Programmation fonctionnelle

Joëlle Despeyroux
INRIA Sophia-Antipolis

Module
"Méthodes formelles et fiabilité du logiciel"
Dea d'informatique, Nice, automne 2000

Introduction

Les langages de programmation:

– Langages impératifs: Fortran, Algol/Pascal/Ada.

– Langage pour l'intelligence artificielle: Lisp.

– Prolog, langage basé sur la logique d'ordre 1.

– Langages fonctionnels: ML, Caml, ..., basés sur le λ-calcul.

Les langages fonctionnels ont une sémantique formelle, permettant de faire des preuves.
Cependant, les langages réels ajoutent au noyau fonctionnel des primitives impératives (entrées/sorties, exceptions, structures modifiables, ...).
Choix ici: Caml, développé par l'équipe Cristal à l'INRIA Rocquencourt.
Référence: *"The Functional Approach to Programming", G. Cousineau et M. Mauny, Cambridge Univ. Press, 1998* .

Expressions

Un programme est une liste d'expressions-définitions, suivies d'une expression à évaluer.
Les expressions de base sont contruites à partir des variables, des fonctions et de l'application de fonctions: le noyau du langage est *le λ-calcul* à la Curry.
Les expressions sont typées. On peut définir des types.

Expressions:
$4/2 * 3 + 1$;; — -: int = 7

Définitions:
let $x = 2 + 3$;; — x: int = 6
let $pi = 3.14$;; — pi: float = 3.14
$x + 1$;; — -: int = 7

Définitions locales:
let $x = 6$ and $y = 1$ in $x + y$;; — -: int = 7
$x + y$ where $x = 6$ and $y = 1$;; — -: int = 7

Types

Types simples:
Les entiers (int avec + etc), les flotants (float avec +. etc), les booléens (bool), les caractères (char) et les chaînes (string).

Produit Cartésien:
fst $((1, 2.0), (3, 4))$;; — - int * float = 1, 2.0

Fonctions:
let $add3(x) = x + 3$;; — add3 = int → int = <fun>
fun $x = x + 3$;; — - = int → int = <fun>

Fonctions d'ordre supérieur:
let $h =$ (fun $f \rightarrow$ (fun $x \rightarrow f(x) + x$));;
let $h =$ fun $f\ x \rightarrow f(x) + x$;;
— h : (int → int) → int → int = <fun>

Types (suite)

Fonctions récursives:

let rec $fact(n) = $ if $n = 0$ then 1 else $n * fact(n-1)$;;
— fact : int $\rightarrow$ int = <fun>

Fonctions mutuellement récursives:

let rec $even(n) = $ if $n = 0$ then $true$ else $odd(n-1)$
and $odd(n) = $ if $n = 0$ then $false$ else $even(n-1)$;;
— even : int $\rightarrow$ bool = <fun>
odd : int $\rightarrow$ bool = <fun>

(Fonction) analyse par cas:

let rec $fact(n) = $ Match n with
$0 \rightarrow 1$
$\mid m \rightarrow m * fact(m-1);;$

Polymorphisme:

let $id = $ fun $x \rightarrow x$;; — id : $\alpha \rightarrow \alpha$ = <fun>
let $fst(x, y) = x$;; — fst : $\alpha \rightarrow \beta \rightarrow \alpha$ = <fun>
Les types sont *synthétisés*, en utilisant un algorithme d'*unification*.

Types (suite)

Example: itération

let rec $iter\ n\ f = $ if $n = 0$ then id else $f \circ (iter\ (n-1)\ f)$;;
— iter : int $\rightarrow (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$ = <fun>
let rec $loop\ p\ f\ x = $ if $(p\ x)$ then x else $loop\ p\ f\ (f\ x)$;;
— loop : $(\alpha \rightarrow bool) \rightarrow (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$ = <fun>
Correspond à for et while dans les lang. imp.

Discussion.
iter: *fonctions primitives récursives.*
loop: *fonctions récursives.*
Les fonctions primitives récursives sont *totales* par construction tandis que les fonctions construites avec loop peuvent ne pas terminer.
Il existe des fonctions récursives totales qui ne sont pas primitives récursives (fonctions loop qui terminent, et ne peuvent s'écrire avec iter).

Structures de données

Enregistrements (produits cartésiens nommés)

let $x = \{re = 1.; im = 0.\}$;;

— x: complex = $\{re = 1; im = 0\}$

let $add_c(x, y) = \{re = x.re +_. y.re; im = x.im +_. y.im\}$;;

— -: complex $\rightarrow$ complex $\rightarrow$ complex

type $point = \{c_x : float; c_y : float\}$;;

type $triangle = \{pA : point; pB : point; pC : point\}$;;

Sommes (unions)

type $bool = true \mid false$;;

type $numbers = Int$ of $int \mid Float$ of $float$;;

$Int(3)$;; — -: num = Int 3

(Sommes) types récursifs

type $tree = Leaf$ of $int \mid Node$ of $tree * tree$;;

Exercice. Compter les valeurs des feuilles d'un arbre.

let rec val = fun $(Leaf\ n) \rightarrow n$

$\mid Node\ (t_1, t_2) \rightarrow val(t_1) + val(t_2)$; ;

type $exp = Cst$ of int

$\mid Var$ of $string$

$\mid Add$ of $exp * exp$

$\mid \cdots$; ;

Structures de données (suite)

(Sommes) types polymorphes

type $\alpha\ tree = Leaf$ of $\alpha \mid Node$ of $\alpha\ tree * \alpha\ tree$;;

Listes

type $\alpha\ list = Nil \mid Cons$ of $\alpha * \alpha\ list$;;

- Syntaxe: $[x; y; z]@[]$.

Exercices.

Définir les fonctions: longueur d'une liste, concaténation de deux listes, somme des éléments d'une liste, ...

Sémantique

Sémantique dénotationnelle

D: domaine des valeurs.
Les fonctions sont des valeurs. $(D \to D) \in D$?
$D \to D =$ l'ens. des *fonctions continues* de D dans D.
Sémantique bien définies mais verbeuses et mathématiques relativement difficiles.
D'où des preuves relativement compliquées et fastidieuses.

Sémantique opérationnelle - réécriture

Règles d'évaluation.
Sémantique et techniques de preuves simples.
Convient mieux pour pour les preuves en machine, partiellement automatisées.

Sémantique - Évaluation

Propriétés des systèmes de réduction

- *système fini*: toute réduction termine,
- *convergence*:
$\forall e, e_1, e_2, e \to e_1$ *et* $e \to e_2 \Rightarrow \exists e_3$ *tq* $e_1 \to e_3$ *et* $e_2 \to e_3$.
- *consistance*: toutes les réductions conduisent au même résultat.
- *consistance faible*: toutes les réductions qui terminent conduisent au même résultat.

Ici, on ne peut pas demander plus car certaines expressions peuvent avoir des réductions qui terminent et d'autres qui ne terminent pas.

Substitution

$(\text{fun } x \to yx)[3/y] = (\text{fun } x \to 3x)$.

Capture de variables:
$(\text{fun } x \to yx)[2x/y] \neq (\text{fun } x \to 2xx)$.

Renommage de variables:
$(\text{fun } x \to yx) = (\text{fun } z \to yz)$.
$(\text{fun } x \to yx)[2x/y] = (\text{fun } z \to yz)[2x/y]$
$= (\text{fun } z \to 2xz)$.

Sémantique - Évaluation (suite)

Stratégies d'évaluation

- *Évaluation par valeur*: on évalue les arguments d'une fonction avant d'appliquer la fonction.

- *Évaluation paresseuse*: on reporte l'évaluation des arguments d'une fonction au moment où on en a besoin pour progresser. Meilleure stratégie dans les cas où une branche infinie est inutile. Exs: conditionnelle, ou.

- *Stratégie de réduction complète*: elle conduit toujours à une valeur, si elle existe.

La stratégie d'évaluation par valeur est incomplète, mais permet au programmeur de mieux gérer les primitives impératives (ordre d'évaluation fixé). $\rightarrow$ stratégie choisie pour Caml.

Règles d'évaluation

$$tuple : \frac{e_1 \hookrightarrow v_1 \quad \cdots \quad e_n \hookrightarrow v_n}{(e_1, \cdots, e_n) \hookrightarrow (v_1, \cdots, v_n)}$$

$$if_1 : \frac{e_1 \hookrightarrow true \qquad e_2 \hookrightarrow v}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \hookrightarrow v} \qquad if_2 : \frac{e_1 \hookrightarrow false \qquad e_3 \hookrightarrow v}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \hookrightarrow v}$$

$$fun : (\text{fun } x \rightarrow e) \hookrightarrow (\text{fun } x \rightarrow e)$$

$$fun_n : (\text{fun } (x_1, \cdots, x_n) \rightarrow e) \hookrightarrow (\text{fun } (x_1, \cdots, x_n) \rightarrow e)$$

$$app_{op} : \frac{e \hookrightarrow op \quad e_i \hookrightarrow v_i \quad op\ v_1 \cdots v_n \hookrightarrow v}{(e\ e_1 \cdots e_n) \hookrightarrow v}$$

$$app : \frac{e_1 \hookrightarrow (\text{fun } x \rightarrow e) \quad e_2 \hookrightarrow v_2 \quad e[v_2/x] \hookrightarrow v}{(e_1\ e_2) \hookrightarrow v}$$

$$app_n : \frac{e_1 \hookrightarrow (\text{fun } (x_1, \cdots, x_n) \rightarrow e) \quad e_2 \hookrightarrow (v_1, \cdots, v_n) \quad e[v_i/x_i] \hookrightarrow v}{(e_1\ e_2) \hookrightarrow v}$$

$$let : \frac{e_1 \hookrightarrow v_1 \qquad e_2[v_1/x] \hookrightarrow v}{\text{let } x = e_1 \text{ in } e_2 \hookrightarrow v}$$

$$rec : \frac{f\ (\text{Rec } f)\ x \hookrightarrow v}{\text{Rec } f\ x \hookrightarrow v} \qquad rec_0 : (\text{Rec } f) \hookrightarrow (\text{Rec } f)$$

$$letrec : \frac{e_2[\text{Rec } (\text{fun } f\ x \rightarrow e_1)/f] \hookrightarrow v}{\text{let rec } f\ x = e_1 \text{ in } e_2 \hookrightarrow v}$$

Évaluation - exemples

$$\dfrac{(\text{fun } x \to x) \hookrightarrow (\text{fun } x \to x) \qquad 0 \hookrightarrow 0 \qquad x[0/x] \equiv 0 \hookrightarrow 0}{((\text{fun } x \to x)\ \ 0) \hookrightarrow 0}\ app$$

On pose: $F = (\text{if } n = 0 \text{ then } 1 \text{ else } n * fact(n-1))$
$Fun = (\text{fun } fact\ n \to F) = (\text{fun } fact \text{ fun } n \to F)$
$F' = (\text{if } n = 0 \text{ then } 1 \text{ else } n * \text{Rec } Fun\ (n-1))$

$$\dfrac{\dfrac{\dfrac{A \qquad 1 \hookrightarrow 1 \qquad B}{(Fun\ (\text{Rec } Fun)\ 1) \hookrightarrow 1}}{(fact\ 1)[\text{Rec } Fun/fact] = (\text{Rec } Fun\ 1) \hookrightarrow 1}}{\text{let rec } fact\ n = F \text{ in } fact(1) \hookrightarrow 1}$$

où A est:

$$\dfrac{Fun \hookrightarrow Fun \qquad (\text{Rec } Fun) \hookrightarrow (\text{Rec } Fun) \qquad (\text{fun } n \to F)[\text{Rec } Fun/fact] = (\text{fun } n \to F')}{(Fun\ (\text{Rec } Fun)) \hookrightarrow (\text{fun } n \to F')}$$

et B est:

$$\dfrac{(1 = 0) \hookrightarrow false \qquad \dfrac{\dfrac{A \qquad 0 \hookrightarrow 0 \qquad B'}{(Fun\ (\text{Rec } Fun)\ 0)) \hookrightarrow 1}}{(\text{Rec } Fun\ 0) \hookrightarrow 1}}{F'[1/n] = (\text{if } 1 = 0 \text{ then } 1 \text{ else } (\text{Rec } Fun\ 0)) \hookrightarrow 1}$$

avec B':

$$\dfrac{(0 = 0) \hookrightarrow true \qquad 1 \hookrightarrow 1}{F'[0/n] = (\text{if } 0 = 0 \text{ then } 1 \text{ else } (\text{Rec } Fun\ 0)) \hookrightarrow 1}$$

Évaluation paresseuse (appel par nom)

$tuple : (e_1, \cdots, e_n) \hookrightarrow (e_1, \cdots, e_n)$

$$app : \dfrac{e_1 \hookrightarrow (\text{fun } x \to e) \quad e[e_2/x] \hookrightarrow v}{(e_1\ e_2) \hookrightarrow v}$$

$$app_n : \dfrac{e_1 \hookrightarrow (\text{fun } (x_1, \cdots, x_n) \to e) \quad e_2 \hookrightarrow (v_1, \cdots, v_n) \quad e[v_i/x_i] \hookrightarrow v}{(e_1\ e_2) \hookrightarrow v}$$

$$let : \dfrac{e_2[e_1/x] \hookrightarrow v}{\text{let } x = e_1 \text{ in } e_2 \hookrightarrow v} \qquad rec : (\text{Rec } f) \hookrightarrow f\ (\text{Rec } f)$$

Remarque.
$(\text{let } x = e_1 \text{ in } e_2) \equiv (\text{fun } x \to (e_2\ \ e_1))$ pour l'évaluation, quelque soit la stratégie.

Structures de données:
Pattern matching:
fun $p_1 \to e_1 \mid \cdots \mid p_n \to e_n$ où p_i est de la forme $C_i(x_1, \cdots, x_{n_i})$.

$Constr : C(e_1, \cdots, e_n) \hookrightarrow C(e_1, \cdots, e_n)$

$$app_c : \dfrac{\begin{array}{c} e \hookrightarrow (\text{fun } \cdots \mid C_i(x_1, \cdots, x_{n_i}) \to E \mid \cdots) \\ e' \hookrightarrow C_i(e_1, \cdots, e_{n_i}) \quad E[e_j/x_j] \hookrightarrow v \end{array}}{(e\ e') \hookrightarrow v}$$

Évaluation - en Sémantique Naturelle

Plus efficace.

valeurs: *valeur* $v ::= bv \mid (v_1, \cdots, v_n) \mid [\text{fun} \;\; x \to e; \rho]$

évaluation:

$$var : \frac{\Gamma(x) = v}{\rho \vdash x \hookrightarrow v} \quad tuple : \frac{\rho \vdash e_1 \hookrightarrow v_1 \quad \cdots \quad \rho \vdash e_n \hookrightarrow v_n}{\rho \vdash (e_1, \cdots, e_n) \hookrightarrow (v_1, \cdots, v_n)}$$

$$if_1 : \frac{\rho \vdash e_1 \hookrightarrow true \qquad \rho \vdash e_2 \hookrightarrow v}{\rho \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \hookrightarrow v}$$

$$if_2 : \frac{\rho \vdash e_1 \hookrightarrow false \qquad \rho \vdash e_3 \hookrightarrow v}{\rho \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \hookrightarrow v}$$

$$fun : \rho \vdash (\text{fun } x \to e) \hookrightarrow [\text{fun } x \to e; \rho]$$

$$app : \frac{\rho \vdash e_1 \hookrightarrow [\text{fun } x \to e; \rho_1] \quad \rho \vdash e_2 \hookrightarrow v_2 \quad (x \mapsto v_2, \rho_1) \vdash e \hookrightarrow v}{\rho \vdash (e_1 \; e_2) \hookrightarrow v}$$

$$let : \frac{\rho \vdash e_1 \hookrightarrow v_1 \qquad (x \mapsto v_1, \rho) \vdash e_2 \hookrightarrow v}{let \; x = e_1 \; in \; e_2 \hookrightarrow v}$$

$$letrec : \frac{\rho' = (f \mapsto [\text{fun } x \to e_1, \rho'], \rho) \vdash e_2 \hookrightarrow v}{\text{let} \;\; \text{rec} \; f \; x = e_1 \text{ in } e_2 \hookrightarrow v}$$

Typage

$$var \; \frac{\Gamma(x) = t}{\Gamma \vdash x : t} \qquad tuple \; \frac{\Gamma \vdash e_1 : t_1 \quad \cdots \quad \Gamma \vdash e_n : t_n}{\Gamma \vdash (e_1, \cdots, e_n) : (t_1 * \cdots * t_n)}$$

$$if \; \frac{\Gamma \vdash e_1 : bool \quad \Gamma \vdash e_2 : t \quad \Gamma \vdash e_3 : t}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t}$$

$$fun \; \frac{(x : \; t_1), \Gamma \vdash e : t}{\Gamma \vdash \text{fun } x \to e \;\; : \;\; t_1 \to t}$$

$$app \; \frac{\Gamma \vdash e : t_1 \to t \qquad \Gamma \vdash e_1 : t_1}{\Gamma \vdash (e \; e_1) : t}$$

$$let \; \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2[e_1/x] : t}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : t}$$

$$letrec \; \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2[\text{Rec (fun } f \to e_1)/f] : t}{\Gamma \vdash \text{let} \;\; \text{rec } f = e_1 \text{ in } e_2 : t}$$

$$Rec : \; \Gamma \vdash Rec : ((t_1 \to t_2) \to (t_1 \to t_2)) \to (t_1 \to t_2)$$

Exemple:

$$\frac{\dfrac{\dfrac{(x : b \to b) \vdash x : b \to b}{\text{fun } x \to x : (b \to b) \to (b \to b)} \quad \dfrac{(x : b) \vdash x : b}{\text{fun } x \to x : b \to b}}{(\text{fun } x \to x \;\; \text{fun } x \to x) : b \to b}}{\text{let } f = \text{fun } x \to x \text{ in } (f \;\; f) : b \to b}$$

Typage avec shémas de type

types : t

shémas de type : $\forall \alpha_1 \cdots \alpha_n .\ t$

Ordre sur les types:

$t \leq \forall \alpha\ t$

$\forall \alpha_1 \cdots \alpha_{i-1}\ \alpha_{i+1} \cdots \alpha_n .\ t \leq \forall \alpha_1 \cdots \alpha_n .\ t$

Instanciation: $t \hookrightarrow t' \leq t$

Généralisation: $t \hookrightarrow \forall \alpha_1 \cdots \alpha_n .\ t$ où $\alpha_1 \cdots \alpha_n$ est l'ensemble des variables de t n'intervenant pas dans l'env.

$$var\ \frac{\Gamma(x) = \forall \alpha_1 \cdots \alpha_n .\ t \qquad t' \leq \forall \alpha_1 \cdots \alpha_n .\ t}{\Gamma \vdash x : t'}$$

$$let\ \frac{\Gamma \vdash e_1 : t_1 \quad (x : \forall \alpha_1 \cdots \alpha_n .\ t_1, \rho) \vdash e_2 : t}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : t} \quad \{\alpha_i\} = \mathrm{Var}(t_1) - \mathrm{Var}(\Gamma)$$

Exemple:

$$\dfrac{\dfrac{\dfrac{x : a \vdash x : a}{\vdash \text{fun } x \to x : a \to a}}{\vdash \text{fun } x \to x : \forall a.a \to a} \quad \dfrac{\dfrac{\Gamma \vdash f : \forall a.a \to a}{\Gamma \vdash f : (b \to b) \to (b \to b)} \quad \dfrac{\Gamma \vdash f : \forall a.a \to a}{\Gamma \vdash f : b \to b}}{\Gamma \vdash (f\ f) : b \to b}}{\vdash \text{let } f = \text{fun } x \to x \text{ in } (f\ f) : b \to b}$$

Preuves de correction de programmes

Raisonnement équationnel

Dans un système convergeant et au moins faiblement consistant, $e \hookrightarrow e' \Rightarrow Val(e) = Val(e')$.

Thm de substitution sur les valeurs:

$Val(e) = Val(e') \Rightarrow Val(e[v/x]) = Val(e'[v/x])$.

Extentionalité

Par définition: $(\forall x.\ f(x) = g(x)) \Rightarrow f = g$.

Problème: $0 = (\text{fun } x \to 0)\ (f\ 1)$?

Avec une stratégie incomplète S, $e = e'$ est interprété comme:

si $Val_S(e), Val_S(e')$ existe$_{nt}$ alors $Val(Val_S(e)) = Val(Val_S(e'))$

Raisonnement par cas

Exemples: $b = true \mid false$. $\quad n = 0 \mid S(m)$.

Induction

Ex. Sur les entiers:

- $P(0)$
- $\forall n.\ P(n) \Rightarrow P(S(n))$

Preuve de terminaison de programme

Évident, par induction. Sinon: ordre bien fondé.

Preuves de correction de programmes: Exemples

En étudiant la **sémantique** d'un langage, on peut réaliser des **preuves de propriétés sémantiques:**

– l'évaluation d'une expression donne une valeur,

– conservation des types (*Subject Reduction Theorem*),

– correction d'un compilateur, d'une traduction,

– ...

Ensuite, en étudiant à la fois un programme donné et la sémantique du langage dans lequel il est écrit, on peut réaliser des **preuves de correction de programmes**:

Preuve qu'un programme donné:

– termine,

– satisfait sa spécification,

– vérifie certains invariants à chaque pas d'itération,

– implémente une politique de sécurité donnée,

– ...

Aspects impératifs

Instructions dépendants

– d'une stratégie d'évaluation particulière (par valeur ici),

– d'une implémentation machine particulière.

Exceptions

Exemples: $n/0$, $hd[]$.
type $exn = Div_0 \mid Failure$ of $String \mid \cdots$;;
Le type exn peut être étendu par l'utilisateur.

Séquences

$$seq_e \; \frac{e_1 \hookrightarrow (expn\ v_1)}{(e_1; e_2) \hookrightarrow (expn\ v_1)} \qquad seq \; \frac{e_1 \hookrightarrow v_1 \quad e_2 \hookrightarrow v_2}{(e_1; e_2) \hookrightarrow v_2} \quad v_1 \neq (expn\ v)$$

Entrées/sorties

read, write, print: fonctions séquentielles

Structures de données modifiables (par affectation)

vecteurs, enregistrements (records), pointeurs, listes circulaires, listes doublement chaînées (prev, next), ...
Référence:
let $c =$ ref 0;; — c: int ref = ref 0
$c := !c + 3$;; — affectation, où $!c$ est la valeur de c.