Contrôles : signaux, exceptions, continuations

Cours de Compilation Avancée (MU4IN504)

Benjamin Canou & Emmanuel Chailloux
Sorbonne Université

Année 2019/2020 – Semaine 5

Plan du cours

Contrôle de haut niveau:

- Signaux : ruptures de calcul et reprise immédiate
- Exceptions : ruptures de calcul
- Continuation : rupture ET reprise de calcul (GOTO fonctionnel)

Ruptures de calcul : signaux

signaux en C:

- Les signaux en C permettent d'informer un processus qu'un événement particulier s'est produit.
- Chaque processus peut gérer un traitement particulier pour un signal donné.
- Le traitement lié à un signal associe à un signal une fonction de traitement.
- Lors du déclenchement de cet événement le programme exécute la fonction associée puis reprend l'exécution du programme là où il l'avait laissé.

différences importantes entre BSD et SYSTEM 5,
voir le fichier `<signal.h>`

Exemple 1

- Le signal SIGINT indique une interruption (`kill -2` ou `^C`) du processus. Poser un récupérateur pour le signal SIGINT qui affiche le message `"Pas d'interruption possible"` en cas d'arrivée de ce signal.

```
1  #include <signal.h>
2  void hand_int(int sig){
3   printf("Pas d'interruption ^C active\n");
4   printf("Taper ^\\ si vous d\'esirez sortir du programme\n");
5   signal(SIGINT,hand_int);
6  }
7  main () { int i;
8   int c=0;
9   signal(SIGINT,hand_int);
10  printf("ça boucle,  essayer ^C pour l'arreter\n");
11  while (1) {
12    i++;
13    if ((i % 100000) == 0) {c++;printf("."); fflush(stdout);}
14  }
15 }
```

Exemple 2 - (1)

```
1  #include <signal.h>
2  #include <stdio.h>
3
4  struct cons {int car; struct cons *cdr;};
5  typedef struct cons *liste_entier;
6
7  liste_entier cons(int a, liste_entier l){
8   liste_entier r;
9   r=(liste_entier)(malloc(sizeof(struct cons)));
10  r->car=a; r->cdr=l; return r;
11 }
12
13 liste_entier intervalle(int a, int b){
14   if (a > b) return NULL;
15   else return cons(a,intervalle(a+1,b));
16 }
17
18 int compteur=0;
```

Signaux : exemple 2 - (2)

```
1  int compte(liste_entier l){
2    sleep(1);
3    if (l == NULL) return 0;
4    else {
5      compteur++;
6      return 1 + compte(l->cdr);
7    }
8  }
9  void hand(){
10   printf(" deja compt\'es %d\n",compteur);
11   signal(SIGINT,hand);
12 }
13 main(int argc, char *argv[]){
14  int r;
15  int x;
16  if (argc == 1) fprintf(stderr,"pas d'arguments\n");
17  else {
18    signal(SIGINT,hand);
19    x = atoi(argv[1]);
20    r = compte(intervalle(1,x));
21    printf("%d elements\n",r);
22  }
23 }
```

Typage et domaine de définition

type inféré $\neq$ domaine de définition:

- c'est une approximation
- exemple : division entière, tête de liste vide
- provient souvent d'un filtrage non exhaustif

Que faire?:

- utiliser une valeur spéciale

```
# asin 2.;;
- : float =          NaN
```

- effectuer une rupture de calcul jusqu'à un récupérateur (exceptions)

Exceptions

Exceptions

Deux visions des ruptures de calcul :

1. Erreurs ou ruptures normales, style de programmation
2. Erreurs graves, doivent arriver rarement

Exceptions en OCaml

Exemple basique : erreurs fatales

- Erreur de langage :

```
# let tab = [| 0 ; 25 ; 2 |] ;;
val tab : int array = [|0; 25; 2|]
# tab.(12) ;;
Exception: Invalid_argument "index out of bounds".
# List.map2 (fun x y -> x + y) [ 1 ] [ 1 ; 2 ];;
Exception: Invalid_argument "List.map2".
```

- Erreur d'exécution :

```
# let rec f () = f () + f () ;;
val f : unit -> int = <fun>
# f () ;;
Stack overflow during evaluation (looping recursion?).
# Array.make 2_000_000_000 0 ;;
Out of memory during evaluation.
```

Exceptions en OCaml

Erreurs locales

Exemple : commande `head`

```
let head fname nlines =
  let fp = open_in fname in
  try
    for i = 1 to nlines do
      printf "%03d: %s\n%!" i (input_line fp)
    done ;
    close_in fp
  with
    End_of_file ->
      fprintf stderr "Not enough lines." ;
      close_in fp
```

Exceptions en OCaml

Style de programmation : ruptures locales

```
let search tab v =
  try
    for i = 0 to Array.length tab - 1 do
      if tab.(i) = v then
        raise Exit
    done ;
    false
  with
    Exit -> true
```

Exceptions en OCaml

Style de programmation : ruptures de récursion

```
let prod_liste l =
  let rec aux = function
    | [] -> 1
    | 0 :: _ -> raise Exit
    | hd :: tl -> hd * aux tl
  in
    try
      aux l
    with Exit -> 0
```

Exceptions en OCaml

Typage

- Langage fonctionnel + polymorphisme paramétrique
  - difficile de connaitre les exceptions lancées par une expr
  - type somme monomorphe extensible `exn`
  - filtrage sur toutes les exceptions du programme

```
# exception MonEx of string ;;
exception MonEx of string
# MonEx "bob" ;;
- : exn = MonEx "bob"
```

- Type d'une rupture de calcul : sans importance

```
raise ;;
- : exn -> 'a = <fun>
```

Exceptions en Java

Définition d'une exception

```
class MonEx extends Exception {
  public MonEx(String s) {
    super ("MON EX <" + s + ">");
  }
}
```

Exceptions en Java

Lancement :

```
public void break () throws MonEx {
  throw new MonEx ("broken") ;
}
```

Exceptions en Java

Récupération :

```
try {
  /* ... */
} catch (MonEx e) {
  /* ... */
} catch (Exception e) {
  /* ... */
} finally {
  /* ... */
}
```

Exceptions en Java

Typage

- Toute exception dérive de `Throwable`
  - `Error` : grave
  - `Exception` : moins grave
  - `RuntimeException` : aïe
- Hiérarchie de classes
  `A'` dérive de `A` $\Rightarrow$ `catch(A)` récupère aussi `A'`
- `throw` élimine la nécessité d'un return dans une branche
- On peut (et doit) indiquer les exceptions avec `throws`

Implantation des exceptions

Deux visions :

- Java :
  - la création coûte chère (stockage de la pile d'appel des méthodes)
  - le lancement peut aussi coûter cher (selon l'imbrication)
  - la récupération quasiment pas (au test de type près)
- OCaml :
  - la création coûte la construction classique d'une valeur d'un type somme
  - le lancement (remontée directe au dernier récupérateur) et la pose de récupérateurs coûtent, mais pas trop trop
  - la récupération utilise le mécanisme de filtrage

Implantation des exceptions

Java : compilation d'un récupérateur

Pas d'instruction bytecode pour `try` et `catch`.
On enregistre la portée de chaque récupérateur à côté.

```
Code_attribute {
  /* extrait de la spécif du format .class */
  u4 code_length;
  u1 code[code_length];
  u2 exception_table_length;
  { u2 start_pc;
    u2 end_pc;
    u2 handler_pc;
    u2 catch_type;
  } exception_table[exception_table_length];
}
```

$\Rightarrow$ Pour chaque instruction, on peut savoir le récupérateur associé.

Implantation des exceptions

Java : compilation d'une rupture

Instruction `athrow` du bytecode :

1. On récupère les récupérateurs associés à la méthode courante.
2. On lance le premier récupérateur dont le type est compatible.
3. Si aucun n'est compatible,
   - 3.1 on remonte d'un cran dans la pile d'appels,
   - 3.2 on réitère.

On parcourt toute la pile.

Implantation des exceptions

En OCaml

On stocke chaque récupérateur sur la pile :

1. pointeur de code du récupérateur,
2. chaînage vers le rattrapeur précédent sur la pile.

On conserve un pointeur vers le dernier récupérateur

Pour lancer une exception :

1. on restaure la pile à l'endroit du dernier récupérateur,
2. on restaure le récupérateur précédent comme dernier récupérateur,
3. on appelle le code du récupérateur.

Etiquettes dynamiques en C

Pour certaines applications l'exécution du programme doit être reprise en un point particulier.

- Pour cela il est nécessaire d'utiliser des étiquettes dynamiques.
- On mémorise l'état de la pile d'exécution du processus dans une variable de type `jmp_buf` défini dans `<setjmp.h>` avec la fonction `setjmp`.
- Cette variable constitue l'étiquette. La fonction `longjmp` permet ensuite de reprendre l'exécution au point prévu en restaurant l'état de la pile.

```
1 int setjmp(jmp_buf env);
2 void longjmp (jmp_buf env, int val);
```

Exemple : produit d'une liste

```
1  int mult_liste_v2(jmp_buf env,liste_entier l) {
2    compteur++;
3    if (l == NULL) return 1;
4    else {
5      if (l->car == 0) longjmp(env,1);
6      else return l->car*mult_liste_v2(env,l->cdr); }
7  }
8  int mult_liste(liste_entier l) {
9    int r;
10   jmp_buf env;
11   compteur=0;
12   switch(setjmp(env)){
13     case 0: { r=mult_liste_v2(env,l);
14           printf("calcul en %d etapes : ",compteur);
15           return r;
16         }
17     case 1: { printf("un zero rencontr\'e \`a l'\'etape %d : "↩
           ,compteur);
18           return 0;
19         }
20     default: {fprintf(stderr,"cas incroyable\n");exit(0);}
21   }
22 }
```

Exceptions en C (1)

Le mécanisme d'étiquettes dynamiques permet de construire un mécanisme d'exceptions (comme en ML).

```
1  include "exception.h"
2
3  exception e;
4
5  Test()
6  {
7    TRY
8      Body();
9    EXCEPT(e)
10     Handler();
11   ENDTRY
12 }
```

avec le déclenchement d'une exception par :

```
1  RAISE(e,v)
```

où e est une exception et v un entier.

Exceptions en C (2)

```
1  #include <setjmp.h>
2  typedef  char * exception;
3
4  typedef struct _ctx_block {
5    jmp_buf env;
6    exception exn;
7    int val;
8    int state;
9    int found;
10   struct _ctx_block *next;
11 } context_block;
12
13 #define  ES_EvalBody 0
14 #define  ES_Exception 1
15
16 extern exception ANY;
17 extern context_block *exceptionStack;
18 extern void _RaiseException();
19
20 #define RAISE(e,v) _RaiseException(&e,v)
```

Exceptions en C (3)

```
1  #define TRY \
2   {\
3    context_block _cb;\
4    int state = ES_EvalBody;\
5    _cb.next=exceptionStack;\
6    _cb.found=0;\
7    exceptionStack=&_cb;\
8    if (setjmp(_cb.env) != 0) state=ES_Exception;\
9    while(1){\
10     if (state == ES_EvalBody){
11
12 #define EXCEPT(e)\
13   if (state == ES_EvalBody) \
14    exceptionStack=exceptionStack->next;\
15   break;\
16  }\
17   if (state == ES_Exception) \
18    if ((_cb.exn == (exception)&e) || (&e == &ANY)) {\
19      int exception_value = _cb.val;\
20      _cb.found=1;
```

Exceptions en C (4)

```
1
2  #define ENDTRY \
3   }\
4   if (state == ES_EvalBody) {
5    exceptionStack=exceptionStack->next;
6    break;}\
7   else {exceptionStack=exceptionStack->next;\
8    if (_cb.found == 0) _RaiseException(_cb.exn,_cb.val);\
9    else break;}\
10    }\
11 }
```

référence : Eric S. Robert, "Implementing Exceptions in C", Rapport de recherche SRC-40, Digital Equipment, 1989.

Exceptions en C : fichier C (5)

```c
#include <stdio.h>
#include "exception.h"

context_block *exceptionStack = NULL;

exception ANY;

void _RaiseException(exception e, int v)
{
  if (exceptionStack == NULL)
  {fprintf(stderr,"Uncaught exception\n");
   exit(0);
  }
  else {
   exceptionStack->val=v;
   exceptionStack->exn=e;
   longjmp(exceptionStack->env,ES_Exception);
  }
}
```

Exceptions en C : exemple

```
1  #include "exception.h"
2  exception Found_zero;
3  int mult_liste_v3(liste_entier l){
4   compteur++;
5   if (l == NULL) return 1;
6   else { if (l->car == 0) RAISE(Found_zero,1);
7          else return l->car*mult_liste_v3(l->cdr);
8        }
9  }
10 int mult_liste(liste_entier l){
11   int r;  compteur=0;
12   TRY
13     { printf("calcul en %d etapes : ",compteur);
14       r=mult_liste_v3(l);
15     }
16   EXCEPT(Found_zero)
17     { printf("un zero rencontr\'e \`a l'\'etape %d : ",↩
           compteur);
18       r=0;
19     }
20   ENDTRY
21   return r;
22 }
```

Continuations

Une continuation est la représentation d'un contexte de calcul sous la forme d'une fonction.

- utilisées pour décrire les ruptures de contrôle dans les formalismes de définition de la sémantique des langages de programmation.
- popularisées par le langage Scheme
- à la base d'un style de programmation appelé CPS (*Continuation Passing Style*) qui permet d'expliciter le contrôle.

Style CPS

La transformation d'une fonction en style CPS se fait en lui ajoutant un argument supplémentaire (la continuation initiale), et en explicitant sous la forme d'une continuation le contexte d'évaluation de chaque calcul intermédiaire.
la fonction fib :

```
1 let rec fib(n) =
2   if n <= 1 then 1 else
3   fib(n-1) + fib(n-2)
```

devient

```
1 let rec fib(n)(cont) =
2  if n <= 1 then cont(1) else
3  fib(n-1)
4     (fun m1 ->
5        fib(n-2) (fun m2 -> cont(m1+m2)))
```

et les appels

```
1 fib(5)(fun x -> x);;
2 fib(5)(print);;
```

Transformation CPS (1)

Soit une fonction `count` qui prend un prédicat et une liste et retourne le nombre d'éléments de la liste qui vérifient ce prédicat, et une fermeture `countZeros` qui correspond à l'application partielle de `count` à un prédicat qui retourne `true` si son argument vaut 0, et `false` sinon.

```
# let count pred =
   let rec f a =
    if null a  then 0
    else if pred(hd a) then 1 + f(tl a)
         else f (tl a)
   in f ;;
val count : ('a -> bool) -> 'a list -> int = <fun>

# let countZeros =
    count (function i -> if i=0 then true else false) ;;
val countZeros : int list -> int = <fun>
```

La transformation modifie chaque fonction en lui rajoutant un argument supplémentaire (la continuation) et revient de la fonction en appelant la continuation :

Transformation CPS (2)

```
# let count (pred,c1) =
   let rec f (a,c2) =
    if null a
    then c2(0)
    else
     let c3 b =
       if b
       then
         let  c4(i) =c2(1+i)
         in
           f(tl(a),c4)
       else
         let c5(i)=c2(i)
         in f(tl(a),c5)
     in
       pred((hd a),c3)
  in c1(f) ;;
val count :
  ('a * (bool -> 'b) -> 'b) * (('a list * (int -> 'b) -> 'b) ↩
      -> 'c) -> 'c = <fun>
```

Le contrôle de la fonction count est devenu explicite. Ce résultat se prête à plusieurs optimisations. Ici la continuation c5 est inutile car identique à c2 et peut donc être supprimée.

Transformation CPS (3)

La variable countZeros devient :

```
1  let countZeros m = let g i = if i=0 then true else false in ↩
       count g m
2  ;;
```

et peut donc se réécrire ainsi :

```
1  let countZeros (m,c0) =
2    let g (i,c1) =
3      if i=0
4      then c1(true)
5      else c1(false)
6    in
7      let c3(a) = a
8      in
9        let c2(a) = (a (m,c3))
10       in
11         c0(count(g,c2))
12 ;;
```

Continuation courante (1)

Certains langages (Scheme, SML/NJ) fournissent une primitive permettant de capturer le contexte courant d'évaluation sous la forme d'une fonction appelée la *continuation courante*, de la lier à un identificateur, et de l'utiliser si nécessaire.

- Cette continuation, lorsqu'elle est utilisée dans un certain contexte, ignore ce contexte et restaure le contexte qu'elle représente.

Continuation courante (2)

- En Scheme, la capture de la continuation courante est effectuée par la primitive `call-with-current-continuation`, ou, plus simplement, `call/cc`, qui reçoit une fonction à un paramètre (`fun k -> b`) et qui l'applique à la continuation courante.
- Le contrôle est donc passé au corps `b` de la fonction, où la continuation capturée est disponible sous le nom `k` du paramètre formel.

Ainsi, la valeur de:

```
1 call/cc
2  (fun throw ->  f (if p then throw(0) else ...))
```

sera 0 si `p` vaut `true`

En Scheme : call/cc (1)

Exemple : produit des éléments d'une liste (classique)

```
1  (define (list_mult_aux return l)
2    (letrec ((r (lambda (l) (print l)
3                  (if (null? l) 1
4                  (if (= (car l) 0) (return 0)
5                   (print(* (car l) (r (cdr l))))))))))
6        (r l)))
7
8  (define (list_mult l)
9     (call/cc (lambda (c) ((list_mult_aux c l)))))
10
11 (list_mult '( 1 2 3))
```

En Scheme : call/cc (2)

Exemple : produit des éléments d'une liste (classique)

```
(list_mult '( 1 2 3 0 4 5))

(1 2 3 0 4 5)
(2 3 0 4 5)
(3 0 4 5)
(0 4 5)
0
```

En Scheme : call/cc (3)

Exemple : produit des éléments d'une liste (call/cc)

```
1  (define (main x)
2      (+ (call/cc (lambda (k0) (f1 k0 x))) 3) )
3
4  (define (f1 k0 x)
5      (+ (call/cc (lambda (k1) (f2 k0 k1 x))) 2) )
6
7  (define (f2 k0 k1  x)
8      (+ (call/cc (lambda (k2) (f3 k0 k1 k2 x))) 1) )
9
10 (define (f3 k0 k1 k2 x)
11    (cond ((< x 10) (k2 x))
12          ((< x 100) (k1 x))
13          (#t  (k0 x))))
```

En Scheme : call/cc (4)

Exemple : produit des éléments d'une liste (call/cc)

```
(main 0)
6

(main 10)
15

(main 100)
103
```

Le call/cc typé (1)

- Exemple 1 :

```
1       (print x);  (* paramtre inutile (unit) *)
2  ---> point de calcul
3       ...
```

- Exemple 2 :

```
1  (3+4*2)      parametre indispensable
2       |
3       |
4   point de calcul
```

On voit bien que sans paramètre pour remplacer la valeur 2 on ne peut pas continuer le calcul

Le call/cc typé (2)

- Le call/cc applique à une fonction passée en paramètre la continuation courante.
- Le type $((' a \Rightarrow' a) \Rightarrow' a) \Rightarrow' a$
- Exemple :

```
1 (3+4*(call_cc
2     (fun k ->
3        10*
4        (if read_bool()
5         then (k 3)
6         else 4))))
```

Implantation du call/cc (1)

- Implémentation classique : le CPS
  - Solution élégante et « rapide »
  - Ralentit les programmes qui ne l'utilise pas
  - Interfaçage avec C difficile

référence :

- Juliusz Chroboczek. Continuation Passing for C: A space-efficient implementation of concurrency
- Gabriel Kerneis : Continuation-Passing C : transformations de programmes pour compiler la concurrence dans un langage impératif

Implantation du call/cc (2)

Machines à pile: :

- Qu'est-ce qu'un point de calcul ?
  - Un contexte d'éxécution
  - Une copie de pile
  - Un tas partagé
- Implémentation lourde
  - Interfaçage avec C plus facile
  - Coût plus «juste».
  - Pb de vitesse

Le call/cc utilisation

- Exception (sous cas du call/cc)
- Interprète : Luc Moreau, Daniel Ribbens, and Pascal Gribomont. Advanced Programming Techniques Using Scheme. In Journées Francophones des Languages Applicatifs
- Nagivateur : Christian Queinnec, The Influence of Browsers on Evaluators or, Continuations to Program Web Servers,icfp2000
- Programmes concurrents : Luc Moreau. Continuing into the Future: the Return, InterSymp'96, Luc Moreau. The Semantics of Scheme with Future.ICFP'96.
- Migrations de calcul: Chailloux - Ravet - Verlaguet : hirondML : http://www.algo-prog.info/hirondml

Conclusion : 1ère partie

Plan du cours:

- Cours 1 : Rappels, analyseurs
- Cours 2 et 3 : Machines virtuelles et bibliothèques d'exécution
- Cours 4 : Modèles mémoire
- Cours 5 : Contrôle de haut niveau : signaux, exceptions, continuations

Que manque-t-il :

- JIT : just in time (expansion de code)
  - JIT compilation of OCaml byte-cde : `https://arxiv.org/pdf/1011.6223.pdf`, `https://arxiv.org/pdf/1011.1783.pdf`
  - chargement dynamique de code
  - concurrence (vmthreads)

Travail à faire : 1ère partie

Projet: GC pour une mini-ZAM

- réalisation d'un récupérateur de mémoire pour une machine virtuelle fonctionnelle (comme pour 3I018) utilisant plusieurs algorithmes :
  - S&C,
  - M&S,
  - générationnel
- contraintes
  - à écrire en C
  - à faire en binôme
  - à rendre avant le 25 mars 2020
- miniZAM : `https://www-apr.lip6.fr/~varoumas/ca/projet.pdf`
- GC : `https://www-apr.lip6.fr/~chaillou/Public/enseignement/2019-2020/ca/`