Evaluation Contexts

There typically many structural congruence (“boring”) rules in real-world programming languages.

It would be nice to have a more compact way to express them.

Evaluation contexts provide a mechanism to do just that.

Structural Operational Semantics (again)

The rules for structural operational semantics can be classified into two types

- structural congruence rules, which constrain the choice of reductions that can be performed next, thus defining both the order of evaluation and whether subexpressions are evaluated lazily (let’s call these “boring” rules)

- reduction rules, which describe the actual computation steps (let’s call these “interesting” rules)

For example, the CBV reduction strategy for the $\lambda$-calculus was captured in the following rules:

\[
\begin{align*}
\beta\text{-reduction: } & \quad e[v/x] = e' \\
& \quad (\lambda x. e) v \rightarrow e' \quad \text{cool!} \\
& \quad e_1 \rightarrow e'_1 \\
& \quad e_1 e_2 \rightarrow e'_1 e_2 \\
& \quad zzz... \\
& \quad e_2 \rightarrow e'_2 \\
& \quad v e_2 \rightarrow v e'_2 \\
& \quad zzz...
\end{align*}
\]

Evaluation Contexts

An evaluation context $E$, sometimes written $E[\cdot]$, is a $\lambda$-term or a metaexpression representing a family of $\lambda$-terms with a special variable $[\cdot]$ called the hole.

If $E[\cdot]$ is an evaluation context, then $E[e]$ represents $E$ with the term $e$ substituted for the hole.
Evaluation Contexts (cont.)

Evaluation contexts: expressions with one hole where “interesting work” is allowed to occur

\[ E ::= [] | E \cdot e | v \cdot (E, e) | (v, E) | E.1 | E.2 | \mathbf{A}(E) | \mathbf{B}(E) | (\text{match } E \text{ with } \mathbf{A}x. e_1 | \mathbf{B}y. e_2) \]

Define “filling the hole” \( E[e] \) in the obvious way (stapling or plugging):
- A metafunction of type EvalContext→Exp→Exp

Semantics: Use two judgments
- \( e \rightarrow e' \) with 1 rule: \( E[e] \rightarrow E[e'] \)
- \( e \rightarrow e' \) with all the “interesting work”:

\[
\begin{align*}
(\lambda x. e) & \rightarrow e[v/x] \\
(v_1, v_2).1 & \rightarrow v_1 \\
(v_1, v_2).2 & \rightarrow v_2
\end{align*}
\]

match \( \mathbf{A}(v) \) with \( \mathbf{A}x. e_1 | \mathbf{B}y. e_2 \rightarrow e_1[v/x] \)

Small Detour: Control Flow

Categories based on the purpose of the constructs.
- Invocation
  - Direct calls: functions, subroutines
  - Indirect calls: function pointers, class methods, closures
- Termination of Scope
  - Structured: break, break to a label, exceptions, CPS
  - Unstructured: goto, setjmp/longjmp, exit
- Selection
  - Structured: if/then/else, match, continue, switch, case
  - Unstructured: goto, computed goto, labeled entries

Control Flow (cont.)

- Iteration
  - Precomputed iteration space: do, foreach
  - Dynamic iteration space: for, while, recursion
- Concurrency
  - Manual: processes, threads, futures, coroutines
  - Automatic: constructs in concurrent/parallel frameworks for reductions
  - Communication and synchronization techniques are critical

Decomposition

Evaluation relies on decomposition (splitting or unstapling the correct subtree)
- Given \( e \), find \( E, e_\alpha, e'_\alpha \) such that \( e = E[e_\alpha] \) and \( e_\alpha \rightarrow e'_\alpha \)

Theorem (Unique Decomposition): There is at most one decomposition of \( e \)
- Hence evaluation is deterministic since at most one primitive step can apply to any expression

Theorem (Progress, restated): If \( e \) is well-typed, then there is a decomposition or \( e \) is a value

Continuations\(^1\)

Can we use functions to represent the control flow of a program?

\(^1\)Includes material based on lecture notes by Mark Hills, Mattox Beckman, Vikram Adve, Gul Agha, and Elsa Gunter (UIUC).
Continuations

Yes, by using the concept of a continuation.

- We will augment each procedure with an additional argument—a function to which it will pass the current computational result.
- The outer procedure “returns” no result; it will be kept in the function argument.
- This function argument, receiving the result, will be called the continuation.
- At its core, the continuation is just “the rest of the computation”; it tells us what we have left to do.
- Continuations can be used to model many control constructs.

First-class Continuations

First-class continuations are a language’s ability to completely control the execution order of instructions.

They can be used to jump:

- to a function that produced the call to the current function
- or to a function that has previously exited.

You can think of them saving the state of the program, however, first-class continuations do not save program data, just the execution context.

The Continuation Sandwich

“Say you’re in the kitchen in front of the refrigerator, thinking about a sandwich. You take a continuation right there and stick it in your pocket. Then you get some turkey and bread out of the refrigerator and make yourself a sandwich, which is now sitting on the counter. You invoke the continuation in your pocket, and you find yourself standing in front of the refrigerator again, thinking about a sandwich. But fortunately, there’s a sandwich on the counter, and all the materials used to make it are gone. So you eat it. :-)


Continuation Passing Style

Writing procedures so that they can take a continuation to which they pass on the computation result, and which return no result is called **continuation passing style** (CPS).

CPS provides a programming technique for all forms of “non-local” control flow:

- exceptions
- GOTO
- generators (e.g., yield in python)
- async (C#)

CPS turns all non-tail calls into tail calls.
- Essentially a higher order functional GOTO

CPS Terminology

- CPS also acts as a compilation technique to implement non-local control flow.
- Especially useful in interpreters
- Also acts as a formalization of non-local control flow in denotational semantics.
Example

A simple reporting continuation:

```ocaml
let report x = (print_int x; print_newline( ));
```

And a function that uses it

```ocaml
let plusk a b k = k (a + b);; plusk 20 22
report;;
```

---

Example: Factorial

```ocaml
(* First, the non-CPS version: *)
let rec factorial n =
  if n = 0 then 1 else n * factorial (n - 1);;
factorial 4;;
```

```ocaml
(* Now, define factorial with continuations *)
let rec factorialk n k =
  if n = 0
     then k 1
  else factorialk (n - 1) (fun m -> k (n * m));;
factorialk 4 print_int;;
```

---

Example: Exceptions

```ocaml
# exception Zero;;
exception Zero

# let rec list_mult_aux list =
  match list with
  [ ] -> 1
  | x :: xs -> if x = 0 then raise Zero
     else x * list_mult_aux xs;;
val list_mult_aux : int list -> int = <fun>

# let rec list_mult list =
  try list_mult_aux list with Zero -> 0;;
val list_mult : int list -> int = <fun>

# list_mult [3;4;2];;
- : int = 24

# list_mult [7;4;0];;
- : int = 0
```

---

Exceptions in OCaml

- The current computation is aborted;
- Control is “thrown” back up the call stack until a matching handler is found
- all intermediate calls waiting for a return value are thrown away.

---

Continuations as Exceptions

```ocaml
# let multk m n k =
  let r = m * n in
  (print_string "product result: ";
   print_int r; print_newline ());
  k r);;
val multk : int -> int -> int -> unit

# let rec list_multk_aux list k kexcp =
  match list with
  [ ] -> k 1
  | x :: xs -> if x = 0 then kexcp 0
    else list_multk_aux xs
    (fun r -> multk r x k) kexcp
val list_multk_aux : int list -> int -> 'a -> 'a

# let rec list_multk list k = list_multk_aux list k (fun a -> a);;
val list_multk : int list -> int -> 'a -> 'a
```

---

Exceptions, Part 2

```ocaml
# list_multk [3;4;2] report;;
product result: 2
product result: 8
product result: 24
- : unit = ()

# list_multk [7;4;0] report;;
0
- : unit = ()
```
Continuations in our CBV λ-Calculus

Now that we have defined $E$ explicitly in our metalanguage, what if we also put it on our language?

- From metalanguage to language is called reification

First-class continuations in one slide:

- $e ::= \ldots | \text{letcc } x. e | \text{throw } e e | \text{cont } E$
- $v ::= \ldots | \text{cont } E$
- $E ::= \ldots | \text{throw } E e | \text{throw } v E$

$E[\text{letcc } x. e] \rightarrow E[(\lambda x. e)(\text{cont } E)]$

- New operational rules for $\rightarrow$ not $\rightarrow_\beta$ because “the $E$ matters”
- $\text{letcc } x. e$ grabs the current evaluation context (“the stack”)
- $\text{throw } (\text{cont } E') v$ restores old context: “jump somewhere”
- $\text{cont } E$ not in source programs: “saved stack (value)”

Examples (exceptions-like)

1 + (letcc $k. 2 + 3$) $\rightarrow^*$ 6
1 + (letcc $k. 2 + (\text{throw } k 3)$) $\rightarrow^*$ 4
1 + (letcc $k. (\text{throw } k (2 + 3))$) $\rightarrow^*$ 6
1 + (letcc $k. (\text{throw } k (\text{throw } k (\text{throw } k 2)))$) $\rightarrow^*$ 3

Note: Breaks the Church-Rosser property. Under full reduction:

- letcc $k. (\text{throw } k (\text{throw } k 1)) + (\text{throw } k 2))$ $\rightarrow^*$ 1
- letcc $k. (\text{throw } k 1) + (\text{throw } k 2))$ $\rightarrow^*$ 2

Refresher: Church-Rosser Theorem

When applying reduction rules to terms in the lambda calculus, the ordering in which the reductions are chosen does not make a difference to the eventual result.

Another view

If you’re confused, think call stacks:

- Exceptions
- Cooperative threads (including coroutines)
  - “yield” captures the continuation (the “how to resume me”) and gives it to the scheduler (implemented in the language), which then throws to another thread’s “how to resume me”
- Other crazy things
  - Often called the “goto of functional programming” — incredibly powerful, but nonstandard uses are usually inscrutable
  - Key point is that we can “jump back in” unlike boring-old exceptions
Where are we

Done:
- Redefined our operational semantics using evaluation contexts
- That made it easy to define first-class continuations
- Example uses of continuations

Now: Rather than adding a powerful primitive, we can achieve the same effect via a whole-program translation into a sublanguage (source-to-source transformation)
- No expressions with nontrivial evaluation contexts
- Every expression becomes a continuation-accepting function
- Never “return” — instead call the current continuation
- Will be able to reintroduce letcc and throw as $O(1)$ operations

The CPS transformation (one way to do it)

A metafunction from expressions to expressions

Example source language (other features similar):

$$
e ::= x | \lambda x. e | e + e | c | e e
$$

$$
v ::= x | \lambda x. e | c
$$

$$
\text{CPS}_E(v) = \lambda k. \text{CPS}_V(v)
$$

$$
\text{CPS}_E(e_1 + e_2) = \lambda k. \text{CPS}_E(e_1) \lambda x_1. \text{CPS}_E(e_2) \lambda x_2. k \ (x_1 + x_2)
$$

$$
\text{CPS}_E(e_1 e_2) = \lambda k. \text{CPS}_E(e_1) \lambda f. \text{CPS}_E(e_2) \lambda x. f \ x \ k
$$

$$
\text{CPS}_V(c) = c
$$

$$
\text{CPS}_V(x) = x
$$

$$
\text{CPS}_V(\lambda x. e) = \lambda x. \lambda k. \text{CPS}_E(e) \ k
$$

To run the whole program $e$, do $\text{CPS}_E(e) \lambda x. x$

Result of the CPS transformation

- Correctness: $e$ is equivalent to $\text{CPS}_E(e) \lambda x. x$
- If whole program has type $\tau_P$ and $e$ has type $\tau$, then $\text{CPS}_E(e)$ has type $(\tau \rightarrow \tau_P) \rightarrow \tau_P$
- Fixes evaluation order: $\text{CPS}_E(e)$ will evaluate $e$ in left-to-right call-by-value
  - Other similar transformations encode other evaluation orders
  - Every intermediate computation is bound to a variable (helpful for compiler writers)
- For all $e$, evaluation of $\text{CPS}_E(e)$ stays in this sublanguage:
  $$
e ::= v | v v | v v v | v (v + v)
$$
  $$
v ::= x | \lambda x. e | c
$$
- Hence no need for a call-stack: every call is a tail-call
  - Now the program is maintaining the evaluation context via a closure that has the next "link" in its environment that has the next "link" in its environment, etc.

Encoding first-class continuations

If you apply the CPS transform, then letcc and throw can become $O(1)$ operations encodable in the source language

$$
\text{CPS}_E(\text{letcc } k. e) = \lambda k. \text{CPS}_E(e) \ k
$$

$$
\text{CPS}_E(\text{throw } e_1 e_2) = \lambda k. \text{CPS}_E(e_1) \lambda x_1. \text{CPS}_E(e_2) \lambda x_2. x_1 \ x_2
$$

or just $x_1$

- letcc gets passed the current continuation just as it needs
- throw ignores the current continuation just as it should

You can also manually program in this style (fully or partially)
- Has other uses as a programming idiom too...

A useful advanced programming idiom

- A first-class continuation can "reify session state" in a client-server interaction
  - If the continuation is passed to the client, which returns it later, then the server can be stateless
  - Suggests CPS for web programming
  - Better: tools that do the CPS transformation for you
    - Gives you a "prompt-client" primitive without server-side state

- Because CPS uses only tail calls, it avoids deep call stacks when traversing recursive data structures
  - See lec13code.ml for this and related idioms

In short, "thinking in terms of CPS" is a powerful technique few programmers have