Control structures, fourth lecture

Continuations and control operators: building blocks for control structures

Xavier Leroy

2024-02-15

Collège de France, chair of software sciences
`xavier.leroy@college-de-france.fr`

The notion of continuation

Given a control point in a program, its continuation is

> the sequence of computations that remain to be done
> once the execution reaches the given control point
> in order to finish the execution of the whole program.

Often, this continuation can be represented within the programming language, as a command or a function.

Examples of continuations

In an imperative language with structured control.

Program	*Continuation of...*
s_1 ❶ ; s_2	❶ s_2
(`if` be `then` ❶ s_1 `else` ❷ s_2); s_3	❶ $s_1; s_3$ ❷ $s_2; s_3$
`while` be `do` ❶ s ❷	❶ s; `while` be `do` s ❷ `while` be `do` s
`for` $i = 1$ `to` 10 `do` ❶ s	❶ s; `while` $i < 10$ `do` $(i = i + 1; s)$

Continuations in functional languages

In languages based on expressions, esp. functional languages, we talk about the continuation of a subexpression e in a program p:

> the continuation of e in p is
> the sequence of computations that remain to be done
> once e is evaluated to its value v_e
> to finish the evaluation and produce the value v_p of p.

The continuation can be viewed as the function $v_e \mapsto v_p$

Examples of continuations

In a language of arithmetic expressions,
with left-to-right evaluation.

Consider the program $p = (1 + 2) \times (3 + 4)$.

The continuation of 1 in p is $\lambda v.\ (v + 2) \times (3 + 4)$.

The continuation of $1 + 2$ in p is $\lambda v.\ v \times (3 + 4)$.

The continuation of $3 + 4$ in p is $\lambda v.\ 3 \times v$ (not $\lambda v.\ (1 + 2) \times v$).

Note that the continuation depends on the evaluation strategy!
(Right-to-left evaluation would result in different continuations.)

Continuations and jumps

Commands such as `goto`, `break`, `return`, or `throw` can be viewed as switching continuations: they continue not with the continuation of the control point that follows syntactically, but with

`goto` L	the continuation of the point labeled L
`break`	the continuation of the enclosing loop
`return`	the continuation of the current function invocation
`throw`	the continuation of the `catch` clause of the nearest `try`

Example: the continuation of `break` in

$$\texttt{while}\ \mathit{be}\ \texttt{do}\ (\texttt{break}; s_1); s_2$$

is s_2 and not $s_1; \texttt{while}\ \ldots; s_2$

In this lecture

Three ways to use continuations:

- as a semantic tool
  (esp. to give semantics to non-local `goto` statements);
- as a functional programming idiom
  writing programs in “continuation-passing style” (CPS);
- by adding control operators to the language
  (like `call/cc` in the Scheme language).

Continuations as a semantic tool

Denotational semantics

(C. Strachey, D. Scott, C. Wadsworth, etc, since 1965.)

Let's associate a mathematical object to each syntactic element of a programming language (expression, command, function, ...), describing its meaning with mathematical precision.

Example: for the language of spreadsheet, we define

$$[\![expr]\!] : \overbrace{(Var \overset{\text{fin}}{\to} Val)}^{\text{environment}} \to Val$$

$$[\![prog]\!] : \wp(Var \overset{\text{fin}}{\to} Val) \qquad \text{(set of solutions)}$$

by induction on the structure of *expr* and *prog*.

Denotational semantics of spreadsheets

Expressions: $[\![expr]\!](Var \xrightarrow{\text{fin}} Val) \to Val$

$$
\begin{aligned}
[\![cst]\!]\ \rho &= cst \\
[\![x]\!]\ \rho &= \rho(x) \\
[\![f(e_1, \ldots, e_n)]\!]\ \rho &= f^*([\![e_1]\!]\ \rho, \ldots, [\![e_n]\!]\ \rho)
\end{aligned}
$$

Programs: $[\![prog]\!] : \wp(Var \xrightarrow{\text{fin}} Val)$

$$[\![x_1 = e_1, \ldots, x_n = e_n]\!] = \{\rho \mid \rho(x_i) = [\![e_i]\!]\ \rho \text{ for } i = 1, \ldots, n\}$$

Denotational semantics of assignment

What is the meaning of assignments such as $x := x + 1$?

Idea: it's a store transformer (store = memory state).

$$[\![stmt]\!] : \overbrace{(Var \xrightarrow{\text{fin}} Val)}^{\text{store "before"}} \rightarrow \overbrace{(Var \xrightarrow{\text{fin}} Val)}^{\text{store "after"}}$$

Some representative cases:

$$[\![x := e]\!]\ \sigma = \sigma[x \leftarrow [\![e]\!]\ \sigma]$$

$$[\![s_1; s_2]\!]\ \sigma = [\![s_2]\!]\ ([\![s_1]\!]\ \sigma)$$

$$[\![\texttt{if}\ be\ \texttt{then}\ s_1\ \texttt{else}\ s_2]\!]\ \sigma = \begin{cases} [\![s_1]\!]\ \sigma & \text{if } [\![be]\!]\ \sigma = \texttt{true} \\ [\![s_2]\!]\ \sigma & \text{if } [\![be]\!]\ \sigma = \texttt{false} \end{cases}$$

Denotational semantics of loops

Idea: add a special denotation $\perp$ for divergence.

$$[\![stmt]\!] : \overbrace{(Var \xrightarrow{\text{fin}} Val)}^{\text{store ``before''}} \to \left( \overbrace{(Var \xrightarrow{\text{fin}} Val)}^{\text{store ``after''}} + \overbrace{\{\perp\}}^{\text{divergence}} \right)$$

We then define

$$[\![\texttt{while } be \texttt{ do } s]\!] = \mathsf{lfp}\,(\lambda d.\lambda\sigma.\texttt{if } [\![be]\!]\, s \texttt{ then } d([\![s]\!]\,\sigma) \texttt{ else } \sigma)$$

where “lfp” is the least fixed point of the given operator.

Denotational semantics of labels and jumps

(F. L. Morris, 1970; Wadsworth and Strachey, 1970; ...)

Idea: the denotation of a command takes as an explicit argument the continuation of this command. This makes it possible to capture the continuation of a label and to associate it to the label in an environment.

$$\llbracket stmt \rrbracket : Env \to \overset{\text{before } stmt}{\overset{\downarrow}{Store}} \to \underbrace{(\overset{\text{after } stmt}{\overset{\downarrow}{Store}} \to \overset{\text{final}}{Res})}_{\text{continuation}} \to \overset{\text{final}}{Res}$$

$$Store = Var \xrightarrow{\text{fin}} Val$$

$$Res = Store + \{\bot\}$$

$$Env = Label \xrightarrow{\text{fin}} (Store \to Res)$$

Continuation-based denotational semantics

For commands that terminate normally: the continuation is applied to the store after execution of the command, producing the final result of the program.

$$
\begin{aligned}
[\![x := e]\!]\,\rho\,\sigma\,k &= k\,(\sigma[x \leftarrow [\![e]\!]\,\sigma]) \\
[\![s_1; s_2]\!]\,\rho\,\sigma\,k &= [\![s_1]\!]\,\rho\,\sigma\,(\lambda\sigma'.\,[\![s_2]\!]\,\rho\,\sigma'\,k) \\
[\![\texttt{if}\ be\ \texttt{then}\ s_1\ \texttt{else}\ s_2]\!]\,\rho\,\sigma\,k &= \begin{cases} [\![s_1]\!]\,\rho\,\sigma\,k & \text{if } [\![be]\!]\,\sigma = \texttt{true} \\ [\![s_2]\!]\,\rho\,\sigma\,k & \text{if } [\![be]\!]\,\sigma = \texttt{false} \end{cases}
\end{aligned}
$$

Denotational semantics of labels and jumps

$\texttt{goto}\ L$ ignores the current continuation; instead, it restarts the continuation associated with L in the environment.

$$[\![\texttt{goto}\ L]\!]\ \rho\ \sigma\ k = \rho(L)\ \sigma$$

A definition of a label L associates the continuation of the definition with L in the environment.

$$\begin{aligned}[\![\texttt{begin}\ s_1;\ L : s_2\ \texttt{end}]\!]\ \rho\ \sigma\ k = {} & [\![s_1; s_2]\!]\ \rho'\ \sigma\ k \\ & \text{where } \rho' = \rho[L \leftarrow k_2] \\ & \text{and } k_2 = \lambda\sigma'.\ [\![s_2]\!]\ \rho'\ \sigma'\ k\end{aligned}$$

Reduction strategies for a functional language

In lecture #3, we saw the need for defining and enforcing the reduction strategy used to execute functional languages:

- Call by value: the function argument is reduced to a value before being substituted in the function body.
- Call by name: the function argument is substituted unevaluated in the function body. It will be evaluated every time the function needs its value.
- Call by need (“lazy evaluation”):
  like call by name, but evaluations are memoized. The argument is evaluated the first time its value is needed, and the value is reused if it is needed again later.

Denotational semantics for a functional language

Naively:

$$
\begin{aligned}
Val &= Num + (Val \to Val) + \{\bot\} \\
[\![expr]\!] &: (Var \overset{\text{fin}}{\to} Val) \to Val \\
[\![x]\!]\ \rho &= \rho(x) \\
[\![\lambda x.\ e]\!]\ \rho &= v \mapsto [\![e]\!]\ (\rho[x \leftarrow v]) \\
[\![e_1\ e_2]\!]\ \rho &= ([\![e_1]\!]\ \rho)\ ([\![e_2]\!]\ \rho)
\end{aligned}
$$

Problem 1: *Val* is ill-defined in set theory (cardinality issue).

Problem 2: it is not apparent which strategy is being implemented by the semantic function application $([\![e_1]\!]\ \rho)\ ([\![e_2]\!]\ \rho)$.

Using Scott domains

Call by name:

$$Res \approx Num + Fun + \{\bot\} + \{err\} \text{ and } Fun = Res \stackrel{\text{cont}}{\to} Res$$

$$[\![e_1\ e_2]\!]\ \rho = \begin{cases} ([\![e_1]\!]\ \rho)\ ([\![e_2]\!]\ \rho) & \text{if } [\![e_1]\!]\ \rho \in Fun \\ \bot & \text{if } [\![e_1]\!]\ \rho = \bot \\ err & \text{otherwise} \end{cases}$$

Call by value:

$$Res \approx Val + \{\bot\} + \{err\} \text{ and } Val \approx Num + Fun \text{ and } Fun = Val \stackrel{\text{cont}}{\to} Res$$

$$[\![e_1\ e_2]\!]\ \rho = \begin{cases} ([\![e_1]\!]\ \rho)\ ([\![e_2]\!]\ \rho) & \text{if } [\![e_1]\!]\ \rho \in Fun \text{ and } [\![e_2]\!] \in Val \\ \bot & \text{if } [\![e_1]\!]\ \rho = \bot \text{ or } [\![e_1]\!]\ \rho \in Fun \text{ and } [\![e_2]\!]\ \rho = \bot \\ err & \text{otherwise} \end{cases}$$

The CPS transformation

Specifying a reduction strategy using continuations

To make explicit the reduction strategy, we could add (semantic) continuations to the denotational semantics of a functional language.

However, a functional language has enough expressive power to enable continuations to be materialized at the syntax level, by a program transformation:

$$\text{functional language} \rightarrow \text{“CPS fragment” of the language}$$

The CPS transformation

The transform of an expression e is a function $\lambda k \ldots$ that:

- takes as argument a function k (the continuation);
- reduces e to a value v (following a given strategy);
- finishes by applying k to v (tail call).

The resulting function is in continuation-passing style (CPS).

CPS transformation for call by value

$$\begin{aligned}
\mathcal{V}(cst) &= \lambda k.\, k\; cst \\
\mathcal{V}(x) &= \lambda k.\, k\; x \\
\mathcal{V}(\lambda x.\, e) &= \lambda k.\, k\; (\lambda x.\, \mathcal{V}(e)) \\
\mathcal{V}(e_1\; e_2) &= \lambda k.\, \mathcal{V}(e_1)\; (\lambda v_1.\, \mathcal{V}(e_2)\; (\lambda v_2.\, v_1\; v_2\; k))
\end{aligned}$$

Variables are bound to values, hence $\mathcal{V}(x) = \lambda k.\, k\; x$.

Evaluation of an application $e_1\; e_2$:
evaluate e_1 to v_1, then evaluate e_2 en v_2, then apply v_1 to v_2.

CPS transformation for call by name

$$
\begin{aligned}
\mathcal{N}(cst) &= \lambda k.\, k\ cst \\
\mathcal{N}(x) &= \lambda k.\, x\ k \\
\mathcal{N}(\lambda x.\, e) &= \lambda k.\, k\ (\lambda x.\, \mathcal{N}(e)) \\
\mathcal{N}(e_1\ e_2) &= \lambda k.\, \mathcal{N}(e_1)\ (\lambda v_1.\, v_1\ (\mathcal{N}(e_2))\ k))
\end{aligned}
$$

Variables are bound to suspended computations, hence $\mathcal{N}(x) = \lambda k.\, x\ k$ or just $\mathcal{N}(x) = x$.

Evaluation of an application $e_1\ e_2$: evaluate e_1 to v_1, then apply v_1 to the suspended computation $\mathcal{N}(e_2)$.

Administrative reductions

CPS transformations produce terms that are more verbose than we would write by hand. In the case of an application of a variable to a variable, we get

$$\mathcal{V}(f\ x) = \lambda k.\ (\lambda k_1.k_1\ f)\ (\lambda v_1.\ (\lambda k_2.k_2\ x)\ (\lambda v_2.\ v_1\ v_2\ k))$$

instead of just $\lambda k.\ f\ x\ k$.

This can be avoided by performing "administrative reductions" $\stackrel{adm}{\to}$ on the result of the CPS transformation:
these are β-reductions that remove the "administrative redexes" introduced by the translation. In particular, we can do

$$(\lambda k.\ k\ v)\ (\lambda x.\ a) \stackrel{adm}{\to} (\lambda x.\ a)\ v \stackrel{adm}{\to} a[x \leftarrow v]$$

whenever v is a value or a variable.

Examples of CPS transformations (after administrative reductions)

$$\mathcal{V}(f(g\,x)) \\ = \lambda k.\, g\, x\, (\lambda v.\, f\, v\, k)))$$

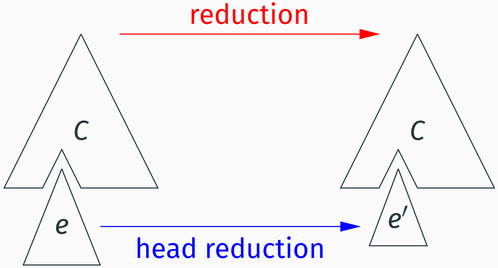
$$\mathcal{N}(f(g\,x)) \\ = \lambda k.\, f\, (\lambda v.\, v\, (\lambda k'.\, g\, (\lambda v'.\, v'\, x\, k'))\, k)$$

$$\mathcal{V}(\texttt{let rec } \mathit{fact} = \lambda n.\, \texttt{if } n = 0 \texttt{ then } 1 \texttt{ else } n * \mathit{fact}(n-1)) \\ = \texttt{let rec } \mathit{fact} = \lambda n.\, \lambda k. \\ \texttt{if } n = 0 \texttt{ then } k\, 1 \texttt{ else } \mathit{fact}\, (n-1)\, (\lambda v.\, k\, (n * v)))$$

Specifying a reduction strategy using operational semantics

As a set of head reductions $e \xrightarrow{\varepsilon} e'$
and a set of reduction contexts C.

$$\frac{e \xrightarrow{\varepsilon} e'}{C[e] \rightarrow C[e']}$$



The usual strategies

Weak lambda-calculus: we can β-reduce anywhere but under a λ.

$$(\lambda x.\, e)\; e' \xrightarrow{\varepsilon} e\{x \leftarrow e'\}$$

$$C ::= [\,] \mid C\; e \mid e\; C$$

Call by name: no reductions in arguments to applications.

$$(\lambda x.\, e)\; e' \xrightarrow{\varepsilon} e\{x \leftarrow e'\}$$

$$C ::= [\,] \mid C\; e$$

Call by value: left-to-right reduction of applications;
β-reduction restricted to values $v ::= cst \mid \lambda x.\, e$.

$$(\lambda x.\, e)\; v \xrightarrow{\varepsilon} e\{x \leftarrow v\}$$

$$C ::= [\,] \mid C\; e \mid v\; C$$

Semantic correctness of CPS transformation

(G. Plotkin, *Call-by-name, call-by-value and the lambda-calculus*, TCS 1(2), 1975)

Executing a program e after CPS transformation CPS consists in applying $\mathcal{V}(e)$ or $\mathcal{N}(e)$ to the initial continuation $\lambda x.\, x$.

Theorem

If $e \xrightarrow{*} cst$ (resp. e diverges) in call by value, then $\mathcal{V}(e)\ (\lambda x.\, x) \xrightarrow{*} cst$ (resp. $\mathcal{V}(e)\ (\lambda x.\, x)$ diverges).

If $e \xrightarrow{*} cst$ (resp. e diverges) in call by name, then $\mathcal{N}(e)\ (\lambda x.\, x) \xrightarrow{*} cst$ (resp. $\mathcal{N}(e)\ (\lambda x.\, x)$ diverges) .

Plotkin's proof

A difficult proof, relying on this simulation diagram:

$$\begin{tikzcd}[column sep=huge, row sep=large]
a \arrow[r, "\text{one CBV reduction}"] \arrow[d, "\text{CPS transformation}"'] & b \arrow[d, "\text{CPS transformation}"] \\
\mathcal{V}(a)\ k \arrow[r, dotted, no head, "\text{(no reductions, in general)}"] \arrow[d, "\text{administrative reductions}"'] & \mathcal{V}(b)\ k \arrow[d, "\text{administrative reductions}"] \\
a : k \arrow[ur, "\text{weak reductions}"] & b : k
\end{tikzcd}$$

$a : k$, the *colon translation*, is $\mathcal{V}(a)\ k$ where some administrative redexes were reduced.

CPS terms

Terms produced by the CPS transformation have a very specific shape, described by the following grammar:

Atoms:	$a ::= x \mid cst \mid \lambda v.\, b \mid \lambda x. \lambda k.\, b$
Function bodies:	$b ::= a \mid a_1\ a_2 \mid a_1\ a_2\ a_3$

$\mathcal{V}(e)$ is an atom, and $\mathcal{V}(e)\ (\lambda x.\, x)$ is a body.

Function applications (to 1 or 2 arguments) are always in tail position.

Reducing CPS terms

Atoms: $a ::= x \mid cst \mid \lambda v.\ b \mid \lambda x.\lambda k.\ b$

Function bodies: $b ::= a \mid a_1\ a_2 \mid a_1\ a_2\ a_3$

Theorem (Indifference to the evaluation order (Plotkin, 1975))

A CPS-transformed program evaluates identically in call by name, in call by value, and in any weak reduction strategy.

Proof.

Starting from $\mathcal{V}(e)\ (\lambda x.x)$, all reducts are closed bodies b, i.e. v or $v_1\ v_2$ or $v_1\ v_2\ v_3$. The only reductions possible in any weak strategy are

$(\lambda x.b)\ v_2 \to b[x \leftarrow v_2]$

$(\lambda x.\lambda k.b)\ v_2\ v_3 \to (\lambda k.b)[x \leftarrow v_2]\ v_3 \to b[x \leftarrow v_2, k \leftarrow v_3]$. □

Programming in continuation-passing style

Programming CPS

When writing code in a functional language, it can be useful to perform the CPS transformation manually on selected parts of the program.

This makes it possible to pass explicitly the continuation of a call to a library function. This function can use the continuation to implement advanced control structures: iterators, coroutines, cooperative threads, ...

“Internal” iteration on a binary tree

```
type 'a tree = Leaf | Node of 'a tree * 'a * 'a tree
```

The usual “internal” iterator in OCaml:

```
let rec tree_iter (f: 'a -> unit) (t: 'a tree) =
  match t with
  | Leaf -> ()
  | Node(l, x, r) -> tree_iter f l; f x; tree_iter f r
```

The same, partially transformed to CPS:

```
let rec tree_iter f t (k: unit -> unit) =
  match t with
  | Leaf -> k ()
  | Node(l, x, r) ->
      tree_iter f l (fun () -> f x; tree_iter f r k)
```

Benefit (?): the recursive traversal runs in constant stack space.

Towards an “external” iterator

A general data type to evaluate sequences of values on demand:

```
type 'a enum = Done | More of 'a * (unit -> 'a enum)
```

(See also: the type `Seq.t` in the OCaml standard library.)

Application: “external” iteration on a binary tree.

```
let rec tree_iter (t: 'a tree) (k: unit -> 'a enum) =
  match t with
  | Leaf -> k ()
  | Node(l, x, r) ->
      tree_iter l (fun () -> More(x, tree_iter r k))

let tree_iterator (t: 'a tree) : 'a enum =
  tree_iter t (fun () -> Done)
```

Using external iterators

The “same fringe problem” mentioned in lecture #2.

```
let same_enums (e1: 'a enum) (e2: 'a enum) : bool =
  match e1, e2 with
  | Done, Done -> true
  | More(x1, k1), More(x2, k2) ->
      x1 = x2 && same_enums (k1 ()) (k2 ())
  | _, _ -> false

let same_fringe (t1: 'a tree) (t2: 'a tree) : bool =
  same_enums (tree_iterator t1) (tree_iterator t2)
```

A Python-style stateful generator

By adding local mutable state, this iterator becomes a Python-style generator that returns the next value in the enumeration at each call.

```
exception StopIteration

let tree_generator (t: 'a tree) : unit -> 'a =
  let current = ref (fun () -> tree_iterator t) in
  fun () ->
    match !current () with
    | Done -> raise StopIteration
    | More(x, k) -> current := k; x
```

A library of cooperative threads

The natural interface (in "direct style"):

`spawn: (unit -> unit) -> unit`

Start a new thread.

`yield: unit  -> unit`

Suspend the current thread;
switch to another runnable thread.

`terminate: unit -> unit`

Stop the current thread forever.

A library of cooperative threads

The CPS interface (with an explicit continuation):

`spawn: (unit -> unit) -> unit`

Start a new thread.

`yield: (unit -> unit) -> unit`

Suspend the current thread;
switch to another runnable thread.

`terminate: unit -> unit`

Stop the current thread forever.

Implementing the library

A queue of runnable threads (suspended, but ready to restart).

```
let ready : (unit -> unit) Queue.t = Queue.create ()

let terminate () =
  match Queue.take_opt ready with
  | None -> ()
  | Some k -> k ()

let yield (k: unit -> unit) =
  Queue.add k ready; terminate()

let spawn (f: unit -> unit) =
  Queue.add f ready
```

Example of use

Print integers from 1 to `count`, yielding at every number:

```
let process name count =
  let rec proc n =
    if n > count then terminate () else begin
      printf "%s%d " name n;
      yield (fun () -> proc (n + 1))
    end
  in proc 1
```

Example of use:

```
let () =
  spawn (fun () -> process "a" 5);
  spawn (fun () -> process "b" 3);
  process "c" 6
```

(Prints `c1 a1 b1 c2 a2 b2 c3 a3 b3 c4 a4 c5 a5 c6`.)

Backtracking with continuations

A continuation can be invoked several times. This can be useful to implement backtracking.

Example: matching regular expressions.

```
type regexp = char list -> (char list -> bool) -> bool
```

The “contract” for a regular expression R:
$R\ \ell\ k$ invokes $k\ l_2$ if $l = l_1.l_2$ and l_1 matches R;
$R\ \ell\ k$ returns `false` if no prefix of l matches R.

In the first case, the continuation k can itself return `false` to signal that it did not match l_2.

```
let string_match (r: bool regexp) (l: char list) : bool =
  r l (fun l' -> l' = [])
```

Definition of the usual regular expressions

```
let epsilon : regexp = fun l k -> k l

let char (c: char) : regexp = fun l k ->
  match l with c' :: l' when c' = c -> k l' | _ -> false

let seq (r1: regexp) (r2: regexp) = fun l k ->
  r1 l (fun l' -> p2 l' k)

let alt (r1: regexp) (r2: regexp) = fun l k ->
  r1 l k || r2 l k

let rec star (r: regexp) : regexp = fun l k ->
  alt (seq r (star r)) epsilon l k

and plus (r: regexp) : regexp = fun l k ->
  seq r (star r) l k
```

“Internal generators” and counting

An “internal generator” = a function that produces several possible results, gives them in turn to a continuation *k*, and combines the results returned by *k*.

```
let bool k = k false + k true

let rec int lo hi k =
  if lo <= hi then k lo + int (lo + 1) hi k else 0

let rec avltree h k =
  if h < 0 then 0 else if h = 0 then k Leaf else
    avltree2 (h-1) (h-1) k
  + avltree2 (h-2) (h-1) k
  + avltree2 (h-1) (h-2) k
and avltree2 hl hr k =
  avltree hl (fun l -> avltree hr (fun r -> k (Node(l, 0, r))))
```

“Internal generators” and counting

The continuation *k* plays the role of a measure: it says how much each possibility contributes to the total.

Ex: counting AVL trees of height 4.

```
let n = avltree 4 (fun _ -> 1)
(* 315 *)
```

Ex: counting dice throws $\geq$ 16.

```
let _3d6 k =
  int 1 6 (fun d1 ->
  int 1 6 (fun d2 ->
  int 1 6 (fun d3 -> k (d1,d2,d3))))
let n = _3d6 (fun (d1,d2,d3) ->
                if d1+d2+d3 >= 16 then 1 else 0)
(* 10 *)
```

Control operators

Control operators

Constructs provided by some functional languages
enabling an expression to reify its continuation,
manipulate it as a first-class value,
and restart this continuation later.

Control operators make it possible to program one's own control structures without using CPS, keeping the program in "direct style".

ISWIM, Algol, and operator J

(P. J. Landin, *The next 700 programming languages*, CACM 9, 1966.)
(P. J. Landin, *Correspondence between ALGOL 60 and Church's Lambda-notation*, CACM 8, 1965.)

The ISWIM language: a precursor to Scheme and ML.

- Extended lambda-calculus with call by value.
- Operational semantics given via the SECD abstract machine.
- Static scoping of variables ($\neq$ Lisp), implemented using closures.

An explanation of Algol by translation to extended ISWIM:

- Mutable state $\rightarrow$ adding ML-style references.
- Non-local "goto" $\rightarrow$ adding the J control operator.

The $\texttt{J}$ control operator

The evaluation of $\texttt{J}(\lambda y.\, e')\ v$ computes the value of $e'\{y \leftarrow v\}$ and returns it directly to f's caller, "jumping over" the remaining computations in the body of f.

Special case: $\texttt{J}\ (\lambda x.\, x)\ v$ behaves like `return` v in C.

Using $\texttt{J}$ to encode labels and `goto`:

$$\begin{array}{rcl} \texttt{begin}\ s_1;\ L : s_2\ \ \texttt{end} & \rightsquigarrow & \lambda\_.\, \texttt{let rec}\ L = \texttt{J}\ (\lambda\_.\, s_2)\ \texttt{in}\ s_1; L\ (\,) \\ \texttt{goto}\ L & \rightsquigarrow & L\ (\,) \end{array}$$

The `callcc` operator (*call with current continuation*)

$$\texttt{callcc}\ (\lambda k.\, e)$$

A construct of the Scheme language that captures its own continuation, turns it into a function, and passes it to $\lambda k.\, e$.

Appears in the literature under various names:

- J. Reynolds, 1972: `escape`.
- G. Sussmann and G. Steele, 1975: `catch` and `throw`.
- The Scheme language, from 1982: `call-with-current-continuation`, shortened as `call/cc`.

Execution of `callcc`

The expression $\texttt{callcc}(\lambda k.\, e)$ evaluates as follows:

- The continuation of this expression is bound to variable k.
- e is evaluated; its value is the value of $\texttt{callcc}(\lambda k.\, e)$.
- If, during the evaluation of e or at any later time, k is applied to a value v, evaluation continues as if $\texttt{callcc}(\lambda k.\, e)$ had returned value v.

  In other words, the continuation of the `callcc` expression is restored and resumed with v as the value of this expression.

From an “internal” iterator to an “external” iterator

Assume given an “internal” iterator such as the following one for binary trees:

```
type 'a tree = Leaf | Node of 'a tree * 'a * 'a tree

let rec tree_iter (f: 'a -> unit) (t: 'a tree) =
  match t with
  | Leaf -> ()
  | Node(l, x, r) -> tree_iter f l; f x; tree_iter f r
```

From an “internal” iterator to an “external” iterator

Using `callcc`, we can stop the traversal as soon as `tree_iter` found one element, and return this element:

```
let tree_iterator (t: 'a tree) : 'a enum =
  callcc (fun k ->
      tree_iter
          (fun x -> k (Some x))
          t;
      None)
```

The call `k (Some x)` stops the traversal and causes `Some x` to be returned as result of `callcc`.

If the tree is empty, the continuation `k` is not called and `callcc` returns `None` as a result.

From an "internal" iterator to an "external" iterator

Using two `callcc`, we can define an "external" iterator (enumerating all elements of the tree on demand) on top of `tree_iter`.

```
type 'a enum = Done | More of 'a * (unit -> 'a enum)

let tree_iterator (t: 'a tree) : 'a enum =
  callcc (fun k ->
      tree_iter
          (fun x -> callcc (fun k' -> k (More(x, k'))))
          t;
      Done)
```

If x_1 is the leftmost element of t, `tree_iterator` t returns `More`(x_1, k_1). When k_1 is called, the traversal restarts where it left, and moves to the next element of t, or terminates.

Implementing structured exceptions with `callcc`

Using an imperative stack of exception handlers.

```
let handlers : (exn -> unit) Stack.t = Stack.create()

let raise exn =
  match Stack.pop_opt handlers with
  | Some hdlr -> hdlr exn
  | None -> fatal_error "uncaught exception"

let trywith body hdlr =
    callcc (fun k ->
        Stack.push (fun e -> k (hdlr e)) handlers;
        let res = body () in
        Stack.drop handlers;
        res)
```

Implementing structured exceptions with `callcc`

The construct

$$\texttt{try } e \texttt{ with } p_1 \rightarrow e_n \mid \ldots \mid p_n \rightarrow e_n$$

translates into

```
trywith
    (fun () -> e)
    (fun exn ->
       match exn with
       | p₁ -> e₁ | ... | pₙ -> eₙ
       | _ -> raise exn)
```

Implementing advanced control structure in direct style

Adding control operators such as `callcc` to a functional language

- make it possible to implement advanced control structures as libraries (coroutines, exceptions, cooperative threads, ...),
- while keeping the main program written in "direct style" (no CPS conversion required).

Semantics and implementation of `callcc`

Semantics:

- by CPS transformation;
- directly, using reduction contexts.

Implementation:

- by CPS transformation on the whole program;
- using multiple call stacks
  (capturing the current continuation = stack copy;
  restarting a captured continuation = stack switching)
- using a persistent data structure to represent the call stack
  ($\rightarrow$ 2022-2023 course).

CPS transformation for `callcc`

$$\mathcal{V}(\texttt{callcc}\ \ f) = \lambda k.\ \mathcal{V}(f)\ (\texttt{resume}\ k)\ k$$

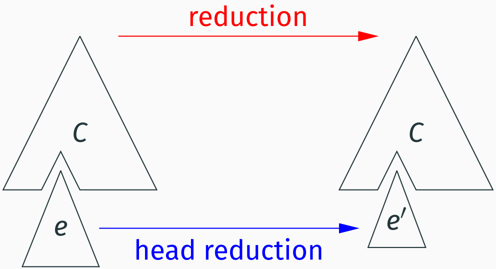
$$\texttt{resume}\ \ k_0 = \lambda v.\ \lambda k.\ k_0\ v$$

The standard CPS transformation uses continuations linearly: every k parameter is used exactly once.

For `callcc` f, we duplicate the continuation k:
it is used once as argument to f (within `resume` k),
and once as continuation for f.

For `resume` k_0, we ignore its continuation k:
execution continues with k_0.

Continuations and reduction contexts



Consider a program p that decomposes as $p = C[e]$, where C is a reduction context and e can head-reduce.

Then, the continuation of e in p is exactly $\lambda v.\ C[v]$, that is, the context C reified as a function. (v not bound in C)

Reduction rules for `callcc`

$$
\begin{array}{rcl}
C[\texttt{callcc}(\lambda k.\, e)] & \rightarrow & C[(\lambda k.\, e)\ (\lambda v.\, \texttt{resume}\ C\ v)] \\
C[\texttt{resume}\ C_0\ v] & \rightarrow & C_0[v]
\end{array}
$$

These are not head-reductions under a context $\xrightarrow{\varepsilon}$,
but whole-program reductions $\rightarrow$.

The rule for `callcc` duplicates the current context C.

The rule for `resume` replaces it by the captured context C_0.

Delimited continuations

Continuations captured by `callcc` are undelimited and abortive: they execute to the end of the program and never return.

For some applications (backtracking, counting), we need continuations that are delimited and composable. For example:

$$\begin{aligned}
& 2 \times \texttt{delim}\,(1 + \texttt{capture}\,(\lambda k.\, k(k\,0))) \\
\xrightarrow{+}\quad & 2 \times (\texttt{let}\ \ k = \lambda v.\, 1 + v\ \ \texttt{in}\ \ k(k\,0)) \\
\xrightarrow{+}\quad & 2 \times ((1 + (1 + 0))) \xrightarrow{+} 4
\end{aligned}$$

(The captured continuation "goes from `capture` to `delim`".)

(Additional benefit: delimited continuations are smaller than undelimited continuations, so capturing them can be less costly.)

Semantics of delimited continuations

Head-reduction rules under a context C, but with a sub-context D that does not mention `delim`:

$$\frac{\mathtt{delim}(D[\mathtt{capture}\ (\lambda k.\, e)]) \xrightarrow{\varepsilon} \ldots}{C[\mathtt{delim}(D[\mathtt{capture}\ (\lambda k.\, e)])] \rightarrow C[\ldots]}$$

Head reductions: (4 variants!)

$$\begin{aligned} \mathtt{delim}(D[\mathtt{capture}\ (\lambda k.\, e)]) &\xrightarrow{\varepsilon} (\lambda k.\, e)\ (\lambda v.\, \mathtt{resume}\ D\ v) \\ \mathtt{resume}\ D\ v &\xrightarrow{\varepsilon} D[v] \end{aligned}$$

Variant: `-ctrl-`

Semantics of delimited continuations

Head-reduction rules under a context C, but with a sub-context D that does not mention `delim`:

$$\frac{\texttt{delim}(D[\texttt{capture}\ (\lambda k.\, e)]) \xrightarrow{\varepsilon} \ldots}{C[\texttt{delim}(D[\texttt{capture}\ (\lambda k.\, e)])] \to C[\ldots]}$$

Head reductions: (4 variants!)

$$\begin{aligned} \texttt{delim}(D[\texttt{capture}\ (\lambda k.\, e)]) &\xrightarrow{\varepsilon} (\lambda k.\, e)\ (\lambda v.\, \texttt{resume}\ D\ v) \\ \texttt{resume}\ D\ v &\xrightarrow{\varepsilon} \texttt{delim}(D[v]) \end{aligned}$$

Variant: `-ctrl-`, `-ctrl+`

Semantics of delimited continuations

Head-reduction rules under a context C, but with a sub-context D that does not mention `delim`:

$$\frac{\texttt{delim}(D[\texttt{capture}\ (\lambda k.\, e)]) \xrightarrow{\varepsilon} \ldots}{C[\texttt{delim}(D[\texttt{capture}\ (\lambda k.\, e)])] \rightarrow C[\ldots]}$$

Head reductions: (4 variants!)

$$\begin{aligned} \texttt{delim}(D[\texttt{capture}\ (\lambda k.\, e)]) &\xrightarrow{\varepsilon} \texttt{delim}((\lambda k.\, e)\ (\lambda v.\, \texttt{resume}\ D\ v)) \\ \texttt{resume}\ D\ v &\xrightarrow{\varepsilon} D[v] \end{aligned}$$

Variant: `-ctrl-`, `-ctrl+`, `+ctrl-`

Semantics of delimited continuations

Head-reduction rules under a context C, but with a sub-context D that does not mention `delim`:

$$\frac{\texttt{delim}(D[\texttt{capture}\ (\lambda k.\, e)]) \xrightarrow{\varepsilon} \ldots}{C[\texttt{delim}(D[\texttt{capture}\ (\lambda k.\, e)])] \to C[\ldots]}$$

Head reductions: (4 variants!)

$$\begin{array}{rcl} \texttt{delim}(D[\texttt{capture}\ (\lambda k.\, e)]) & \xrightarrow{\varepsilon} & \texttt{delim}((\lambda k.\, e)\ (\lambda v.\, \texttt{resume}\ D\ v)) \\ \texttt{resume}\ D\ v & \xrightarrow{\varepsilon} & \texttt{delim}(D[v]) \end{array}$$

Variant: `-ctrl-`, `-ctrl+`, `+ctrl-`, `+ctrl+`.

A menagerie of delimited control operators

(D. Hillerström, citation in references.)

Name	**Taxonomy**	**Continuation behaviour**	**Canonical reference**
control/prompt	+**ctrl**−	Composable	Felleisen [81]
shift/reset	+**ctrl**+	Composable	Danvy and Filinski [62]
spawn	−**ctrl**+	Composable	Hieb and Dybvig [116]
splitter	−**ctrl**−	Abortive, composable	Queinnec and Serpette [234]
fcontrol	−**ctrl**−	Composable	Sitaram [250]
cupto	−**ctrl**−	Composable	Gunter et al. [111]
catchcont	−**ctrl**−	Composable	Longley [177]
effect handlers	−**ctrl**+	Composable	Plotkin and Pretnar [228]

Table A.2: Classification of first-class delimited control operators (listed in chronological order).

Summary

Summary

Continuations are a powerful concept

- to understand and formalize the semantics of non-local jumps;
- to program in functional languages with full control over the ordering and interleaving of computations
  - in continuation-passing style
  - or in direct style, using control operators.

See also: the seminar talks by Andrew Kennedy (22/02) and Olivier Danvy (29/02).

See also: lectures #5 and #6 on effect handlers, a modern, elegant presentation of delimited control.

References

References

Programming with continuations:

- Daniel P. Friedman and Mitchell Wand, *Essentials of Programming Languages*, MIT Press, 2008. Chapters 5 and 6.

The menagerie of control operators:

- Daniel Hillerström, *Foundations for Programming and Implementing Effect Handlers*, PhD, Edinburgh, 2021. Appendix A, *Continuations*.

A history of the notion of continuation:

- John C. Reynolds, *The Discoveries of Continuations*, LISP and Symbolic Computation 6(3–4), 1993.