École Normale Supérieure **Année scolaire 2006/2007**
Prédoctorat, spécialité « Informatique »

Cours « Langages de programmation et compilation »

Travaux dirigés — P. Cousot

TD 3

TD 3 – 1. Syntaxe concrète d'un langage fonctionnel

Le programme OCAML donné en annexe définit la syntaxe abstraite d'un langage fonctionnel du premier ordre (sans fonctions en paramètres) ainsi qu'un interpréteur `evalprog` du langage pour des programmes présentés sous cette forme abstraite. Les fichiers `print.mli`, `aintv.mli` et `aintv.ml` pour OCAML sont disponibles sur la toile à l'adresse :

`https://www.di.ens.fr/~cousot/cours/compilation/programmes_TD_fournis/`

Voici un exemple d'exécution de ce programme :

```
% ocaml

        Objective Caml version 3.08.1

# #load "print.cmo";;
# open Print;;
# #load "aintv.cmo";;

evalprog p      : execution du programme p (syntaxe abstraite) ;
trace_eval ()   : trace d'execution elementaire ;
untrace_eval () : pas de trace (par defaut).
# open Aintv;;
# (* example :                              *)
(* let                                     *)
(*   f(x, y)   = if x = 0 then y           *)
(*             els  e f(x - 1, y) fi       *)
(* in                                      *)
(*   f(2, -1)                              *)
  let pr0 =
([("f",([(FPVAL "x"); (FPVAL "y")],
              (IF
          ((EQUAL ((VAR "x"),(CST "0"))),
            ((VAR "y"),
           (CALL ("f",[(    SUB ((VAR "x"),(CST "1"))); (VAR "y")])))  ))))],
         (CALL ("f",[(CST "2"); (CST "-1")]))) ;;
val pr0 : (string * (Aintv.fpar list * Print.expr)) list * Print.expr =
  ([("f",
     ([FPVAL "x"; FPVAL "y"],
      IF (EQUAL (VAR "x", CST "0"),
       (VAR "y", CALL ("f", [SUB (VAR "x", CST "1"); VAR "y"])))))],
   CALL ("f", [CST "2"; CST "-1"]))
# evalprog pr0;;
-1
- : unit = ()
# ^D
```

%

Les fichiers `aintvex`i`.ml`, $i = 0, \ldots, 10$ sont disponibles sur la toile à l'adresse :

`https://www.di.ens.fr/~cousot/cours/compilation/programmes_TD_fournis`.

Ils contiennent d'autres exemples de programmes en forme abstraite.

Un programme est une suite de déclarations de fonctions, qui peut être vide, suivie d'une expression spécifiant la valeur retournée par l'exécution du programme. Les fonctions, en général mutuellement récursives, ont un corps formé d'une expression retournant une valeur (booléenne, entière, etc.). Outre les constantes entières, les variables désignant les valeurs des paramètres et les opérations arithmétiques binaires uselles (`+`, `-`, `*`, `/`), les expressions peuvent comporter des conditionnelles dont le test est une expression logique formée des constantes logiques `true` et `false`, des comparaisons d'égalité (`=`) et de stricte inégalité (`<`) de valeurs d'expressions arithmétiques et par combinaisons utilisant les opérateurs logiques de négation (`not`) et conjonction (`&`).

On commence par ajouter une syntaxe concrète au langage, dont voici quelques exemples :

```
1 + 1;;

let succ(x) = x + 1 in succ(0);;

let f() = 1; g(x, y) = if y = 0 then (x + f()) else y fi; in g(5, 0);;
```

On trouvera d'autres exemples de programmes concrets dans les fichiers `cintvex0.ml`, ..., `cintvex10.ml`disponibles sur la toile à l'adresse :

`https://www.di.ens.fr/~cousot/cours/compilation/programmes_TD_fournis`.

Question 3.1.1 *Concevoir une grammaire hors-contexte concrète du langage.*

Question 3.1.2 *Programmer en OCAML les fonctions (spécifiées dans l'interface* `print.mli`*) d'impression :*

`print_parval` *: d'un paramètre effectif par valeur ;*

`print_parameter_binding` *: d'une liaison de paramètre ;*

`print_bindings` *: d'une liste de liaisons de paramètres ;*

`print_activation` *: d'une activation de fonction (paire (nom de fonction appelée, liaison de paramètres) ;*

`print_environnement` *: d'un environnement ;*

`print_expr_list` *: d'une liste d'expressions (paramètres effectifs) ;*

`print_expr` *: d'une expression ;*

`print_expr_env` *: d'une paire (expression, environnement) ;*

`print_reseval` *: d'une paire (environnement, valeur).*

Les fonctions `print_expr_env` *et* `print_reseval` *sont utilisées par l'interpréteur* `aintv.ml` *pour faire des traces d'exécution.*

Question 3.1.3 *Programmer en OCAML l'impression* `print_prog` *d'une représentation abstraite d'un programme dans sa forme concrète. On pourra utiliser les boîtes de formattage de OCAML.*

Annexe : liste de l'interpréteur du langage fonctionnel

```
1  (* print.mli, Objective Caml version 3.08.1 *)
2
3  (*** valeurs a l'execution ***)
4  type value = ECST of string (* Erreur a l'execution  *)
5             | BCST of bool   (* Valeur booleenne   *)
6             | NCST of int ;; (* Valeur entiere *)
7  (* impression des valeurs a l'execution *)
8  val echo_val : value -> unit;;
9
10 (* expression *)
11 type expr = CST of string
12             | VAR of string
13             | ADD of expr * expr
14             | SUB of expr * expr
15             | MULT of expr * expr
16             | DIV of expr * expr
17             | LESS of expr * expr
18             | EQUAL of expr * expr
19             | NOT of expr
20             | AND of expr * expr
21             | IF of expr * (expr * expr)
22             | CALL of string * (expr list);;
23
24 (* impression d'une expression *)
25 val print_expr : expr -> unit ;;
26
27 (* liste d'expressions *)
28 type expr_list = expr list ;;
29
30 (* impression d'une liste d'expressions *)
31 val print_expr_list : expr_list -> unit ;;
32
33 (*** pile a l'execution ***)
34
35 (* parametre effectif par valeur *)
36 type parval = VALUE of value ;;
37 (* impression d'un parametre effectif par valeur *)
38 val print_parval : parval -> unit ;;
39
40 (* liaison de parametre : paire (nom du parametre formel, *)
41 (* valeur du parametre effectif)                          *)
42 type parameter_binding = (string * parval) ;;
43 (* impression d'une liaison de parametre *)
44 val print_parameter_binding : parameter_binding -> unit ;;
45
46 (* liaison de parametres : liste de paires (nom du parametre *)
47 (* formel, valeur du parametre effectif)                     *)
48 type bindings = parameter_binding list ;;
49 (* impression d'une liste de liaisons de parametres *)
50 val print_bindings : bindings -> unit ;;
```

```
51
52  (* activation (de fonction) = paire (nom de fonction appelee, *)
53  (* liaison de parametres)                                     *)
54  type activation = string * bindings ;;
55  (* impression d'une activation de fonction *)
56  val print_activation : activation -> unit ;;
57
58  (* environnement : liste d'activations *)
59  type environment = activation list ;;
60  (* impression d'un environnement *)
61  val print_environnement : environment -> unit ;;
62
63  (* impression d'une paire (expression, environnement) *)
64  val print_expr_env : expr * environment -> unit ;;
65
66  (* impression d'une paire (environnement, valeur) *)
67  val print_reseval : environment * value -> unit ;;
68
```

Interface `aintv.mli` en CAML

```
69  (* aintv.mli, Objective Caml version 3.08.1 *)
70
71  open Print;;
72
73  (*** erreurs a l'execution (detectables par un compilateur) ***)
74  exception Failure of string ;;
75
76  (*** syntaxe abstraite ***)
77
78  (* parametre formel par valeur *)
79  type fpar = FPVAL of string ;;
80
81  (* liste de parametres formels *)
82  type fpar_list = fpar list ;;
83
84  (* fonction anonyme : liste des parametres formel, corps *)
85  type anonymous_function = (fpar_list * expr) ;;
86
87  (* declaration de fonction :  nom fonction, fonction anonyme *)
88  type function_declaration  = (string * anonymous_function) ;;
89
90  (* liste de declarations de fonctions *)
91  type function_declaration_list = function_declaration list ;;
92
93  (* programme : liste de declarations de fonctions, corps *)
94  type prog = function_declaration_list * expr ;;
95
96  (* recherche de la liste de parametres formels et du corps *)
97  (* d'une fonction dans une liste de declarations           *)
98  type isdeclfun = FUNDECLARED of anonymous_function
```

```
99                  | FUNnotDECLARED ;;
100 val fparsbody :
101     string * function_declaration_list -> isdeclfun ;;
102
103 (*** pile a l'execution ***)
104
105 (* appel de fonction : empiler la liaison de parametres b sur *)
106 (* l'environnement r                                          *)
107 val push : activation * environment -> environment ;;
108
109 (* retour de fonction: depiler la liaison de parametres b de *)
110 (* l'environnement r                                         *)
111 val pull :
112     environment * activation -> environment * activation ;;
113
114 (* valeur d'un parametre formel dans la pile d'execution *)
115 type find = FOUND of parval | NotFOUND ;;
116
117 (* recherche dans une liaison de parametres *)
118 val val_in_bindings : activation -> find ;;
119
120 (* recherche dans la pile d'execution *)
121 val val_in_env : string * environment -> find ;;
122
123 (* affectation d'une valeur a un parametre formel dans la *)
124 (* pile d'execution                                       *)
125 type assign = ASSIGNED of bindings | NotASSIGNED of bindings ;;
126
127 (* affectation dans une liaison de parametres *)
128 val assign_in_bindings : string * value * bindings -> assign ;;
129
130 (* affectation dans la pile d'execution *)
131 val assign_env : string * value * environment -> environment ;;
132
133 (*** interpretation d'un programme ***)
134
135 (* trace d'execution elementaire *)
136 val istracing : bool ref  ;;
137 val trace_eval : unit -> unit ;;
138 val untrace_eval : unit -> unit ;;
139
140 (* impression du resultat de l'evaluation *)
141 val echo_result : environment * value -> unit ;;
142
143 (* evaluation du programme p *)
144 val evalprog : prog -> unit ;;
145
146 val message_initial : unit;;
```

Module `aintv.ml` en CAML

```
147 (* aintv.ml, Objective Caml version 3.08.1 *)
```

```
148
149  open Print;;
150
151  (*** erreurs a l'execution (detectables par un compilateur) ***)
152  exception Failure of string ;;
153
154  (*** syntaxe abstraite ***)
155
156  (* parametre formel par valeur *)
157  type fpar = FPVAL of string ;;
158
159  (* liste de parametres formels *)
160  type fpar_list = fpar list ;;
161
162  (* fonction anonyme : liste des parametres formel, corps *)
163  type anonymous_function = (fpar_list * expr) ;;
164
165  (* declaration de fonction :  nom fonction, fonction anonyme *)
166  type function_declaration  = (string * anonymous_function) ;;
167
168  (* liste de declarations de fonctions *)
169  type function_declaration_list = function_declaration list ;;
170
171  (* programme : liste de declarations de fonctions, corps *)
172  type prog = function_declaration_list * expr ;;
173
174  (* recherche de la liste de parametres formels et du corps *)
175  (* d'une fonction dans une liste de declarations           *)
176  type isdeclfun = FUNDECLARED of anonymous_function
177                 | FUNnotDECLARED ;;
178
179  (* valeur d'un parametre formel dans la pile d'execution *)
180  type find = FOUND of parval | NotFOUND ;;
181
182  (* affectation d'une valeur a un parametre formel dans la *)
183  (* pile d'execution                                       *)
184  type assign = ASSIGNED of bindings | NotASSIGNED of bindings ;;
185
186  (* recherche de la liste de parametres formels et du corps *)
187  (* d'une fonction dans une liste de declarations           *)
188  let rec fparsbody =
189    function (f, (fj, h) :: fds) -> if f = fj then FUNDECLARED(h)
190                                            else fparsbody(f, fds)
191     | (f, [])              -> FUNnotDECLARED ;;
192
193  (*** interpretation des fonctions primitives ***)
194
195  let iadd =
196   function ((NCST n1), (NCST n2)) -> (NCST (n1 + n2))
197    |  ((ECST s), e)          -> (ECST s)
198    |  (e, (ECST s))          -> (ECST s)
199    |  (e1, e2)               -> (ECST "addition of non-integer values")
```

```
200 ;;
201
202 let isub =
203  function ((NCST n1), (NCST n2)) -> (NCST (n1 - n2))
204   |  ((ECST s), e)          -> (ECST s)
205   |  (e, (ECST s))          -> (ECST s)
206   |  (e1, e2)               -> (ECST "subtraction of non-integer values")
207 ;;
208
209 let imult =
210  function ((NCST n1), (NCST n2)) -> (NCST (n1 * n2))
211   |  ((ECST s), e)          -> (ECST s)
212   |  (e, (ECST s))          -> (ECST s)
213   |  (e1, e2)               -> (ECST "multiplication of non-integer values")
214 ;;
215
216 let idiv =
217  function ((NCST n1), (NCST n2)) -> (NCST (n1 / n2))
218   |  ((ECST s), e)          -> (ECST s)
219   |  (e, (ECST s))          -> (ECST s)
220   |  (e1, e2)               -> (ECST "division of non-integer values")
221 ;;
222
223 let iless =
224  function ((NCST n1), (NCST n2)) -> (BCST (n1 < n2))
225   |  ((ECST s), e)          -> (ECST s)
226   |  (e, (ECST s))          -> (ECST s)
227   |  (e1, e2)               -> (ECST "inequality of non-integer values")
228 ;;
229
230 let iequal =
231  function ((NCST n1), (NCST n2)) -> (BCST (n1 = n2))
232   |  ((ECST s), e)          -> (ECST s)
233   |  (e, (ECST s))          -> (ECST s)
234   |  (e1, e2)               -> (ECST "equality of non-integer values")
235 ;;
236
237 let inot =
238  function (BCST true)  -> (BCST false)
239   |  (BCST false) -> (BCST true)
240   |  (ECST s)     -> (ECST s)
241   |  e            -> (ECST "negation of non-boolean values")
242 ;;
243
244 let iand =
245  function ((BCST b1), (BCST b2)) -> (BCST (b1 & b2))
246   |  ((ECST s), e)          -> (ECST s)
247   |  (e, (ECST s))          -> (ECST s)
248   |  (e1, e2)               -> (ECST "conjunction of non-boolean values")
249 ;;
250
251 (*** pile a l'execution ***)
```

```
252
253  (* appel de fonction : empiler la liaison de parametres b sur *)
254  (* l'environnement r                                          *)
255  let push =
256   function (b,  r) -> b :: r ;;
257
258  (* retour de fonction: depiler la liaison de parametres b de *)
259  (* l'environnement r                                         *)
260  let pull =
261   function (b :: r, v) -> (r, v)
262    |  _           -> raise (Failure "pull") ;;
263
264  (* recherche dans une liaison de parametres *)
265  let rec val_in_bindings =
266   function (x, (y, v) :: b) -> if x = y then (FOUND v)
267                                else (val_in_bindings (x, b))
268    |  (x, [])          -> NotFOUND ;;
269
270  (* recherche dans la pile d'execution *)
271  let rec val_in_env =
272   function (x, (f, b) :: r) -> (match val_in_bindings (x, b) with
273                                  (FOUND v) -> (FOUND v)
274                                |   NotFOUND  -> (val_in_env (x, r))
275                             )
276    |  (x, [])          -> NotFOUND ;;
277
278  (* affectation dans une liaison de parametres *)
279  let rec assign_in_bindings =
280   function (x, v, ((y, p) :: b))
281         -> if x = y then
282              (ASSIGNED ((x, (VALUE v)) :: b))
283            else
284              (match assign_in_bindings (x, v, b) with
285                (ASSIGNED bm)    -> (ASSIGNED ((y, p) :: bm))
286              | (NotASSIGNED bm) -> (NotASSIGNED ((y, p) :: bm))
287              )
288   |   (x, v, [])
289         -> (NotASSIGNED []);;
290
291  (* affectation dans la pile d'execution *)
292  let rec assign_env =
293   function (x, v, (f, b) :: r)
294         -> (match assign_in_bindings (x, v, b) with
295              (ASSIGNED bm)    -> (f, bm) :: r
296            | (NotASSIGNED bm) -> (f, bm) :: (assign_env (x, v, r))
297            )
298    |  (x, v, [])
299         -> [("main",
300              [(x, (VALUE (ECST "variable affectee mais non declaree")))])];;
301
302  (*** interpretation d'un programme ***)
303
```

```
304  (* trace d'execution elementaire *)
305  let istracing = ref false ;;
306  let trace_eval () = istracing := true ;;
307  let untrace_eval () = istracing := false ;;
308
309  (* impression du resultat de l'evaluation *)
310  let echo_result re =
311    match re with
312      ([], v) -> echo_val v; print_string "\n"
313    | _       -> print_string "\n echo_result: unexpected result \n" ;;
314
315  open Print;;
316
317  (* evaluation du programme p *)
318  let evalprog = function (fds, ep) ->
319   let rec
320    (* (bind ((fps, aps), r)) lier les parametres formels fps *)
321    (* aux parametres effectifs aps dans l'environnement r.   *)
322    bind =
323        function ((((FPVAL x) :: fps), (e :: aps)), r)
324                -> if !istracing then
325                    (print_string ("\nbind == "^x); print_newline ()
326                    );
327                    let (r', v') = (eval (e, r)) in
328                     let (r'', b) = (bind ((fps, aps), r')) in
329                       (r'', ((x, (VALUE v')) :: b))
330          |  (([], []), r)  ->  (r, [])
331          |  (([], aps), r) -> raise (Failure "too many actual parameters")
332          |  ((fps, []), r) -> raise (Failure "too few actual parameters")
333   and
334    (* eval (e, r) evalue l'expression e dans l'environnement r et *)
335    (* retourne  (r', v),  ou r' est l'environnement modifie et v  *)
336    (* est la valeur de e                                          *)
337    eval c =
338      if !istracing then
339          (print_string "\neval ===> \n";
340           print_expr_env c
341          );
342      let reseval =
343        match c with
344            ((CST "true"), r)  -> (r, (BCST true))
345          |  ((CST "false"), r) -> (r, (BCST false))
346          |  ((CST n), r) -> (r, (try (NCST (int_of_string n)) with
347                                   Failure s -> (ECST "boolean or integer required")
348                                  ))
349          |  ((VAR x), r) ->
350               (match val_in_env(x, r) with
351                  (FOUND (VALUE v)) -> (r, v)
352               |  NotFOUND -> (r, (ECST ("variable " ^ x ^ " not declared")))
353               )
354          |  ((ADD (e1, e2)), r) ->
355               let (r1, v1) = (eval (e1, r)) in
```

```
356                 let (r2, v2) = (eval (e2, r1)) in
357                   (r2, iadd(v1, v2))
358           |  ((SUB (e1, e2)), r) ->
359                let (r1, v1) = (eval (e1, r)) in
360                  let (r2, v2) = (eval (e2, r1)) in
361                    (r2, isub(v1, v2))
362           |  ((MULT (e1, e2)), r) ->
363                let (r1, v1) = (eval (e1, r)) in
364                  let (r2, v2) = (eval (e2, r1)) in
365                    (r2, imult(v1, v2))
366           |  ((DIV (e1, e2)), r) ->
367                let (r1, v1) = (eval (e1, r)) in
368                  let (r2, v2) = (eval (e2, r1)) in
369                    (r2, idiv(v1, v2))
370           |  ((LESS (e1, e2)), r) ->
371                let (r1, v1) = (eval (e1, r)) in
372                  let (r2, v2) = (eval (e2, r1)) in
373                    (r2, iless(v1, v2))
374           |  ((EQUAL (e1, e2)), r) ->
375                let (r1, v1) = (eval (e1, r)) in
376                  let (r2, v2) = (eval (e2, r1)) in
377                    (r2, iequal(v1, v2))
378           |  ((NOT e), r) ->
379                let (rp, v) = (eval (e, r)) in
380                  (rp, inot(v))
381           |  ((AND (e1, e2)), r) ->
382                let (r1, v1) = (eval (e1, r)) in
383                  let (r2, v2) = (eval (e2, r1)) in
384                    (r2, iand(v1, v2))
385           | ((IF (e1, (e2, e3))), r) ->
386                (match (eval (e1, r)) with
387                  (r1, (BCST b)) -> if b then (eval (e2, r1))
388                                    else (eval (e3, r1))
389                | (r1, (NCST n)) -> (r1, (ECST "integer result in a test"))
390                | (r1, (ECST s)) -> (r1, (ECST s))
391                )
392           | ((CALL (f, aps)), r) ->
393                (match (fparsbody (f, fds)) with
394                  (FUNDECLARED (fps, bf)) ->
395                    let (r', b) = (bind ((fps, aps), r)) in
396                    (pull (eval (bf, (push ((f, b), r')))) ) )
397                | FUNnotDECLARED ->
398                    (r, (ECST ("function " ^ f ^ " not declared")))
399                )
400       in
401         if !istracing then
402           (print_string "\n<=== \n";
403            print_reseval reseval
404           );
405         reseval
406     in
407        (echo_result (eval (ep, []))) ;;
```

```
408
409  let message_initial =
410  print_newline ();
411  print_string "evalprog p      : execution du programme p (syntaxe abstraite) ;";
412  print_newline ();
413  print_string "trace_eval ()   : trace d'execution elementaire ;";
414  print_newline ();
415  print_string "untrace_eval () : pas de trace (par defaut).";
416  print_newline ();;
```