Jezyki programowania i kontynuacje

1. Programowanie z kontynuacjami

Dariusz Biernacki

dabi@ii.uni.wroc.pl

Instytut Informatyki

Uniwersytet Wrocławski

Definicja i intuicje

Kontynuacja:

- reszta obliczen w danym punkcie programu
- informacja gdzie i jak kontynuowac obliczenia
- funkcja wyznaczajaca dla wyniku czesciowego wynik koncowy dzialania programu (wartosc lub stan)

Przyklad: arytmetyka

Kalkulator wyrazen arytmetycznych (od lewej do prawej): wyznaczenie wartosci wyrazenia $(3 + 5) * 2$.

oblicz	**a nastepnie**
$(3 + 5) * 2$	stop
$3 + 5$	oblicz 2 i pomnoz wyniki; stop
3	oblicz 5 i dodaj wyniki; oblicz 2 i pomnoz wyniki; stop
5	dodaj 3; oblicz 2 i pomnoz wyniki; stop
8	oblicz 2 i pomnoz wyniki; stop
2	pomnoz przez 8; stop
16	stop

Kontynuacje w informatyce

- Implementacje jezykow programowania: stos w czasie wykonania programu
- Semantyka operacyjna: semantyka redukcyjna, ewaluatory i maszyny abstrakcyjne
- Semantyka denotacyjna (kontynuacyjna): jawny porzadek obliczen, skoki, etc.
- Programowanie funkcyjne: styl kontynuacyjny (Continuation-Passing Style) i operatory kontroli
- Kompilacja jezykow funkcyjnych: SML i Scheme
- Logika i izomorfizm Curry'ego-Howarda: logika klasyczna i translacja przez podwojne zaprzeczenie

Programowanie z kontynuacjami (1)

Kontynuacje sa dostepne dla programisty na dwoch poziomach

- technika programowania w stylu kontynuacyjnym (CPS): funkcje akceptuja dodatkowy parametr – kontynuacje reprezentowana jako funkcja
- programowanie z operatorami kontroli
  - znane z imperatywnych jezykow: etykiety i skoki, wyjatki, nielokalne wyjscia z petli, etc.
  - specyficzne dla jezykow funkcyjnych – udostepniaja biezaca kontynuacje jako obiekt pierwszej klasy

Programowanie z kontynuacjami (2)

Niektore zastosowania

- nielokalne wyjscia z rekursji i iteracji
- obliczenia niedeterministyczne (nawroty)
- synchroniczne procesy wspolbiezne (coroutines, multitasking)
- programowanie interakcji sieciowych
- programy mobilne
- programowanie systemowe
- reprezentacja efektow (monady)
- normalizacja i czesciowa ewaluacja

Kontynuacje w programach (1)

```
fun f () = true

fun main () =
    if (f()) then print "Tak"
    else print "Nie"

(**********************************)

fun f () = true

fun f_continuation v =
    if v then print "Tak" else print "Nie"

fun main () = f_continuation (f())
```

Kontynuacje w programach (2)

```
fun f_continuation v =
    if v then print "Tak" else print "Nie"

fun f () = f_continuation true

fun main () = f ()

(**********************************)

fun f k = k true

fun main () =
    f (fn v => if v then print "Tak"
               else print "Nie")
```

Kontynuacje w programach (3)

```
fun fact 0
    = 1
  | fact n
    = n*(fact (n-1))

fun main n
    = fact n
```

Kontynuacje dla kolejnych wywolan rekurencyjnych `fact 4`:

```
fact 4:  fn v => v
fact 3:  fn v => 4 * v
fact 2:  fn v => 4 * 3 * v
fact 1:  fn v => 4 * 3 * 2 * v
fact 0:  fn v => 4 * 3 * 2 * 1 * v
```

Kontynuacje w programach (4)

Silnia w stylu kontynuacyjnym:

```
fun fact 0 k
    = k 1
  | fact n k
    = fact (n-1) (fn v => k (n * v))

fun main n =
    fact n (fn v => v)
```

Styl kontynuacyjny

- funkcje akceptuja dodatkowy argument funkcyjny – kontynuacje
- funkcja zwraca wartosc poprzez wyslanie jej do swojej kontynuacji
- wszystkie wywolania funkcji sa ogonowe
- wyniki obliczen czesciowych sa nazwane
- kolejnosc obliczen jest jawnie zakodowana w postaci programu

Istnieje wiele metod automatycznej transformacji programow do stylu kontynuacyjnego.

Transformacja do CPS (1)

Zalozmy, ze funkcje f, g i h sa jednoargumentowe, a k reprezentuje biezaca kontynuacje. Wowczas wyrazenie

```
f (g h) j
```

zostanie przetransformowane do

```
g h (fn v => f v (fn u => u j k))
```

Transformacja do CPS (2)

```
datatype aexp = Num of int
      | Add of aexp * aexp
      | Mul of aexp * aexp

(* eval : aexp -> int *)
fun eval (Num i) = i
  | eval (Add (a1, a2))
    = (eval a1) + (eval a2)
  | eval (Mul (a1, a2))
    = (eval a1) * (eval a2)

fun main e = eval e
```

Transformacja do CPS (3)

```
(*  eval : aexp -> (int -> 'a) -> 'a  *)
fun eval (Num i) k
    = k i
  | eval (Add (a1, a2)) k
    = eval a1 (fn i1 =>
       eval a2 (fn i2 => k (i1 + i2)))
  | eval (Mul (a1, a2)) k
    = eval a1 (fn i1 =>
        eval a2 (fn i2 => k (i1 * i2)))

(*  main : aexp -> int  *)
fun main e = eval e (fn i => i)
```

Transformacja ze stylu kontynuacyjnego (1)

Czasami interesuje nas transformacja odwrotna: dla danego programu w stylu kontynuacyjnym, znalezc program, ktorego obrazem przez transformacje do stylu kontynuacyjnego jest dany program.

Istnieja metody automatycznej transformacji programow ze stylu kontynuacyjnego.

Transformacja ze stylu kontynuacyjnego (2)

Obrazem wyrazenia

```
g h (fn v => f v (fn u => u j k))
```

jest wyrazenie

```
let val v = g h
in let val u = f v
   in u j
   end
end
```

rownowazne

```
f (g h) j
```

Transformacja ze stylu kontynuacyjnego (3)

```
fun append_cps [] ys k
    = k ys
  | append_cps (x::xs) ys k
    = append_cps xs ys (fn vs => k (x::vs))


fun append [] ys
    = ys
  | append (x::xs) ys
    = x :: (append xs ys)
```

Zmiana reprezentacji kontynuacji

- Kontynuacja w programie jest utozsamiana ze stosem
- Pytania
  - Czy ta odpowiedniosc moze byc zaobserwowana na poziomie programow funkcyjnych?
  - Jaka jest reprezentacja pierwszego rzedu dla kontynuacji w programach w stylu kontynuacyjnym?
  - Czy istnieje mechaniczna (a moze automatyczna) metoda przejscia z jednej reprezentacji do drugiej?
- Odpowiedz
  - defunkcjonalizacja
  - refunkcjonalizacja

Defunkcjonalizacja

Transformacja programow przeksztalcajaca programy wyzszego rzedu do semantycznie rownowaznych im programow pierwszego rzedu, poprzez zmiane reprezentacji lambda abstrakcji.

Algorytm: zakladamy, ze chcemy zdefunkcjonalizowac okre-

slona przestrzen funkcyjna t1 -> t2.

Algorytm defunkcjonalizacji (1)

- identyfikujemy wszystkie lambda abstrakcje “zamieszkujace” dana przestrzen funkcyjna
- wprowadzamy typ danych t do reprezentowania lambda abstrakcji – po jednym konstruktorze dla kazdej lambda abstrakcji; kazdy konstruktor jest typu s1 * ... * sn -> t, gdzie s1,...,sn sa typami zmiennych wolnych w ciele danej lambda abstrakcji
- wprowadzamy funkcje apply : t -> t1 -> t2, ktora dla kazdego konstruktora typu t i wartosci typu t1, zwraca wartosc aplikacji lambda abstrakcji reprezentowanej przez ten konstruktor na tej wartosci

Algorytm defunkcjonalizacji (2)

- kazde wprowadzenie danej lambda abstrakcji zastepujemy uzyciem odpwiadajacego jej konstruktora zaaplikowanego do zmiennych wolnych w ciele lambda abstrakcji
- kazda eliminacje lambda abstrakcji z danej przestrzeni zastepujemy wywolaniem funkcji apply

Defunkcjonalizacja: przyklad (1)

```
(*  aux : (int -> int) -> int  *)
fun aux f = f 1 + f 10

(*  main : int -> int -> bool -> int  *)
fun main a b c
    = aux (fn x => a + x)
      * aux (fn y => if c then b else y)
```

Defunkcjonalizacja: przyklad (2)

```
datatype lam = LAM1 of int
             | LAM2 of int * bool

fun apply (LAM1 a) x = a + x
  | apply (LAM2 (b, c)) y = if c then b else y

fun aux_def f = apply f 1 + apply f 10

fun main_def a b c = aux_def (LAM1 a)
                   * aux_def (LAM2 (b, c))
```

Defunkcjonalizacja kontynuacji (1)

```
(*  fact_cps : int -> (int -> int) -> int  *)
fun fact_cps 0 k
    = k 1
  | fact_cps n k
    = fact_cps (n-1) (fn v => k (n*v))

(*  fact : int -> int  *)
and fact n = fact_cps n (fn v => v)
```

Defunkcjonalizacja kontynuacji (2)

```
datatype cont = CONT0 | CONT1 of int * cont

(*  fact_cps_def : int -> cont -> int  *)
fun fact_cps_def 0 k
    = apply k 1
  | fact_cps_def n k
    = fact_cps_def (n-1) (CONT1 (n, k))
(*  apply : cont -> int  *)
and apply CONT0 v = v
  | apply (CONT1 (n, k)) v
    = apply k (n * v)

(*  fact : int -> int  *)
and fact n = fact_cps_def n CONT0
```

Defunkcjonalizacja kontynuacji (3)

```
datatype stack = EMPTY | PUSH of int * stack

(*  fact_cps_def : int -> stack -> int  *)
fun fact_cps_def 0 s
    = pop_and_mul s 1
  | fact_cps_def n s
    = fact_cps_def (n-1) (PUSH (n, s))
(*  pop_and_mul : stack -> int  *)
and pop_and_mul EMPTY v = v
  | pop_and_mul (PUSH (n, s)) v
    = pop_and_mul s (n * v)

(*  fact : int -> int  *)
and fact n = fact_cps_def n EMPTY
```

Refunkcjonalizacja

Transformacja programow przeksztalcajaca programy pierwszego rzedu do semantycznie rownowaznych im programow wyzszego rzedu.

Algorytm (zakladamy, ze zidentyfikowalismy typ t, oraz funkcje apply typu t -> t1 -> t2 bedace w obrazie defunkcjonalizacji)

- zastap wywolania funkcji apply f x przez f x
- zastap wystapienia konstruktorow typu t przez lambda abstrakcje o ciele zdefiniowanym przez odpowiadajaca temu konstruktorowi klauzule funkcji apply
- usun definicje typu t
- usun definicje funkcji apply

Refunkcjonalizacja: przyklad (1)

Konwolucja dwoch listy: `conv ([x1,...,xn], [y1,...,yn] ==> [(x1,yn),...,(xn,y1)]`

```
fun conv (xs, ys)
    = let fun reverse (nil, a)
                = pair (a, ys, nil)
            | reverse (x::xs, a)
                = reverse (xs, x::a)
          and pair (nil, nil, r)
                = r
            | pair (x::a, y::ys, r)
                = pair (a, ys, (x,y)::r)
      in reverse (xs, nil) end
```

Refunkcjonalizacja: przyklad (2)

```
datatype cont = CONT0 | CONT1 of int * cont

fun conv xs ys
    = let fun reverse (nil, k)
                = apply (k, ys, nil)
            | reverse (x::xs, k)
                = reverse (xs, CONT1 (x, k))
          and apply (CONT0, nil, r)
                = r
            | apply (CONT1 (x, k), y::ys, r)
                = apply (k, ys, (x, y)::r)
      in reverse (xs, CONT0) end
```

Refunkcjonalizacja: przyklad (3)

```
fun conv (xs, ys)
    = let fun reverse (nil, k)
              = k (ys, nil)
            | reverse (x::xs, k)
              = reverse (xs,
                         fn (y::ys, r)
                          => k (ys, (x,y)::r))
      in reverse (xs, fn (nil, r) => r) end
```

Refunkcjonalizacja: przyklad (4)

```
fun conv (xs, ys)
    = let fun reverse nil
              = (ys, nil)
            | reverse (x::xs)
              = let val (y::ys, r)
                        = reverse xs
                in (ys, (x,y)::r) end
          val (nil, r) = reverse xs
      in r end
```

Big Picture

Trzy semantycznie rownowazne postaci programu:

1. program w DS
2. program w CPS
3. program ze stosem, implementujacy system przejsc pierwszego rzedu

Transformacje pozwalajace na mechaniczne przejscia pomiedzy nimi:

- CPS transformacja (1 -> 2) i jej odwrotnosc (2 -> 1)
- defunkcjonalizacja kontynuacji (2 -> 3) i ich refunkcjonalizacja (3 -> 2)

Manipulowanie kontynuacja (1)

```
(*  multlist : int list -> int  *)
fun multlist xs =
  let fun walk [] k
          = k 1
        | walk (0::xs) k
          = 0
        | walk (x::xs) k
          = walk xs (fn v => k (x * v))
  in walk xs (fn v => v) end
```

Manipulowanie kontynuacja (2)

```
datatype aexp = ... | Div of aexp * aexp

datatype result = VALUE of int | ERROR

(* eval : aexp -> (int -> result) -> result *)
  | eval (Div (a1, a2)) k
    = eval a1 (fn i1 =>
                 eval a2 (fn i2 =>
                            division i1 i2 k))

fun division i1 i2 k
    = if i2 = 0 then ERROR else k (i1 div i2)

fun main e = eval e (fn i => VALUE i)
```

Operatory kontroli

- manipulowanie biezaca kontynuacja bez koniecznosci programowania w stylu kontynuacyjnym
- dostep do biezacej kontynuacji jako obiektu pierwszej klasy

callcc (call with current continuation)

- Motywacja
  - Nielokalne wyjscia (CPS)
  - Symulacja innych efektow kontroli w zestawieniu z przypisaniem
- Semantyka wyrazenia `callcc (fun k => e)`
  - biezaca kontynuacja (w pewnej konkretnej reprezentacji) staje sie wartoscia `k`
  - obliczana jest wartosc wyrazenia `e`
  - jezeli `k` nie jest uzyta w `e`, to wartosc wyrazenia `e` jest wartoscia wyrazenia `callcc (fun k => e)`
  - jezeli `k` jest zaaplikowana do wartosci `v`, to biezaca kontynuacja zostaje zastapiona kontynuacja zwiazana z `k` i `v` jest wartoscia calego wyrazenia

callcc (2)

- Standard ML of New Jersey: struktura `SMLofNJ.Cont`

```
type 'a cont
val callcc : ('a cont -> 'a) -> 'a
val throw : 'a cont -> 'a -> 'b
```

- Scheme: `call-with-current-continuation` (`call/cc`)

callcc (3)

- Przechwycona kontynuacja (`1+[]`) nie zostaje uzyta:

```
1 + callcc (fn k => 10 + 100)
==> 111
```

- Przechwycona kontynuacja (`1+[]`) zastepuje biezaca (`1+(10+[])`):

```
1 + callcc (fn k => 10 + (throw k 100))
==> 101
```

Nielokalne wyjscia z rekursji

```
(*  multlist : int list -> int  *)
fun multlist xs =
  callcc
   (fn exit =>
       let fun walk []
                = 1
             | walk (0::xs)
                = throw exit 0
             | walk (x::xs)
                = x * (walk xs)
       in walk xs end)
```

Obsluga bledow

```
(*  eval aexp -> result  *)
fun eval_aexp a =
    callcc
     (fn exit =>
         let fun division i1 i2 =
               if i2 = 0
               then throw exit ERROR
               else i1 div i2
             fun eval (Num i) = i
                 ...
               | eval (Div (a1, a2))
                 = division (eval a1)
                            (eval a2)
         in VALUE (eval a) end)
```

Obliczenia z nawrotami (1)

- Operator `amb` McCarthy'ego
- Niedeterminizm: `amb( )` generuje dwie galezie obliczen, w jednej przyjmujac wartosc true, w drugiej false
- Interesuje nas ogolna funkcja `backtrack`, ktora dla danej funkcji uzywajacej operatora `amb`, zbiera wyniki ze wszystkich galezi (wykonuje nawroty)

Obliczenia z nawrotami (2)

```
(* backtrack :
    ((unit -> bool) -> 'a) -> 'a list *)

val res = backtrack
            (fn amb => if (amb ())
                       then if (amb ())
                            then 1
                            else 2
                       else if (amb ())
                            then 3
                            else 4)

(* val res = [4,3,2,1] : int list *)
```

Obliczenia z nawrotami (3)

```
fun backtrack f =
  let val res = ref []
      val conts = ref []
  in (res :=
        f (fn () =>
             callcc (fn k =>
                        (conts := k :: !conts;
                         true))) :: !res;
      case !conts of
          [] => !res
        | (k::conts') => (conts := conts';
                          throw k false))
  end
```

Obliczenia z nawrotami (4)

Kontynuacja przechwycona przez `callcc` w programie

```
backtrack (fn amb => if (amb()) then 1 else 2)
```

“reprezentuje” kontekst

```
(res := (if [] then 1 else 2) :: !res;
 case !conts of
   [] => !res
 | (k::conts') => (conts := conts';
                   throw k false))
```

Coroutines (1)

- Procedury, ktore moga dobrowolnie oddawac sterowanie innej procedurze
- Ponowne odzyskanie sterowania powoduje wznowienie obliczen od punktu, w ktorym sterowanie bylo oddane
- Implementujemy funkcje `coroutine-maker`, ktora z danej funkcji tworzy coroutine

Coroutines (2)

```
(define producer
  (coroutine-maker
    (lambda (resume d)
      (letrec ((loop (lambda (n)
                       (resume consumer n)
                       (loop (+ n 1)))))
        (loop 0)))))
```

Coroutines (3)

```
(define consumer
  (coroutine-maker
    (lambda (resume d)
      (letrec
          ((loop
             (lambda ()
               (let ((n (resume producer d)))
                     (display n)
                     (newline)
                     (loop)))))
        (loop)))))
```

Coroutines (4)

```
(define coroutine-maker
  (lambda (proc)
    (let ((saved-cont '()))
      (let ((update-cont!
             (lambda (k)
               (set! saved-cont k))))
        (let ((resumer
               (resume-maker update-cont!))
              (first-time #t))
          (lambda (value)
            (if first-time
                (begin (set! first-time #f)
                       (proc resumer value))
                (saved-cont value)))))))))
```

Coroutines (5)

```
(define resume-maker
  (lambda (update-proc!)
    (lambda (next-coroutine value)
      (call/cc
        (lambda (continuation)
          (update-proc! continuation)
          (next-coroutine value))))))
```

Coroutines (6)

Kontynuacje przechwytywane przez procedury `producer` i `consumer` “reprezentuja” konteksty, odpowiednio:

```
(let ((n []))
  (display n)
  (newline)
  (loop))
```

oraz

```
(let ((d []))
  (loop (+ n 1)))
```

Literatura

- Continuations in Programming Practice: Introduction and Survey, M.Felleisen, A.Sabry
- Definitional Interpreters for Higher-Order Programming Languages, J.Reynolds
- The Discoveries of Continuations, J.Reynolds
- Defunctionalization at Work, O.Danvy, L.Nielsen
- Refunctionalization at Work, O.Danvy, K.Millikin
- Applications of Continuations, D.Friedman
- Call with Current Continuation Patterns, D.Ferguson, D.Deugo