Gentle introduction to delimited continuations

Дмитрий Косарев

19 ноября 2018 г.

Обзор

- call-with-current-continuation (callCC)
- ✔ `shift/reset`
- ✔ Примеры
- ✔ Предикативный полиморфизм
- Импредикативный полиморфизм
- CPS-преобразование
- `prompt`
- `cupto`
- Через монады [DPJS07]

Что такое продолжение (continuation)?

Continuation

Это остаток вычисления

- Текущее вычисление: внутри []
- Остаток вычисления: снаружи []

Пример: 3+[5*2]−1

- Текущее вычисление: 5*2
- Остаток вычисления: 3+[·]−1

“Если дали значение для ”дырки” [·], то прибавить 3 и вычесть 1“, т.е. $\underline{\texttt{fun}}\ \texttt{x}\ \rightarrow\ 3 + \texttt{x} - 1$.

Что такое продолжение (continuation)?

Continuation

Это остаток вычисления

Продолжения можно потерять по мере вычисления.
Например: `3+[5*2]-1`

- Заменим `[·]` на `raise Abort`:
  `3+[raise Abort]-1`
- Теряющееся продолжение `3+[·]-1` является текущим

Разграниченные продолжения (continuation)?

Continuation

Это остаток вычисления

Синтаксис

`reset (fun () → M)`

Например: `reset (fun () → 3 + [5*2]) − 1`

- Текущее вычисление: 5*2
- Текущее разграниченное продолжение: 3+[·]

Shift

Синтаксис

$\texttt{shift}\ (\underline{\texttt{fun}}\ \texttt{k}\ \rightarrow\ \texttt{M})$

- забывает текущее продолжение
- сохраняет забытое как k
- и исполняет $\texttt{M}$

Например:

$$\texttt{reset}\ (\underline{\texttt{fun}}\ ()\ \rightarrow\ 3 + [\texttt{shift}\ (\underline{\texttt{fun}}\ \texttt{k}\ \rightarrow\ \texttt{M})]) - 1$$
$$\Downarrow$$
$$\texttt{reset}\ (\underline{\texttt{fun}}\ ()\ \rightarrow\ [\texttt{shift}\ (\underline{\texttt{fun}}\ \texttt{k}\ \rightarrow\ \texttt{M})]) - 1$$
$$\text{где } \mathsf{k} = \mathsf{reset}\ (\mathsf{fun}\ ()\ \text{->}\ 3 + [\cdot])$$

Полученными продолжениями можно не пользоваться

$\texttt{shift}\ (\underline{\texttt{fun}}\ \_\ \rightarrow\ \texttt{M})$

- Сохраненное продолжение просто отбрасывается
- Очень похоже на исключения

Пример:

$$\texttt{reset}\ (\underline{\texttt{fun}}\ ()\ \rightarrow\ 3 + [\texttt{shift}\ (\underline{\texttt{fun}}\ \_\ \rightarrow\ 2)])\ -\ 1$$

$$\Downarrow$$

$$\texttt{reset}\ (\underline{\texttt{fun}}\ ()\ \rightarrow\ \qquad\qquad 2)\ -\ 1$$

$$\text{где } k = \text{reset (fun () -> } 3 + [\cdot])$$

$$\Downarrow$$

$$2\ -\ 1$$

$$\Downarrow$$

$$1$$

Упражнение

Дан список чисел, нужно их перемножить, а если встретился ноль, то *сразу* вернуть ноль.

```
let rec times lst = match lst with
  | [] → 1
  | 0 :: tl → ???
  | h :: tl → h * times tl
```

Вызывать функцию будем так:

```
reset (fun () → times [1; 2; 0; 4])
```

Ответ на упражнение

Дан список чисел, нужно их перемножить, а если встретился ноль, то *сразу* вернуть ноль.

```
# let rec times lst = match lst with
  | [] -> 1
  | 0 :: tl -> shift (fun _ -> 0)
  | h :: tl -> h * times tl
;;
times : int list => int = <fun>
# reset (fun () -> times [1;2;0;4]) ;;
- : int = 0
# reset (fun () -> times [1;2;3;4]) ;;
- : int = 24
```

Как сохранять продолжения

```
shift (fun k → k)
```

- Возвращаем продолжение сразу и как есть
- А потом его можно вызывать!

Пример: `reset (fun () → 3 + [...] − 1)`

```
# let f = reset (fun () → 3 + shift (fun k → k) − 1) ;;
f : int => int = <fun>
# f 10;;
− : int = 12

# let f x = reset (fun () → 3 + shift (fun k → k) − 1) x;;
f : int → int = <fun>
# f 10;;
− : int = 12
```

Упражнение

```
shift (fun k → k)
```

Вот identity для списка:

```
(* id : 'a list → 'a list *)
let rec id lst = match lst with
  | [] → []                        (* modify here *)
  | h :: tl → h :: id tl;;
```

Внесите изменение в строчке, чтобы извлечь продолжение, если функция вызывается вот так:

```
reset (fun () → id[1;2;3]) ;;
```

Что это продолжение делает?

Решение упражнения

```
(* id : 'a list → 'a list *)
let rec id lst = match lst with
  | [] → shift (fun k → k)
  | h :: tl → h :: id tl;;
id : int list => int list = <fun>
```

Решение упражнения

```
(* id : 'a list → 'a list *)
let rec id lst = match lst with
  | [] → shift (fun k → k)
  | h :: tl → h :: id tl;;
id : int list => int list = <fun>

# let append123 = reset (fun () → id[1;2;3]) ;;
append123 : int list => int list = <fun>
# append123 [4;5;6];;
- : int list = [1;2;3;4;5;6]
```

Композиция

$$\underline{\texttt{fun}}\ \texttt{x}\ \rightarrow\ \texttt{reset (f x)}$$

$$\Downarrow$$

$$\texttt{compose f g}\ = \underline{\texttt{fun}}\ \texttt{x}\ \rightarrow\ \texttt{reset (f (g x))}$$

Fix с callCC и delimCC

```
let k = callCC (fun x → c) in
k k

let fix0 f =
  reset (let x = shift (fun c → c c) in
         f (fun a → x x a))
```

Answer types

```
let rec append lst = match lst with
  | [] → shift (fun k → k)
  | h :: tl → h :: append tl;;

let append123 = reset (fun () → append [1;2;3]) ;;
```

$$1 :: 2 :: 3 :: \bullet \quad : \; \texttt{'a list}$$
$$\Downarrow \text{shift}$$
$$\lambda xs \rightarrow 1 :: 2 :: 3 :: xs \quad : \; \texttt{'a list} \rightarrow \texttt{'a list}$$

Новый вид записи типов:
`'a list / 'a list → 'a list / ('a list → 'a list)`

т.е. ∀ `'a` функция по значению с типом `'a list` возвращает `'a list` в непосредственном контексте; однако, в процессе тип результата (answer type) текущего контекста модифицируется до `'a list → 'a list`.

Полиморфизм по answer type (1/2)

Произвольные функции с типом `S → T` должны рассматриваться как полиморфные функции с типом `S/'a → T/'a`.

Рассмотрим (как бы мономорфную) функцию `let add1 x = 1+x`

```
reset (fun () → add1 x; ())
reset (fun () → add1 x; true)
```

Два типа `int/unit → int/unit` и `int/unit → int/unit` (анти)унифицируются до `int/'a → int/'a`.

Полиморфизм по answer type (2/2)

```
let rec visit lst = match lst with
| [] → shift (fun h → [])
| h :: tl → h :: shift (fun k →
                                (k []) :: reset (k (visit rest)))
let rec prefix lst = reset (visit lst)
```

Первый `shift` начинает конструирование префиксов, возвращая `[] : 'a list list`.
Второй `shift` выражает consing и применяется два раза: 1) к пустому списку чтобы получить текущий ответ и 2) чтобы сконструировать список длинных префиксов.
Продолжение `k` используется два раза в разных контекстах

- `'a list / 'a list list → 'a list / 'a list list`
- `'a list / 'a list → 'a list / 'a list`

Пару слов про Prompt и STLC

STLC обладает свойством strong normalization: последовательность редукций любого терма завершается. С добавлением delimCC – уже нет.

```
let loop () =
  let p = new_prompt () in
  let delta () = shift p (fun f v → f v v) () in
  push_prompt p (fun () → let r = delta () in
                          fun v → r
                ) delta ;;
```

Выводится тип `loop : unit → 'a`, но по сути это функция `fun () → omega` и она виснет.

Типобезопасный printf

```
let int x = string_of_int x
let str (x:string) = x

let % to_str = shift (fun k → fun x → k (to_str x))
let sprintf p = reset p

sprintf (fun () → "Hello world!")
sprintf (fun () → "Hello " ^ % str ^ "!") "world"
sprintf (fun () → "The value " ^ % str ^ " is " ^ % int) "x" 4
```

У sprintf "зависимое" поведение с типом (unit / string → string / 'a) → 'a. Без полиморфизма так нельзя было.

State monad (1/2)

Создание:

```
reset (fun () → M) 3
```

Доступ к состоянию:

```
# let get () =
    shift (fun k → fun state → k state state) ;;
get : unit => 'a = <fun>
```

Запускаем вычисление:

```
# let run_state thunk =
    reset (fun k → let result = think () in
                   fun state → result) 0 ;;
run_state : (unit => 'a) => 'b = <fun>
```

State monad (2/2)

Работаем с состоянием (пример):

```
# let tick () =
    shift (fun k → fun state → k () (state+1) ) ;;
tick : unit => unit = <fun>

# run_state (fun () →
     tick ();                    (* state = 1 *)
     tick ();                    (* state = 2 *)
     let a = get () in
     tick ();                    (* state = 3 *)
     get () − a);;
− : int = 1
```

Вызываем несколько раз

```
(* either : 'a → 'a → 'a *)
let either a b () = shift (fun k → k a; k b)
```

```
# reset (fun () →
    let x = either 0 1 in
    print_int x
    print_newline ());;
0
1
- : unit = ()
```

Generate&test

$$(P \vee Q) \wedge (P \vee \neg Q) \wedge (\neg P \vee \neg Q)$$

```
# reset (fun () →
    let p = either true false in
    let q = either true false in
    if (p||q) && (p || not q) && (not p || not q)
    then (print_string (string_of_bool p);
          print_string ", ";
          print_string (string_of_bool q);
          print_newline () );;

true, false
- : unit = ()
```

Дифференцирование парсеров (1/2)

Подробнее у Олега

```
type stream_req = ReqDone
                | ReqChar of int * (char option → stream_req)

let stream_inv p = fun pos →
  shift p (fun sk → ReqChar (pos,sk))
val stream_inv : stream_req Delimcc.prompt →
                 int → char option = <fun>

let cont str (ReqChar (pos,k) as req) = filler str pos req;;
val cont : string → stream_req → stream_req = <fun>

let finish (ReqChar (pos,k)) = filler "" pos (k None);;
val finish : stream_req → stream_req = <fun>
```

Дифференцирование парсеров (2/2)

```
let rec filler buffer buffer_pos = function
  | ReqDone  →  ReqDone
  | ReqChar (pos,k) as req  →
      let pos_rel = pos − buffer_pos in
      let _ =
        (* we don't accumulate already emptied buffers. We could. *)
        assert (pos_rel >= 0)
      in
      if pos_rel < String.length buffer then
        (* desired char is already in buffer *)
        filler buffer buffer_pos (k (Some buffer.[pos_rel]))
      else
        (* buffer underflow. Ask the user to fill the buffer *)
        req
;;
val filler : string → int → stream_req → stream_req = <fun>
```

Синтаксис для $\lambda_{let}^{s/r}$

- Значения
  $v ::= c \mid x \mid \lambda x.e \mid \texttt{fix}\ f.x.e$
- Выражения
  $e ::= v \mid e_1 e_2 \mid \mathcal{S}k.e \mid \langle e \rangle \mid \text{let}\ x = e_1\ \texttt{in}\ e_2 \mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$
- Мономорфные типы
  $\alpha, \beta, \gamma, \delta ::= t \mid b \mid (\alpha/\gamma \to \beta/\delta)$
- Полиморфные типы
  $A ::= \alpha \mid \forall t.A$
- Evaluation context (e-context):
  $\mathsf{E} ::= [] \mid vE \mid Ee \mid \texttt{let}\ x = E\ \texttt{in}\ e \mid \texttt{if}\ E\ \texttt{then}\ e\ \texttt{else}\ e \mid \langle E \rangle$
- Pure e-context:
  $\mathsf{F} ::= [] \mid vF \mid Fe \mid \texttt{let}\ x = F\ \texttt{in}\ e \mid \texttt{if}\ F\ \texttt{then}\ e\ \texttt{else}\ e$
- RedEx:
  $\mathsf{R} ::= (\lambda x.e)v \mid \texttt{let}\ x = F\ \texttt{in}\ e \mid \texttt{if}\ F\ \texttt{then}\ e\ \texttt{else}\ e \mid \langle E \rangle \mid \langle F[\mathcal{S}k.e] \rangle$

Правила редукции для $\lambda^{s/r}_{let}$

$$
\begin{array}{rcl}
(\lambda x.e)v & \rightsquigarrow & e[v/x] \\
(\textbf{fix}\ f.x.e)v & \rightsquigarrow & e[\ \textbf{fix}\ f.x.e/f][v/x] \\
\langle v \rangle & \rightsquigarrow & v \\
\langle F[\mathcal{S}k.e] \rangle & \rightsquigarrow & \langle \textbf{let}\ k = \lambda x.\langle F[x] \rangle\ \textbf{in}\ e \rangle \\
\textbf{let}\ x = v\ \textbf{in}\ e & \rightsquigarrow & e[v/x] \\
\textbf{if true then}\ e_1\ \textbf{else}\ e_2 & \rightsquigarrow & e_1 \\
\textbf{if false then}\ e_1\ \textbf{else}\ e_2 & \rightsquigarrow & e_2
\end{array}
$$

Пример редукции в $\lambda_{let}^{s/r}$

$$
\begin{array}{ll}
 & \texttt{prefix [1;2]} \\
\rightsquigarrow & \langle 1 :: \mathcal{S}k.(k[] :: \langle k(\texttt{visit [2]})\rangle)\rangle \\
\rightsquigarrow & \langle \texttt{let k} = \lambda x.\langle 1 :: x\rangle \texttt{ in } k\texttt{[]} :: \langle k(\texttt{visit [2]})\rangle\rangle \\
\rightsquigarrow & \langle(\lambda x.\langle 1 :: x\rangle)\ \texttt{[]} :: \langle(\lambda x.\langle 1 :: x\rangle)(\texttt{visit [2]})\rangle\rangle \\
\rightsquigarrow^{+} & \langle \texttt{[1]} :: \langle(\lambda x.\langle 1 :: x\rangle)(2 :: \mathcal{S}k.(k\texttt{[]} :: (k(\texttt{visit []})\rangle))\rangle\rangle \\
\rightsquigarrow & \langle \texttt{[1]} :: \langle \texttt{let k = } \lambda x.\langle(\lambda x.\langle 1 :: x\rangle)(2 :: x)\rangle \texttt{ in} \\
 & \qquad k\texttt{[]} :: \langle k(\texttt{visit []})\rangle\rangle\rangle \\
\rightsquigarrow & \langle \texttt{[1]} :: \langle(\lambda x.\langle(\lambda x.\langle 1 :: x\rangle)(2 :: x)\rangle)\ \texttt{[] ::} \\
 & \qquad \langle(\lambda x.\langle(\lambda x.\langle 1 :: x\rangle)(2 :: x)\rangle)(\texttt{visit []})\rangle\rangle\rangle \\
\rightsquigarrow^{+} & \langle \texttt{[1]} :: \langle \texttt{[1;2]} :: \langle(\lambda x.\langle(\lambda x.\langle 1 :: x\rangle)(2 :: x)\rangle)(\mathcal{S}h.\texttt{[]})\rangle\rangle\rangle \\
\rightsquigarrow & \langle \texttt{[1]} :: \langle \texttt{[1;2]} :: \texttt{let } h = \lambda x.\langle(\lambda x.\langle 1 :: x\rangle)(2 :: x)\rangle)x\rangle \texttt{ in []}\rangle\rangle \\
\rightsquigarrow & \langle \texttt{[1]} :: \langle \texttt{[1;2]} :: \texttt{[]}\rangle\rangle \\
\rightsquigarrow^{+} & \texttt{[[1]; [1;2]]}
\end{array}
$$

Вывод типов

$$\Gamma, \alpha \vdash e : \tau, \beta$$

В контексте Γ выражение e имеет тип τ и процесс вычисления e изменяет answer type с α на β.

При CPS-преобразовании тип у образа e был бы $(\tau^* \to \alpha^*) \to \beta^*$.

При добавлении типов хотелось бы сохранить *type preservation property*: при вычислении выражения тип не должен меняться.

Система типов

Мономорфная система типов для `shift` и `reset` есть у Danvy & Filinski [DF89].

Полиморфизм туда можно добавлять разными способами, ограничивая полиморфизм выражения **let** `x` = e_1 **in** e_2.

- Value restriction: e_1 должно быть значением.
- "Слабые" типовые переменные: тип e_1 может быть обобщен (generalized) только когда он не связан с побочными эффектами.
- Полиморфизм по имени: вычисление e_1 откладывается до тех пор, когда x таки начнет использоваться в e_2, это приводит к call-by-name семантике для e_1.
- Pure выражения из [AK07] .

Pure выражения

Ограничим `let x =` e_1 `in` e_2, чтобы e_1 было чисто от эффектов управления, т.е. являлось *pure*.

Pure $\sim$ полиморфно по answer types.

$$\Gamma, \alpha \vdash e : \tau, \alpha$$

Примеры:

- значения
- $\langle e \rangle$, т.к. все эффекты отделены reset'ом.

Обозначать будем так: $\Gamma \vdash_p e : \tau$.

Правила вывода типов (1/2)

$A \succ \tau$: инстанциация полиморфных переменных из A какими-то мономорфными типами.

Gen$(\sigma, \Gamma) \sim \forall t_1 ... t_n . \sigma$, где $t_1 ... t_n = \mathrm{FTV}(\sigma) - \mathrm{FTV}(\Gamma)$.

$$\frac{(x : \sigma) \in \Gamma \qquad \sigma \succ \tau}{\Gamma \vdash_p x : \tau} \text{ (var)}$$

$$\frac{\Gamma \vdash_p M : \tau}{\Gamma, \alpha \vdash_p M : \tau, \alpha} \text{ (exp)}$$

$$\frac{\Gamma \vdash M_1 : \tau_1 \qquad \Gamma, x : \mathrm{Gen}(\tau_1, \Gamma), \alpha \vdash M_2 : \tau_2, \beta}{\Gamma, \alpha \vdash \mathrm{let\ x\ } = M_1 \mathrm{\ in\ } M_2 : \tau_2, \beta} \text{ (let)}$$

Правила вывода типов (2/2)

$$\frac{\Gamma, k : \forall t.(\tau/t \to \alpha/t), \gamma \vdash M : \gamma, \beta}{\Gamma, \alpha \vdash \mathcal{S}k.M : \tau, \beta} \text{ (shift)}$$

$$\frac{\Gamma, \gamma \vdash M : \gamma, \tau}{\Gamma, \alpha \vdash_p \langle M \rangle : \tau} \text{ (reset)}$$

$$\frac{\Gamma, x : \tau_1, \alpha \vdash M : \tau_2, \beta}{\Gamma \vdash_p \lambda x.M : (\tau_1/\alpha \to \tau_2/\beta)} \text{ (fun)}$$

$$\frac{\Gamma, \gamma \vdash M_1 : (\tau_1/\alpha \to \tau_2/\beta), \delta \qquad \Gamma, \beta \vdash M_2 : \tau_1, \gamma}{\Gamma, \alpha \vdash M_1 M_2 : \tau_2, \delta} \text{ (app)}$$

Свойства системы типов (1/2)

Subject reduction

Если и $\Gamma; \alpha \vdash e_1 : \tau; \beta$ выводимо, и $e_1 \rightsquigarrow^+ e_2$, тогда $\Gamma; \alpha \vdash e_2 : \tau; \beta$.
Аналогично, если $\Gamma \vdash_p e_1 : \tau$, то $\Gamma \vdash_p e_2 : \tau$

Слабая непротиворечивость системы типов (*weak type soundness*): правильно протипизированные программы работают хорошо.

Сильная непротиворечивость системы типов (*strong type soundness*): результат такого же типа, что исходный терм.

Прогресс и единственность разложения

Если выводится $\vdash_p \langle e \rangle : \tau$, то либо e просто значение, либо $\langle e \rangle$ можно единственным образом разложить в форму $E[R]$, где E – контекст, а R – RedEx.

Из двух свойств следует непротиворечивость(soundness).

Свойства системы типов (2/2)

$W' : (\Gamma, e) \mapsto (\theta; \alpha, \tau, \beta)$ как расширение HM.

Principal type и вывод типов

Можно построить алгоритм W' для $\lambda_{let}^{s/r}$ такой что

1. W' завершается
2. Если W' вернул $(\theta; \alpha, \tau, \beta)$, то $\Gamma\theta; \alpha \vdash e : \tau, \beta$ выводимо. Кроме этого, для любых таких $(\theta'; \alpha', \tau', \beta')$, что $\Gamma\theta'; \alpha' \vdash e : \tau', \beta'$ выводимо, верно $(\Gamma\theta'; \alpha', \tau', \beta') \equiv (\theta; \alpha, \tau, \beta)\phi$ для некоторой подстановки ϕ.
3. Если W' завершился с ошибкой, то $\Gamma\theta; \alpha \vdash e : \tau, \beta$ не выводимо ни для каких $(\theta; \alpha, \tau, \beta)$.

Confluence & strong normalization

1. Редукции $\rightsquigarrow$ в $\lambda_{let}^{s/r}$ не зависят от порядка.
2. Редукции $\rightsquigarrow$ в $\lambda_{let}^{s/r}$ без *fix* всегда завершаются.

Конец

Дальше только список литературы

Ссылки I

Kenichi Asai and Yukiyoshi Kameyama, *Polymorphic delimited continuations*, Programming Languages and Systems (Berlin, Heidelberg) (Zhong Shao, ed.), Springer Berlin Heidelberg, 2007, pp. 239–254.

Olivier Danvy and Andrzej Filinski, *A functional abstraction of typed contexts*, DIKU, University of Copenhagen (1989).

R. Kent Dyvbig, Simon Peyton Jones, and Amr Sabry, *A monadic framework for delimited continuations*, J. Funct. Program. **17** (2007), no. 6, 687–730.