Controle de haut niveau
Concurrence et parallélisme
Cours de Compilation Avancée (MI190)

Benjamin Canou
Université Pierre et Maire Curie

Année 2010/2011 – Semaine 5

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

Rattrapage :

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

Exceptions en Java

Typage

- Toute exception dérive de `Throwable`
  - `Error` : grave
  - `Exception` : mons grave
  - `RuntimeException` : aïe
- Hiérarchie de classes
  `A'` dérive de `A` $\Rightarrow$ `catch(A)` rattrappe 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 :
  seul le lancement coûte, mais il coûte cher
- OCaml :
  le lancement et la pose de rattrapeurs coûtent, mais pas trop

Implantation des exceptions

Java : compilation d'un rattrapeur

Pas d'instruction bytecode pour `try` et `catch`.
On enregistre la portée de chaque rattrapeur 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 rattrappeur associé.

Implantation des exceptions

Java : compilation d'une rupture

Instruction `athrow` du bytecode :

1. On récupère les rattrapeurs associés à la méthode courante.
2. On lance le premier rattrappeur 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 rattrapeur sur la pile :

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

On conserve un pointeur vers le dernier rattrapeur

Pour lancer une exception :

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

Étiquettes dynamiques en C

Mécanisme `setjmp/longjmp`, extension du `goto` :

- `setjmp` enregistre le niveau de pile, le pointeur de code et les registres,
- `longjmp` le restaure, l'exécution reprend à l'appel à `setjmp` correspondant,
- la valeur de retour de setjmp indique si c'est le premier appel, ou le résultat d'un retour via `longjmp`.

```
if (setjmp(jmpbuf)) {
  /* retour via longjump */
} else {
  /* première exécution */
}
```

Exceptions en C

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

```
#include "exception.h"
exception MonEx;

void test () {
  TRY
    /* ... */
  EXCEPT(MonEx)
    /* ... */
  ENDTRY
}
```

Exceptions en C : implantation (1)

Fichier H :

```
#include <setjmp.h>

typedef char * exception;
typedef struct _ctx_block {
  jmp_buf env;
  exception exn;
  int val;
  int state;
  int found;
  struct _ctx_block *next;
} context_block;

#define ES_EvalBody 0
#define ES_Exception 1

extern exception ANY;
extern context_block *exceptionStack;
extern void _RaiseException();

#define RAISE(e,v) _RaiseException(&e,v)
```

Exceptions en C : implantation (2)

```
#define TRY \
  {\
    context_block _cb;\
    int state = ES_EvalBody;\
    _cb.next=exceptionStack;\
    _cb.found=0;\
    exceptionStack=&_cb;\
    if (setjmp(_cb.env) != 0) state=ES_Exception;\
    while(1){\
      if (state == ES_EvalBody){

#define EXCEPT(e)\
    if (state == ES_EvalBody) exceptionStack=exceptionStack->next;\
    break;\
    }\
    if (state == ES_Exception) \
      if ((_cb.exn == (exception)&e) || (&e == &ANY)) {\
        int exception_value = _cb.val;\
        _cb.found=1;
```

Exceptions en C : implantation (3)

```
#define ENDTRY \
   }\
  if (state == ES_EvalBody) {exceptionStack=exceptionStack->next;break;}\
   else {exceptionStack=exceptionStack->next;\
       if (_cb.found == 0) _RaiseException(_cb.exn,_cb.val); else break;}\
   }\
}
```

Fichier C :

```
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);
  }
}
```

Continuations

Qu'est-ce qu'une continuation ?

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 :

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

devient

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

et les appels

```
fib 5 (fun x -> x);;
fib 5 print_int;;
```

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.
Lorsqu'elle est utilisée, le contexte d'où elle est appelée est écrasé et remplacé par celui qu'elle contient.

Continuation courante (2)

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

Ainsi, la valeur de:

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

sera 0 si `p` vaut `true`

Continuation courante (exemple)

Produit des éléments d'une liste

```
(define (list_mult_aux return l)
  (letrec ((r (lambda (l) (print l)
              (if (null? l) 1
              (if (= (car l) 0) (return 0)
               (print(* (car l) (r (cdr l))))))))))
     (r l)))

(define (list_mult l)
   (call/cc (lambda (c) ((list_mult_aux c l)))))

(list_mult '( 1 2 3))
```

Implantation des continuations

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

Machines à pile :

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

Concurrence et compilation

Concurrence et gestion mémoire

- Allocations concurrentes :
  - Accès en exclusion mutuelle (coûteux)
  - Allocations locales $\Rightarrow$ risque de pointeurs latéraux (entre les tas locaux)
- Récupération en présence de threads :
  - Peut nécessiter de stopper les threads
  - Nécessité de stopper les threads à un endroit cohérent
    OCaml : à l'allocation, Java : checkpoints
- Récupération en parallèle avec l'exécution :
  - Plus difficile de savoir si une valeur est vivante
  - Besoin de connaître les pointeurs en cours d'utilisation

Concurrence et optimisations

Attention aux optimisations valables sur le code séquentiel !

Exemple : la propagation de constantes.

```
var x = 0;
var y;
{
  if (x == 0) {
    y = 2;
  } else {
    y = 3;
  }
} || {
  x = 1;
}
```

Concurrence et optimisations

Attention aux optimisations valables sur le code séquentiel !

Exemple : la propagation de constantes.

```
var x = 0;
var y;
{
  if (x == 0) {
    y = 2;
  } else {
    y = 3;
  }
} || {
  x = 1;
}
```

```
var x = 0;
var y;
{
  y = 2;
} || {
  x = 1;
}
```

Parallélisation automatique

Difficile de paralléliser un programme efficacement :

- Difficile de prédire le nombre de threads lancés
- Si on lance trop de threads pour la tâche, on peut perdre en performances
- **Mais on veut le faire !**
  Nombreuses raisons : *sécurité, facilité, portabilité, coûts, etc...*

Solution à la mode : utilisation d'un pool de threads.
On préalloue N threads et ils exécutent les calculs provenant d'une file d'attente.

- Sans réécriture, expressivité limitée.
- Réécriture du programme à la CPS, ou contrôle explicite.
- Autant de threads que nécessaire mais seulement N en parallèle et/ou réutilisation (compromis).

Parallélisation automatique (exemple)

Exemple : `parfor (int i = 0;i < 12;i++) a[i] += b[i]`

1. Une structure de contrôle
2. Un modèle de compilation
3. Machinerie dans la bibliothèque d'exécution

Plusieurs stratégies :

- Un thread par index :

thread	0	1	2	3	4	5	6	7	8	9	10	11
index	0	1	2	3	4	5	6	7	8	9	10	11

- Distribution initiale :

thread	0	1	2	3
index	0 1 2	3 4 5	6 7 8	9 10 11

- Distribution dynamique :

thread	0	1	2	3

tâches	0	1	2	3	4	5	6	7	8	9	10	11

Quand une tâche est finie, le thread en prend une dans la liste.

Parallélisme, cache et bande passante

Un programme parallélisé peut se comporter moins bien qu'une version séquentielle :

- Deux threads utilisent des parties de mémoire différentes dont les adresses dans le cache sont les mêmes,
  - conflits de cache permanents sur les architectures à cache partagé,
  - une version séquentielle n'accèderait pas en même temps à ces adresses, donc pourrait aller plus vite
- Un programme séquentiel utilise déjà toute la bande passante mémoire
  - On ne peut pas gagner en ajoutant des threads,
  - On peut perdre à cause de la machinerie supplémentaire (et des problèmes de cache).

Références

- Implantations :
  - Map Reduce
  - OpenMP : pour C (AtejiPX pour Java, etc.)
  - CamlP3L : squelettes de parallélisme
  - Grand Central Dispatch (buzzword Apple) : pool de threads
  - OpenCL/CUDA : stream processing, pour GPU
- À lire :
  - The problem with threads, *LEE Edward A.*