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
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.

<table>
<thead>
<tr>
<th>Program</th>
<th>Continuation of...</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_1 ; s_2$</td>
<td>$1 ; s_2$</td>
</tr>
<tr>
<td>$(\text{if } be \text{ then } s_1 \text{ else } s_2); s_3$</td>
<td>$1 ; s_1; s_3$; $2 ; s_2; s_3$</td>
</tr>
<tr>
<td>while $be$ do $s_2$</td>
<td>$1 ; s; \text{while } be \text{ do } s$; $2 ; \text{while } be \text{ do } s$</td>
</tr>
<tr>
<td>for $i = 1$ to $10$ do $s$</td>
<td>$1 ; s; \text{while } i &lt; 10 \text{ do } (i = i + 1; s)$</td>
</tr>
</tbody>
</table>
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

```plaintext
while be do (break; s₁); s₂
```

is `s₂` and not `s₁`; while `...; s₂`
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
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

\[
\begin{align*}
\llbracket expr \rrbracket : (\text{Var}^{\text{fin}} \rightarrow \text{Val}) & \rightarrow \text{Val} \\
\llbracket prog \rrbracket : \wp(\text{Var}^{\text{fin}} \rightarrow \text{Val}) & \quad (\text{set of solutions})
\end{align*}
\]

by induction on the structure of \emph{expr} and \emph{prog}. 

(C. Strachey, D. Scott, C. Wadsworth, etc, since 1965.)
Denotational semantics of spreadsheets

Expressions:

\[ [\text{expr}] : (\text{Var} \xrightarrow{\text{fin}} \text{Val}) \rightarrow \text{Val} \]

\[ [\text{cst}] \rho = \text{cst} \]

\[ [x] \rho = \rho(x) \]

\[ [f(e_1, \ldots, e_n)] \rho = f^*([e_1] \rho, \ldots, [e_n] \rho) \]

Programs:

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

\[ [[x_1 = e_1, \ldots, x_n = e_n]] = \{ \rho \mid \rho(x_i) = [e_i] \rho \text{ for } i = 1, \ldots, n \} \]
Denotational semantics of assignment

What is the meaning of assignments such as \( x := x + 1 \)?

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

\[
\begin{align*}
\llbracket stmt \rrbracket : (\text{Var} \xrightarrow{\text{fin}} \text{Val}) & \rightarrow (\text{Var} \xrightarrow{\text{fin}} \text{Val}) \\
\text{store “before”} & \quad \text{store “after”}
\end{align*}
\]

Some representative cases:

\[
\begin{align*}
\llbracket x := e \rrbracket \sigma &= \sigma[x \leftarrow \llbracket e \rrbracket \sigma] \\
\llbracket s_1; s_2 \rrbracket \sigma &= \llbracket s_2 \rrbracket (\llbracket s_1 \rrbracket \sigma) \\
\llbracket \text{if } be \text{ then } s_1 \text{ else } s_2 \rrbracket \sigma &= \begin{cases} 
\llbracket s_1 \rrbracket \sigma & \text{if } \llbracket be \rrbracket \sigma = \text{true} \\
\llbracket s_2 \rrbracket \sigma & \text{if } \llbracket be \rrbracket \sigma = \text{false}
\end{cases}
\end{align*}
\]
Denotational semantics of loops

Idea: add a special denotation \( \perp \) for divergence.

\[
\begin{align*}
\llbracket \text{stmt} \rrbracket : (\text{Var} \xrightarrow{\text{fin}} \text{Val}) & \rightarrow \\
& \left( (\text{Var} \xrightarrow{\text{fin}} \text{Val}) + \{ \perp \} \right)
\end{align*}
\]

We then define

\[
\llbracket \text{while } \text{be} \text{ do } s \rrbracket = \text{lfp}(\lambda d. \lambda \sigma. \text{if } \llbracket \text{be} \rrbracket s \text{ then } d(\llbracket s \rrbracket \sigma) \text{ else } \sigma)
\]

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.

\[
\begin{align*}
\llbracket \text{stmt} \rrbracket : & \ Env \to Store \to (Store \to Res) \to Res \\
\text{Store} = & \ Var \xrightarrow{\text{fin}} Val \\
\text{Res} = & \ Store + \{\bot\} \\
\text{Env} = & \ Label \xrightarrow{\text{fin}} (Store \to Res)
\end{align*}
\]
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.

\[
[x := e] \rho \sigma k = k (\sigma[x \leftarrow [e] \sigma])
\]

\[
[s_1; s_2] \rho \sigma k = [s_1] \rho \sigma (\lambda \sigma'. [s_2] \rho \sigma' k)
\]

\[
[\text{if } be \text{ then } s_1 \text{ else } s_2] \rho \sigma k = \begin{cases} 
[s_1] \rho \sigma k & \text{if } [be] \sigma = \text{true} \\
[s_2] \rho \sigma k & \text{if } [be] \sigma = \text{false}
\end{cases}
\]
Denotational semantics of labels and jumps

goto \ L \text{ ignores the current continuation; instead, it restarts the}
continuation associated with \( L \) in the environment.

\[
\llbracket \text{goto} \ L \rrbracket \ \rho \sigma \ k = \rho(\text{\( L \)}) \sigma
\]

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

\[
\llbracket \text{begin } s_1; \ L : s_2 \text{ end} \rrbracket \ \rho \sigma \ k = \llbracket s_1; s_2 \rrbracket \ \rho' \sigma \ k
\]

where \( \rho' = \rho[\text{\( L \)} \leftarrow k_2] \)

and \( k_2 = \lambda\sigma'.' \llbracket s_2 \rrbracket \ \rho' \sigma' \ k \)
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.
Naively:

\[ \text{Val} = \text{Num} + (\text{Val} \rightarrow \text{Val}) + \{\bot\} \]

\[ [\text{expr}] : (\text{Var} \xrightarrow{\text{fin}} \text{Val}) \rightarrow \text{Val} \]

\[ [x] \rho = \rho(x) \]

\[ [\lambda x. \ e] \rho = v \mapsto [e] (\rho[x \leftarrow v]) \]

\[ [e_1 \ e_2] \rho = ([e_1] \rho) ([e_2] \rho) \]

Problem 1: \text{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:

\[
Res \approx Num + Fun + \{\bot\} + \{err\} \quad \text{and} \quad Fun = Res \xrightarrow{\text{cont}} Res
\]

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

Call by value:

\[
Res \approx Val + \{\bot\} + \{err\} \quad \text{and} \quad Val \approx Num + Fun \quad \text{and} \quad Fun = Val \xrightarrow{\text{cont}} Res
\]

\[
[e_1 \ e_2] \rho = \begin{cases} (\llbracket e_1 \rrbracket \rho) (\llbracket e_2 \rrbracket \rho) & \text{if } \llbracket e_1 \rrbracket \rho \in Fun \text{ and } \llbracket e_2 \rrbracket \rho \in Val \\ \bot & \text{if } \llbracket e_1 \rrbracket \rho = \bot \text{ or } \llbracket e_1 \rrbracket \rho \in Fun \text{ and } \llbracket e_2 \rrbracket \rho = \bot \\ err & \text{otherwise} \end{cases}
\]
The CPS transformation
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 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

\[
V(cst) = \lambda k. k \text{ cst}
\]
\[
V(x) = \lambda k. k \text{ x}
\]
\[
V(\lambda x. e) = \lambda k. k (\lambda x. V(e))
\]
\[
V(e_1 e_2) = \lambda k. V(e_1) (\lambda v_1. V(e_2) (\lambda v_2. v_1 v_2 k))
\]

Variables are bound to values, hence \(V(x) = \lambda k. k x\).

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

\[ \mathcal{N}(\text{cst}) = \lambda k. \; k \; \text{cst} \]
\[ \mathcal{N}(x) = \lambda k. \; x \; k \]
\[ \mathcal{N}(\lambda x. \; e) = \lambda k. \; k \; (\lambda x. \; \mathcal{N}(e)) \]
\[ \mathcal{N}(e_1 \; e_2) = \lambda k. \; \mathcal{N}(e_1) \; (\lambda v_1. \; v_1 \; (\mathcal{N}(e_2)) \; k) \]

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 \( \mathcal{N}(e_2) \).
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

\[ \nu(f \; x) = \lambda k. \; (\lambda k_1. \; k_1 f) \; (\lambda v_1. \; (\lambda k_2. \; k_2 \; x) \; (\lambda v_2. \; v_1 \; v_2 \; k)) \]

instead of just \( \lambda k. \; f \; x \; k \).

This can be avoided by performing “administrative reductions” \( \xrightarrow{adm} \) on the result of the CPS transformation: these are \( \beta \)-reductions that remove the “administrative redexes” introduced by the translation. In particular, we can do

\[(\lambda k. \; k \; v) \; (\lambda x. \; a) \xrightarrow{adm} (\lambda x. \; a) \; v \xrightarrow{adm} a[x \leftarrow v] \]

whenever \( v \) is a value or a variable.
Examples of CPS transformations (after administrative reductions)

\[ V(f(g \, x)) \]
\[ = \lambda k. \, g \, x \, (\lambda v. \, f \, v \, k)) \]

\[ N(f(g \, x)) \]
\[ = \lambda k. \, f \, (\lambda v. \, v \, (\lambda k'. \, g \, (\lambda v'. \, v' \, x \, k'))) \, k) \]

\[ V(\text{let rec } \text{fact} = \lambda n. \, \text{if } n = 0 \, \text{then } 1 \, \text{else } n \, * \, \text{fact}(n - 1)) \]
\[ = \text{let rec } \text{fact} = \lambda n. \, \lambda k. \]
\[ \quad \text{if } n = 0 \, \text{then } k \, 1 \, \text{else } \text{fact}(n - 1) \, (\lambda v. \, k \, (n \, * \, v))) \]
Specifying a reduction strategy using operational semantics

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

$$e \xrightarrow{\varepsilon} e'$$

$$C[e] \rightarrow C[e']$$

[Diagram showing reduction and head reduction]
The usual strategies

**Weak lambda-calculus:** we can $\beta$-reduce anywhere but under a $\lambda$.

$$(\lambda x. e)\ e' \xrightarrow{\bar{e}} e\{x \leftarrow e'\}$$

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

**Call by name:** no reductions in arguments to applications.

$$(\lambda x. e)\ e' \xrightarrow{\bar{e}} e\{x \leftarrow e'\}$$

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

**Call by value:** left-to-right reduction of applications; $\beta$-reduction restricted to values $v ::= \text{cst} \mid \lambda x. e$.

$$(\lambda x. e)\ v \xrightarrow{\bar{e}} e\{x \leftarrow v\}$$

$C ::= [] \mid C\ e \mid v\ C$
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 $V(e)$ or $N(e)$ to the initial continuation $\lambda x. x$.

**Theorem**

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

If $e \rightarrow^* \text{cst}$ (resp. $e$ diverges) in call by name, then $N(e) (\lambda x. x) \rightarrow^* \text{cst}$ (resp. $N(e) (\lambda x. x)$ diverges).
A difficult proof, relying on this simulation diagram:

$a : k$, the *colon translation*, is $\nu(a) \ k$ where some administrative redexes were reduced.
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: \[ b ::= a \mid a_1 a_2 \mid a_1 a_2 a_3 \]

\( V(e) \) is an atom, and \( V(e) (\lambda x. x) \) is a body.

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

Atoms: \( a ::= x \mid \text{cst} \mid \lambda v. b \mid \lambda x. \lambda k. b \)

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 \( \nu(e) (\lambda x. x) \), 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

\[
(\lambda x. b) v_2 \rightarrow b[x \leftarrow v_2] \\
(\lambda x. \lambda k. b) v_2 v_3 \rightarrow (\lambda k. b)[x \leftarrow v_2] v_3 \rightarrow b[x \leftarrow v_2, k \leftarrow v_3].
\]
Programming in continuation-passing style
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

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

The usual “internal” iterator in OCaml:

```ml
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:

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

```ocaml
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.

```ocaml
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
```
The natural interface (in “direct style”):

\[
\text{spawn: } (\text{unit} \to \text{unit}) \to \text{unit}
\]
Start a new thread.

\[
\text{yield: } \text{unit} \to \text{unit}
\]
Suspend the current thread;
switch to another runnable thread.

\[
\text{terminate: } \text{unit} \to \text{unit}
\]
Stop the current thread forever.
A library of cooperative threads

The CPS interface (with an explicit continuation):

\texttt{spawn}: (\texttt{unit} -> \texttt{unit}) -> \texttt{unit}

Start a new thread.

\texttt{yield}: (\texttt{unit} -> \texttt{unit}) -> \texttt{unit}

Suspend the current thread;
switch to another runnable thread.

\texttt{terminate}: \texttt{unit} -> \texttt{unit}

Stop the current thread forever.
A queue of runnable threads (suspended, but ready to restart).

```ocaml
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 \texttt{count}, yielding at every number:

\begin{verbatim}
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
\end{verbatim}

Example of use:

\begin{verbatim}
let () =
  spawn (fun () -> process "a" 5);
  spawn (fun () -> process "b" 3);
  process "c" 6
\end{verbatim}

(Prints c1 a1 b1 c2 a2 b2 c3 a3 b3 c4 a4 c5 a5 c6.)
A continuation can be invoked several times. This can be useful to implement backtracking.

Example: matching regular expressions.

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

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

```ocaml
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))))
```
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.

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

Ex: counting dice throws $\geq 16$.

```ocaml
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”.
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 (≠ Lisp), implemented using closures.

An explanation of Algol by translation to extended ISWIM:

- Mutable state → adding ML-style references.
- Non-local “goto” → adding the \( \top \) control operator.
The $J$ control operator

The evaluation of $J(\lambda y. e') \, v$ 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: $J (\lambda x. x) \, v$ behaves like `return v` in C.

Using $J$ to encode labels and `goto`:

```
begin s_1; L : s_2 \ end \ \rightsquigarrow \ \lambda _. \ let \ rec \ L = J(\lambda _. s_2) \ in \ s_1; L ()
goto L \ \rightsquigarrow \ L ()
```
The callcc operator \(\text{call with current continuation}\)

\[
\text{callcc}(\lambda k. e)
\]

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:

- G. Sussmann and G. Steele, 1975: catch and throw.
- The Scheme language, from 1982: call-with-current-continuation, shortened as call/cc.
Execution of \texttt{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 \texttt{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:

```ocaml
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:

```ml
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.

```ocaml
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 $\text{More}(x_1, k_1)$. 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.

```ocaml
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 \texttt{callcc}

The construct

\[
\text{try } e \text{ with } p_1 \rightarrow e_n \mid \ldots \mid p_n \rightarrow e_n
\]

translates into

\[
\text{trywith (fun () -> e) (fun exn -> (fun (match exn with }
\mid p_1 \rightarrow e_1 \mid \ldots \mid p_n \rightarrow e_n \\
\mid _ \rightarrow \text{raise exn})})
\]
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 \texttt{callcc}

\textbf{Semantics:}

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

\textbf{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

\[ \mathcal{V}(\text{callcc } f) = \lambda k. \mathcal{V}(f) (\text{resume } k) k \]

\[ \text{resume } k_0 = \lambda v. \lambda k. k_0 v \]

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 \text{resume } k), and once as continuation for \( f \).

For \text{resume } k_0, we ignore its continuation \( k \): execution continues with \( k_0 \).
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 \texttt{callcc}

\[ C[\text{callcc}(\lambda k. e)] \rightarrow C[(\lambda k. e)(\lambda v. \text{resume } C \ v)] \]

\[ C[\text{resume } C_0 \ v] \rightarrow C_0[v] \]

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

The rule for \texttt{callcc} duplicates the current context $C$.

The rule for \texttt{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:

\[
2 \times \text{delim} \left(1 + \text{capture} \left(\lambda k. k(k \ 0)\right)\right)
\]

\[
\xrightarrow{+} 2 \times (\text{let } k = \lambda v. 1 + v \text{ in } k(k \ 0))
\]

\[
\xrightarrow{+} 2 \times ((1 + (1 + 0))) \xrightarrow{+} 4
\]

(The captured continuation “goes from capture to delim”.)

(Additional benefit: delimited continuations are smaller than undelimited continuations, so capturing them can be less costly.)
Head-reduction rules under a context $C$, but with a sub-context $D$ that does not mention `delim`:

$$\text{delim}(D[\text{capture } (\lambda k. e)]) \xrightarrow{\varepsilon} \ldots$$

$$\overbrace{\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad C[\text{delim}(D[\text{capture } (\lambda k. e)])] \quad \rightarrow \quad C[\ldots]$$

Head reductions: (4 variants!)

$$\text{delim}(D[\text{capture } (\lambda k. e)]) \xrightarrow{\varepsilon} (\lambda k. e)(\lambda v. \text{resume } D \ v)$$

$$\text{resume } D \ v \xrightarrow{\varepsilon} D[v]$$

Variant: -ctrl-
Head-reduction rules under a context $C$, but with a sub-context $D$ that does not mention $\text{delim}$:

$$\text{delim}(D[\text{capture } (\lambda k. e)]) \xrightarrow{\varepsilon} \ldots$$

$$C[\text{delim}(D[\text{capture } (\lambda k. e)])] \rightarrow C[\ldots]$$

Head reductions: (4 variants!)

$$\text{delim}(D[\text{capture } (\lambda k. e)]) \xrightarrow{\varepsilon} (\lambda k. e)(\lambda v. \text{resume } D v)$$

$$\text{resume } D v \xrightarrow{\varepsilon} \text{delim}(D[v])$$

Variant: $-\text{ctrl}-, -\text{ctrl}+$

**Semantics of delimited continuations**
Semantics of delimited continuations

Head-reduction rules under a context $C$, but with a sub-context $D$ that does not mention $\text{delim}$:

$$\text{delim}(D[\text{capture } (\lambda k. e)]) \xrightarrow{\varepsilon} \ldots$$

$$C[\text{delim}(D[\text{capture } (\lambda k. e)])] \rightarrow C[\ldots]$$

Head reductions: (4 variants!)

$$\text{delim}(D[\text{capture } (\lambda k. e)]) \xrightarrow{\varepsilon} \text{delim}((\lambda k. e)(\lambda v. \text{resume } D v))$$

$$\text{resume } D v \xrightarrow{\varepsilon} D[v]$$

Variant: $-\text{ctrl}-, -\text{ctrl}+, +\text{ctrl}-$
Semantics of delimited continuations

Head-reduction rules under a context $C$, but with a sub-context $D$ that does not mention $\text{delim}$:

\[
\text{delim}(D[\text{capture } (\lambda k. e)]) \xrightarrow{\varepsilon} \ldots
\]

\[
C[\text{delim}(D[\text{capture } (\lambda k. e)])] \rightarrow C[\ldots]
\]

Head reductions: (4 variants!)

\[
\text{delim}(D[\text{capture } (\lambda k. e)]) \xrightarrow{\varepsilon} \text{delim}((\lambda k. e)(\lambda v. \text{resume } D v))
\]

\[
\text{resume } D v \xrightarrow{\varepsilon} \text{delim}(D[v])
\]

Variant: $-\text{ctrl}-, -\text{ctrl}+, +\text{ctrl}-, +\text{ctrl}+$. 

58
A menagerie of delimited control operators

(D. Hillerström, citation in references.)

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

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


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: