Jezyki programowania i kontynuacje

3. Kontynuacje ograniczone

Dariusz Biernacki

dabi@ii.uni.wroc.pl

Instytut Informatyki

Uniwersytet Wrocławski

Kontynuacje ograniczone i skladalne

- Kontynuacje nieograniczone (abortywne)
  - reprezentuja cala reszte programu
  - modeluja skoki (“nigdy nie wracaja”))
- Kontynuacje ograniczone (skladalne)
  - reprezentuja prefiks reszty programu
  - modeluja funkcje

Kontynuacje ograniczone - motywacja

- wieksza sila wyrazu niz kontynuacje nieograniczone
- bezposredni zwiazek z dwuwarstwowym CPS-em, czyli modelem obliczen z kontynuacja sukcesu i porazki
- mnostwo zastosowan praktycznych
- wygodne w rozwazaniach teoretycznych

Kontynuacje ograniczone w programach

Trzy postaci

- Styl kontynuacyjny, gdzie nie wszystkie wywolania sa ogonowe – Continuation-Composing Style (CCS)
- Dwuwarstwowy styl kontynuacyjny (2CPS)
- Operatory kontroli dla kontynuacji ograniczonych

Continuation-Composing Style

- funkcje przyjmuja dodatkowy parametr funkcyjny – kontynuacje ograniczona
- kontynuacja ograniczona reprezentuje prefiks reszty obliczen
- zagniezdzone wywolania funkcji i kontynuacji modeluja zlozenie kontynuacji
- przyklad:

```
let val k = fn v => v + 1
in k (let val k' = fn v => 10 + v
      in 100 + k' (k' 0)
      end)
end
```

Prefiksy listy w CCS

```
(*  prefixes_ccs : 'a list -> 'a list list  *)
fun prefixes_ccs xs
    = let fun walk (nil, k)
                = nil
            | walk (x :: xs, k)
              = (k (x :: nil))
                ::
                (walk (xs, fn vs => k (x :: vs)))
      in walk (xs, fn vs => vs) end

(*walk:'a list->('a list->'a list)->'a list list*)
```

Niedeterminizm w CCS (1)

Operator niedeterministycznego wyboru oraz operator porazki:

```
fun amb_ccs (k:bool->unit)
    = (k true; k false)


fun fail_ccs (k:'a->unit)
    = ()
```

Niedeterminizm w CCS (2)

```
fun test_ccs k =
  amb_ccs (fn b =>
            if b then amb_ccs (fn b =>
                               if b then k 1
                               else fail_ccs k
    else k 3))

- test_ccs
    (fn v => print ((Int.toString v) ^ "\n"))
1
3
val it = () : unit
```

Dwuwartswowy CPS (2CPS)

- funkcje akceptuja dwie kontynuacje: kontynuacje ograniczona i meta-kontynuacje
- meta-kontynuacja jest kontynuacja dla kontynuacji ograniczonej
- wszystkie wywolania sa ogonowe (program znow jest w CPS, tyle, ze uzywa 2 kontynuacji)
- CPS transformacja programow w CCS prowadzi do programow w 2CPS
- ```
  let val k = fn v => fn mk => mk (v + 1)
  in let val k' = fn v => fn mk => mk (10 + v)
     in k' 0 (fn v => k' v (fn v => k (100 + v)
                                (fn v => v))) end end
  ```
  ```

Prefiksy w 2CPS

```
(*  prefixes_cps : 'a list -> 'a list list  *)
fun prefixes_cps xs =
let fun walk (nil, k, mk) = mk nil
      | walk (x :: xs, k, mk) =
        k (x :: nil,
           fn vs =>
            walk (xs,
                  fn (vs, mk) => k (x :: vs, mk),
                  fn vss => mk (vs :: vss)))
  in walk (xs,
           fn (vs, mk) => mk vs,
           fn vss => vss) end
```

Niedeterminizm w 2CPS (1)

Kontynuacja sukcesu (k) i kontynuacja porazki (mk):

```
fun amb_cps (k:bool->(unit->'a)->'a)
            (mk:unit->'a)
    = k true (fn () => k false mk)


fun fail_cps (k:'a->(unit->'b)->'b)
             (mk:unit->'b)
    = mk ()
```

Niedeterminizm w 2CPS (2)

```
fun test_cps k mk =
  amb_cps
   (fn b => fn mk' =>
       if b then k 1 mk'
       else amb_cps (fn b => fn mk'' =>
                       if b then fail_cps k mk''
                       else k 3 mk'') mk') mk

-  test_cps (fn v => fn k2' =>
              k2' (print ((Int.toString v)^"\n")))
           (fn v => v)
1
3
val it = () : unit
```

Operatory shift i reset (1)

Operatory kontroli pozwalajace na wyrazenie CCS w DS:

- `reset` – ogranicznik kontroli, wyznacza biezaca kontynuacje ograniczona
- `shift` – przechwytuje biezaca kontynuacje ograniczona i udostepnia ja jako wartosc pierwszej klasy

Operatory shift i reset (2)

Implementacja Filinskiego (SML):

```
signature SHIFT_AND_RESET =
  sig
    type answer
    val shift : ((’a -> answer) -> answer) -> ’a
    val reset : (unit -> answer) -> answer
  end

functor
  Shift_and_Reset(<param>: sig type answer end)
  : SHIFT_AND_RESET
```

Operatory shift i reset (3)

```
structure SR = Shift_and_Reset (type answer = int)

- 1 + (SR.reset
        (fn () => 10 + (SR.shift
                          (fn k => 100 + k (k 0)))))

val it = 121 : int
```

Prefiksy listy w DS

```
(*  prefixes_ds : 'a list -> 'a list list  *)
fun prefixes_ds xs =
let fun walk nil
      = shift (fn k => nil)
    | walk (x :: xs)
      = shift (fn k =>
                 (k (x :: nil))
                 ::
                 (reset (fn () => k (x :: (walk xs)))))
in reset (fn () => walk xs) end

(* walk:'a list->'a list *)
```

Niedeterminizm w DS (1)

```
structure SR
  = Shift_and_Reset (type answer = unit);

(*  amb : unit -> bool  *)
fun amb () = SR.shift (fn k => (k true; k false))

(*  fail : unit -> 'a  *)
fun fail () = SR.shift (fn k => ())
```

Niedeterminizm w DS (2)

```
fun test ()
    = if amb() then 1
      else if amb() then fail()
           else 3

- SR.reset
   (fn () => print ((Int.toString (test()))^"\n"))
1
3
val it = () : unit
```

Niedeterminizm w DS (3)

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

fun backtrack f =
let val res = ref []
    fun amb () = SR.shift
                   (fn k => (k true; k false))
    fun fail () = SR.shift (fn k => ())
    fun emit v = res := v :: !res
    val () = SR.reset (fn () => emit (f amb fail))
in !res end
```

Transformacja do CCS

- Jezyk ISWIM$_s$

$$e ::= x \mid \lambda x.e \mid e_0\, e_1 \mid \ulcorner m \urcorner \mid succ\ e \mid \mathcal{S}x.e \mid \langle\!\langle e \rangle\!\rangle$$

- Transformacja do CCS:

$$\begin{aligned}
\overline{x} &= \lambda k.k\, x \\
\overline{\lambda x.e} &= \lambda k.k\, (\lambda x k.\, \overline{e}\, k) \\
\overline{e_0\, e_1} &= \lambda k.\overline{e_0}\, (\lambda v_0.\, \overline{e_1}\, (\lambda v_1.\, v_0\, v_1\, k)) \\
\overline{\ulcorner m \urcorner} &= \lambda k.k \ulcorner m \urcorner \\
\overline{succ\ e} &= \lambda k.\overline{e}\, (\lambda v.\, k\, (succ\ v)) \\
\overline{\mathcal{S}x.e} &= \lambda k.\overline{e}\{(\lambda v k'.\, k'\, (k\, v))/x\}\, (\lambda v.\, v) \\
\overline{\langle\!\langle e \rangle\!\rangle} &= \lambda k.k\, (\overline{e}\, (\lambda v.\, v))
\end{aligned}$$

Transformacja do 2CPS

$$\overline{\overline{\mathcal{S}x.e}} = \lambda k_1 k_2.\bar{\bar{e}}\{(\lambda v k_1' k_2'.\, k_1\, v\, (\lambda w.\, k_1'\, w\, k_2'))/x\}\, (\lambda v k_2.\, k_2\, v)\, k_2$$

$$\overline{\overline{\langle\!\langle e \rangle\!\rangle}} = \lambda k_1 k_2.\bar{\bar{e}}(\lambda v k_2.\, k_2\, v)\, (\lambda v.\, k_1\, v\, k_2)$$

W pozostalych konstrukcjach jezyka meta-kontynuacja nie odgrywa zadnej roli i moze byc η-zredukowana do postaci podanej na poprzednim slajdzie.

Ewaluator w 2CPS (1)

- Wyrazenia:

  $\mathsf{exp} \ni e ::= \ulcorner m \urcorner \mid x \mid \lambda x.e \mid e_0\, e_1 \mid succ\ e \mid \langle\!\langle e \rangle\!\rangle \mid \mathcal{S}k.e$

- Wartosci: $\mathsf{val} \ni v ::= m \mid f$

- Typ odpowiedzi, meta-kontynuacji, kontynuacji i funkcji:

$$\begin{aligned} \mathsf{ans} &= \mathsf{val} \\ k_2 \in \mathsf{cont}_2 &= \mathsf{val} \to \mathsf{ans} \\ k_1 \in \mathsf{cont}_1 &= \mathsf{val} \times \mathsf{cont}_2 \to \mathsf{ans} \\ f \in \mathsf{fun} &= \mathsf{val} \times \mathsf{cont}_1 \times \mathsf{cont}_2 \to \mathsf{ans} \end{aligned}$$

- Poczatkowa kontynuacja i meta-kontynuacja:

$$\begin{aligned} \theta_1 &= \lambda(v,\, k_2).\, k_2\ v \\ \theta_2 &= \lambda v.\, v \end{aligned}$$

- Srodowisko: $\mathsf{env} \ni \rho ::= \rho_{mt} \mid \rho\{x \mapsto v\}$

Ewaluator w 2CPS (2)

- Funkcja ewaluacyjna:
  $\mathsf{eval} : \mathsf{exp} \times \mathsf{env} \times \mathsf{cont}_1 \times \mathsf{cont}_2 \longrightarrow \mathsf{ans}$

$$
\begin{aligned}
\mathsf{eval}\ (\ulcorner m \urcorner,\ \rho,\ k_1,\ k_2) &= k_1\ (m,\ k_2) \\
\mathsf{eval}\ (x,\ \rho,\ k_1,\ k_2) &= k_1\ (\rho(x),\ k_2) \\
\mathsf{eval}\ (\lambda x.e,\ \rho,\ k_1,\ k_2) &= k_1\ (\lambda(v,\ k_1',\ k_2').\, \mathsf{eval}\ (e,\ \rho\{x \mapsto v\},\ k_1',\ k_2'),\ k_2) \\
\mathsf{eval}\ (e_0\, e_1,\ \rho,\ k_1,\ k_2) &= \mathsf{eval}\ (e_0,\ \rho, \\
&\quad \lambda(f,\ k_2').\, \mathsf{eval}\ (e_1,\ \rho,\ \lambda(v,\ k_2'').\, f\ (v,\ k_1,\ k_2''),\ k_2'),\ k_2) \\
\mathsf{eval}\ (succ\ e,\ \rho,\ k_1,\ k_2) &= \mathsf{eval}\ (e,\ \rho,\ \lambda(m,\ k_2').\, k_1\ (m+1,\ k_2'),\ k_2) \\
\mathsf{eval}\ (\langle\!\langle e \rangle\!\rangle,\ \rho,\ k_1,\ k_2) &= \mathsf{eval}\ (e,\ \rho,\ \theta_1,\ \lambda v.\, k_1\ (v,\ k_2)) \\
\mathsf{eval}\ (\mathcal{S}k.e,\ \rho,\ k_1,\ k_2) &= \mathsf{eval}\ (e,\ \rho\{k \mapsto c\},\ \theta_1,\ k_2) \\
&\quad \text{gdzie } c = \lambda(v,\ k_1',\ k_2').\, k_1\ (v,\ \lambda v'.\, k_1'\ (v',\ k_2'))
\end{aligned}
$$

Ewaluator w 2CPS (3)

- Glowna funkcja: evaluate : exp $\longrightarrow$ val

$$\text{evaluate}\,(e) \quad = \quad \text{eval}\,(e,\, \rho_{mt},\, \theta_1,\, \theta_2)$$

Maszyna abstrakcyjna (1)

- Wyrazenia: $e ::= \ulcorner m \urcorner \mid x \mid \lambda x.e \mid e_0\, e_1 \mid succ\ e \mid \langle\!\langle e \rangle\!\rangle \mid \mathcal{S}k.e$
- Wartosci:

$$v ::= m \mid [x,\ e,\ \rho] \mid C_1$$

- Srodowisko: $\rho ::= \rho_{mt} \mid \rho\{x \mapsto v\}$
- Konteksty ewaluacyjne:

$$C_1 ::= [\,] \mid \mathsf{ARG}\,((e, \rho),\ C_1) \mid \mathsf{SUCC}\,(C_1) \mid \mathsf{FUN}\,(v,\ C_1)$$

- Meta-konteksty: $C_2 ::= \bullet \mid C_2 \cdot C_1$

Maszyna abstrakcyjna (2)

e	$\Rightarrow$	$\langle e,\ \rho_{mt},\ [\,],\ \bullet \rangle_{eval}$
$\langle \ulcorner m \urcorner,\ \rho,\ C_1,\ C_2 \rangle_{eval}$	$\Rightarrow$	$\langle C_1,\ m,\ C_2 \rangle_{cont_1}$
$\langle x,\ \rho,\ C_1,\ C_2 \rangle_{eval}$	$\Rightarrow$	$\langle C_1,\ \rho\,(x),\ C_2 \rangle_{cont_1}$
$\langle \lambda x.e,\ \rho,\ C_1,\ C_2 \rangle_{eval}$	$\Rightarrow$	$\langle C_1,\ [x,\ e,\ \rho],\ C_2 \rangle_{cont_1}$
$\langle e_0\, e_1,\ \rho,\ C_1,\ C_2 \rangle_{eval}$	$\Rightarrow$	$\langle e_0,\ \rho,\ \mathsf{ARG}\,((e_1, \rho),\ C_1),\ C_2 \rangle_{eval}$
$\langle succ\ e,\ \rho,\ C_1,\ C_2 \rangle_{eval}$	$\Rightarrow$	$\langle e,\ \rho,\ \mathsf{SUCC}\,(C_1),\ C_2 \rangle_{eval}$
$\langle \langle\!\langle e \rangle\!\rangle,\ \rho,\ C_1,\ C_2 \rangle_{eval}$	$\Rightarrow$	$\langle e,\ \rho,\ [\,],\ C_2 \cdot C_1 \rangle_{eval}$
$\langle \mathcal{S}k.e,\ \rho,\ C_1,\ C_2 \rangle_{eval}$	$\Rightarrow$	$\langle e,\ \rho\{k \mapsto C_1\},\ [\,],\ C_2 \rangle_{eval}$

Maszyna abstrakcyjna (3)

$$
\begin{array}{rcl}
\langle [\,],\, v,\, C_2 \rangle_{cont_1} & \Rightarrow & \langle C_2,\, v \rangle_{cont_2} \\
\langle \mathsf{ARG}\,((e, \rho),\, C_1),\, v,\, C_2 \rangle_{cont_1} & \Rightarrow & \langle e,\, \rho,\, \mathsf{FUN}\,(v,\, C_1),\, C_2 \rangle_{eval} \\
\langle \mathsf{SUCC}\,(C_1),\, m,\, C_2 \rangle_{cont_1} & \Rightarrow & \langle C_1,\, m+1,\, C_2 \rangle_{cont_1} \\
\langle \mathsf{FUN}\,([x,\, e,\, \rho],\, C_1),\, v,\, C_2 \rangle_{cont_1} & \Rightarrow & \langle e,\, \rho\{x \mapsto v\},\, C_1,\, C_2 \rangle_{eval} \\
\langle \mathsf{FUN}\,(C_1',\, C_1),\, v,\, C_2 \rangle_{cont_1} & \Rightarrow & \langle C_1',\, v,\, C_2 \cdot C_1 \rangle_{cont_1} \\
\hline
\langle C_2 \cdot C_1,\, v \rangle_{cont_2} & \Rightarrow & \langle C_1,\, v,\, C_2 \rangle_{cont_1} \\
\hline
\langle \bullet,\, v \rangle_{cont_2} & \Rightarrow & v
\end{array}
$$

Reprezentacja monad

W CPS mozna reprezentowac dowolna monade `(t, unit, bind)`

- typ odpowiedzi kontynuacji: `'a t`
- reprezentacja operacji `m` typu `'a t`: `fn k => bind m k`
- translacja odwrotna, z wyrazenia kontynuacyjnego `r : ('a->'a t) -> 'a t` do monadycznego: `r unit`

Dzieki operatorom `shift` i `reset`, konstrukcja powyzsza przenosi sie do DS.

Hierarchia CPS (1)

Zbieranie wynikow produkowanych przy uzyciu amb i `fail`:

```
fun emit_2c v k mk = v :: (k () mk)

fun emit_3c v k mk mmk
    = k () mk (fn vs => mmk (v::vs))
```

W DS konieczne jest uzycie operatorow `shift2` i `reset2`, ktore pozwalaja na dostep do kontynuacji i meta-kontynuacji

```
fun emit v
    = shift2 (fn k => v::(k ()))
```

Hierarchia CPS (2)

- Iterujac CPS transformacje otrzymujemy hierarchie CPS
- Na poziomie n, mamy do dyspozycji operatory $\texttt{shift}_i$ i $\texttt{reset}_i$, gdzie $1 \leq i \leq n$
- CPS transformacja i ewaluator uzywaja n+1 kontynuacji
- Maszyna abstrakcyjna uzywa n+1 kontekstow ewaluacyjnych

Inne operatory

- Istnieja inne operatory operujace na kontynuacjach ograniczonych
- Sa one definiowane na gruncie operacyjnym, bez odwolania do CPS
- Np. dynamiczne operatory Felleisena, `control` i `prompt` sa zdefiniowane przez mala modyfikacje maszyny dla statycznych operatorow `shift` i `reset`, ktora sprawia, ze maszyna ta nie jest w zdefunkcjonalizowanej formie
- Mozna zdefiniowac CPS dla operatorow dynamicznych, ale to wymaga kontynuacji rekursywnych

Literatura

- Abstracting Control, O.Danvy, A.Filinski
- An Operational Foundation for Delimited Continuations in the CPS Hierarchy, M.Biernacka et al.
- Representing Monads, A.Filinski
- The Theory and Practice of First-class Prompts, M.Felleisen
- A Monadic Framework for Delimited Continuations, K.Dybvig et al.