash !\ell],\, s + [\ell = V]\rangle \quad (\ell \notin \mathit{domain}(s))$
deref_v	$\langle !\ell,\, s\rangle \rightarrow \langle s(\ell),\, s\rangle$
assign	$\langle !\ell \leftarrow V,\, s\rangle \rightarrow \langle (),\, s[\ell \backslash V]\rangle$

On distingue deux sortes d'occurrences d'une variable x modifiable:

- **occurrences gauches: à gauche d'une affectation $x \leftarrow M$.**
- **occurrences droites: partout sauf le cas précédent.**

On peut traduire `letvar` dans le calcul précédent, en remplacant toute occurence droite d'une variable modifiable x par $!x$ et en changeant toute déclaration $\texttt{letvar}\, x \leftarrow M \;\texttt{in}\; N$ par $\texttt{let}\; x = \texttt{ref}\; M \;\texttt{in}\; N$.

Exemple:

$\texttt{let}\; c = \texttt{ref}\, \underline{0} \;\texttt{in}\; c := !c + 1;\; !c$

$\texttt{let}\; c = \texttt{ref}\, \underline{0} \;\texttt{in}\; \texttt{let}\; f = (\lambda x.c := !c + 1; x) \;\texttt{in}\; f(\underline{4}) + f(\underline{5}) + !c$

$\texttt{let}\; c = \texttt{ref}\, \underline{0} \;\texttt{in}\; \texttt{let}\; f = (\lambda x.\, \texttt{let}\; y = !c \;\texttt{in}\; c := x; y) \;\texttt{in}\; f(\underline{2}) + f(\underline{3})$

Remarque

$\langle M, s\rangle \to \langle M', s'\rangle$ **implique** $\langle C[M], s\rangle \to \langle C[M'], s'\rangle$ **si** $C[\,] \neq C'[[\,] \leftarrow N]$

Valeur gauche — Valeur droite

Les variables modifiables standard des langages impératifs ont donc deux types de valeurs possibles:

- à gauche de $\leftarrow$, leur valeur est la référence vers leur contenu (*L-value*)
- à droite de $\leftarrow$ et partout ailleurs, leur évaluation donne leur contenu (avec déréférencement automatique) (*R-value*)

On a aussi ce phénomène avec les éléments de tableaux, puisque dans $v[i] \leftarrow v[i] + 1$, l'évaluation n'est pas la même des deux cotés de $\leftarrow$.

Certains langages permettent les appels de paramètre par référence (Fortran, Pascal, C++). Alors c'est la valeur gauche de l'argument qui se substitue au paramètre.

D'autres langages ne les autorise pas, car trouvant l'écriture dangereuse (C, Java, ML).

Exercice (difficile) Rajouter les paramètres par référence à PCF.

Exemple

Considérons le calcul itératif suivant des nombres de Fibonacci.

$\texttt{ifz}\ n\ \texttt{then}\ x \leftarrow \underline{0}\ \texttt{else}$
$\quad \texttt{letvar}\ y \leftarrow \underline{0}\ \texttt{in}$
$\quad x \leftarrow \underline{1};$
$\quad \texttt{letvar}\ k \leftarrow \underline{1}\ \texttt{in}$
$\quad \texttt{while}\ k \neq n\ \texttt{do}$
$\qquad \texttt{let}\ t = y\ \texttt{in}\ ($
$\qquad\quad y \leftarrow x;$
$\qquad\quad x \leftarrow x + t;$
$\qquad\quad k \leftarrow k + \underline{1})$

On veut montrer que $n \geq 0$ au début de ce programme implique $x = \mathit{fib}(n)$ à la fin.

Assertions

$\{n \geq 0\}$

$\texttt{ifz}\, n\, \texttt{then}\, x \leftarrow \underline{0}\, \texttt{else}$

$\quad \{n > 0 \wedge x = 0\}$

$\quad \texttt{letvar}\, y \leftarrow \underline{0}\, \texttt{in}$

$\quad x \leftarrow \underline{1};$

$\quad \{n > 0 \wedge x = \mathit{fib}\,(1) \wedge y = \mathit{fib}\,(0)\}$

$\quad \texttt{letvar}\, k \leftarrow \underline{1}\, \texttt{in}$

$\quad \{k \leq n \wedge x = \mathit{fib}\,(k) \wedge y = \mathit{fib}\,(k-1)\}$

$\quad \texttt{while}\, k \neq n\, \texttt{do}$

$\qquad \{k < n \wedge x = \mathit{fib}\,(k) \wedge y = \mathit{fib}\,(k-1)\}$

$\qquad \texttt{let}\ t = y\ \texttt{in}\ ($

$\qquad\quad \{k < n \wedge x = \mathit{fib}\,(k) \wedge y = \mathit{fib}\,(k-1) \wedge t = \mathit{fib}\,(k-1)\}$

$\qquad\quad y \leftarrow x;$

$\qquad\quad \{k < n \wedge x = \mathit{fib}\,(k) \wedge y = \mathit{fib}\,(k) \wedge t = \mathit{fib}\,(k-1)\}$

$\qquad\quad x \leftarrow x + t;$

$\qquad\quad \{k < n \wedge x = \mathit{fib}\,(k+1) \wedge y = \mathit{fib}\,(k) \wedge t = \mathit{fib}\,(k-1)\}$

$\qquad\quad k \leftarrow k + \underline{1})$

$\{x = \mathit{fib}\,(n)\}$

Invariants de boucle

$\{n \geq 0\}$

$\texttt{ifz}\, n\, \texttt{then}\, x \leftarrow \underline{0}\, \texttt{else}$

$\quad \{n > 0 \wedge x = 0\}$

$\quad \texttt{letvar}\, y \leftarrow \underline{0}\, \texttt{in}$

$\quad x \leftarrow \underline{1};$

$\quad \{n > 0 \wedge x = \mathit{fib}\,(1) \wedge y = \mathit{fib}\,(0)\}$

$\quad \texttt{letvar}\, k \leftarrow \underline{1}\, \texttt{in}$

$\quad \{k \leq n \wedge x = \mathit{fib}\,(k) \wedge y = \mathit{fib}\,(k-1)\}$

$\quad \texttt{while}\, k \neq n\, \texttt{do}$

$\qquad \{k < n \wedge x = \mathit{fib}\,(k) \wedge y = \mathit{fib}\,(k-1)\}$

$\qquad \texttt{let}\ t = y\ \texttt{in}\ ($

$\qquad\quad \{k < n \wedge x = \mathit{fib}\,(k) \wedge y = \mathit{fib}\,(k-1) \wedge t = \mathit{fib}\,(k-1)\}$

$\qquad\quad y \leftarrow x;$

$\qquad\quad \{k < n \wedge x = \mathit{fib}\,(k) \wedge y = \mathit{fib}\,(k) \wedge t = \mathit{fib}\,(k-1)\}$

$\qquad\quad x \leftarrow x + t;$

$\qquad\quad \{k < n \wedge x = \mathit{fib}\,(k+1) \wedge y = \mathit{fib}\,(k) \wedge t = \mathit{fib}\,(k-1)\}$

$\qquad\quad k \leftarrow k + \underline{1})$

$\{x = \mathit{fib}\,(n)\}$

Méthode des assertions [Floyd, Manna]

On a placé des assertions logiques en divers points du programme à vérifier quand le contrôle passe en ce point. Certains langages (Caml) permettent l'écriture de telles assertions, et l'interpréteur lève une exception si l'assertion est fausse.

Les assertions peuvent aussi se démontrer statiquement, avant que le programme ne s'exécute. Il faut alors faire une preuve dans un système logique contenant des formules du genre

$$\vdash \{P\}M\{Q\}$$

où P et Q sont deux prédicats et M un terme de programme.

Cette formule est vraie si $P(s)$ et $\langle M, s\rangle \rightarrow^* \langle V, s'\rangle$ impliquent $P(s')$. Autrement dit, si la pré-condition P est vraie avant l'exécution de M, et si M termine, la post-condition Q est vraie après l'exécution de M.

Logique de Floyd-Hoare

$$\vdash \{P[x\backslash M]\}\ x \leftarrow M\ \{P\}$$

$$\frac{\vdash P \supset P' \qquad \vdash \{P'\}\ M\ \{Q'\} \qquad \vdash Q' \supset Q}{\vdash \{P\}\ M\ \{Q\}}$$

$$\frac{\vdash \{P\}\ M\ \{Q\} \qquad \vdash \{Q\}\ N\ \{R\}}{\vdash \{P\}\ M; N\ \{R\}}$$

$$\frac{\vdash \{P \wedge Q\}\ N\ \{P\}}{\vdash \{P\}\ \mathtt{while}\, Q\, \mathtt{do}\ N\ \{P \wedge \neg Q\}}$$

$$\frac{\vdash \{P \wedge R\}\ M\ \{Q\} \qquad \vdash \{P \wedge \neg R\}\ N\ \{Q\}}{\vdash \{P\}\ \mathtt{if}\, R\, \mathtt{then}\ M\ \mathtt{else}\ N\ \{Q\}}$$

$$\vdash \{P\}\ ()\ \{P\}$$

$$\frac{\vdash \{P\}\ x \leftarrow M; N\ \{Q\} \qquad (x \notin FV(P) \cup FV(Q))}{\vdash \{P\}\ \mathtt{letvar}\, x \leftarrow M\ \mathtt{in}\ N\ \{Q\}}$$

$$\frac{\vdash \{P\}\ M\ \{P'\} \qquad \vdash \{Q\}\ M\ \{Q'\}}{\vdash \{P \wedge Q\}\ M\ \{P' \wedge Q'\}}$$

$$\frac{\vdash \{P\}\ M\ \{P'\} \qquad \vdash \{Q\}\ M\ \{Q'\}}{\vdash \{P \vee Q\}\ M\ \{P' \vee Q'\}}$$

Exercice Se servir de la logique ainsi décrite pour faire rigoureusement la preuve de correction du calcul de Fibonacci.

Correction partielle – Correction totale

Il n'est pas question de faire ici un cours sur la vérification (cf. DEA Sémantique, Preuve et Programmation).

$$(1) \qquad \vdash \{x \leq 10\} \; \texttt{while} \, x \neq 10 \; \texttt{do} \; x \leftarrow x + 1 \; \{x = 10\}$$

$$(2) \qquad \vdash \{\textsf{vrai}\} \; \texttt{while} \, x \neq 10 \; \texttt{do} \; x \leftarrow x + 1 \; \{x = 10\}$$

$$(3) \qquad \vdash \{x > 10\} \; \texttt{while} \, x \neq 10 \; \texttt{do} \; x \leftarrow x + 1 \; \{\textsf{faux}\}$$

(1) est fortement (totalement) correct,
(2) et (3) sont partiellement corrects

La pré-condition de (1) garantit la terminaison.

Chaque fois que l'on a une difficulté avec un morceau de programme critique, il est bon d'écrire formellement les assertions à vérifier. Faire la preuve mécaniquement est beaucoup plus difficile. Programmes zéro-fautes.

Alias

Deux références x et y sont deux alias dans un état mémoire s ssi $s(x) = \ell = s(y)$. Par exemple

$\texttt{let}\ x = \texttt{ref}\ \underline{4}\ \texttt{in}$
$\texttt{let}\ y = x\ \texttt{in}$
$\quad x := 3;$
$\quad !y$

Dans un langage sans références explicites, les alias sont plus difficiles à détecter

$\texttt{letvar}\ x \leftarrow \underline{4}\ \texttt{in}$
$\texttt{let}\ y = x\ \texttt{in}$
$\quad x := 3;$
$\quad y$

Les alias existent aussi avec les éléments de tableaux. Par $v[i]$ est un alias de $v[j]$ quand $i = j$.

En Pascal, ca se complique avec les paramètres par référence. En C, c'est aussi assez complexe puisqu'on peut prendre l'adresse de tout objet. En C++, on peut faire les deux!!

Détection d'alias

Trouver les alias, ou prouver leur inexistence, permet de détecter des modules indépendants. Important pour

- optimisation de code. On fait alors des modifications de code plus agressives.
- parallélisation. Deux modules indépendants peuvent s'exécuter en parallèle. Le cas pour les tableaux en Fortran.
- vérification de programme. Par exemple, l'absence d'alias permet de garantir la validité de l'exécution concurrente.

En TD – la suite

- ajouter les références à l'évaluateur symbolique de PCF
- finir l'interpréteur et ajouter les références
- faire le vérificateur de type ou le synthétiseur

A la maison et prochaine fois

- Alias
- Programmation logique
- Prolog