

Functional programming languages

Part III: program transformations

Xavier Leroy

INRIA Rocquencourt

MPRI 2-4-2, 2007

Generalities on program transformations

In the broadest sense: all translations between programming languages that preserve the meaning of programs.

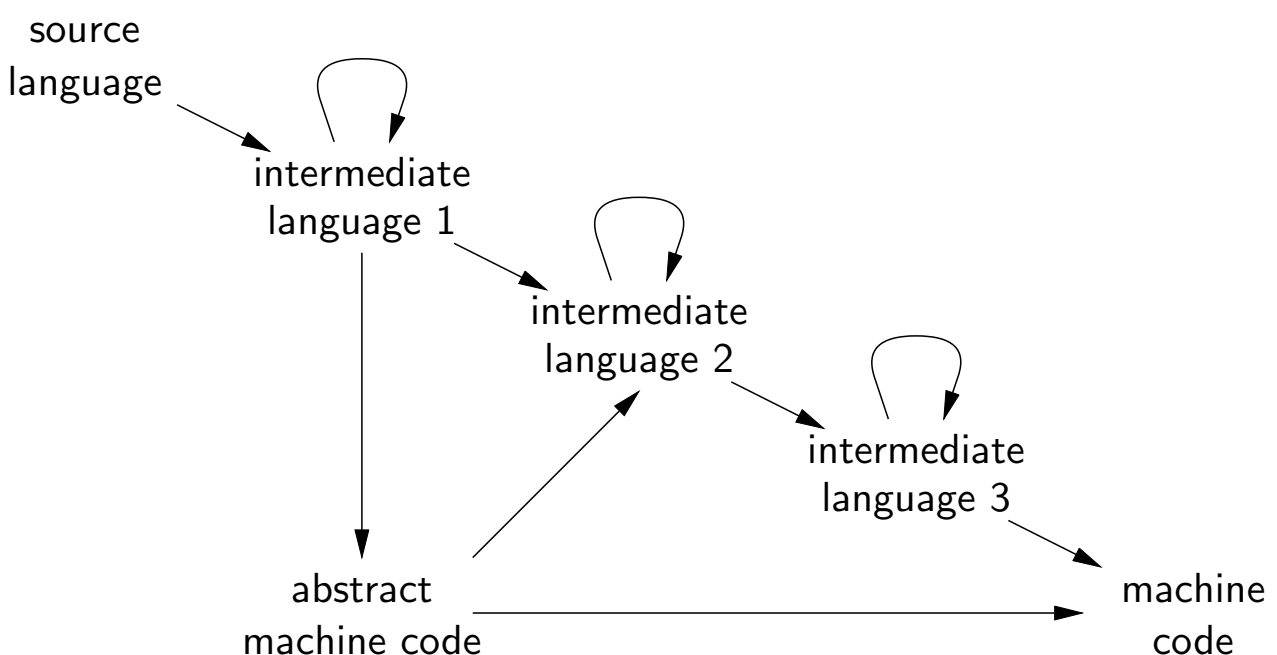
Focus in this lecture: examples of program transformations that eliminate high-level features of a (mostly) functional programming language and target a smaller or lower-level language.

I.e. translate from language L_1 having features A, B, C, D to language L_2 having features B, C, D, E .

Uses for program transformations

- 1 As passes in a compiler.
Progressively bridge the gap between high-level source languages and machine code.
- 2 To give semantics to the source language.
The semantics of feature A is defined in terms of that of features B, C, D, E .
- 3 To program in languages that lack the desired feature A .
E.g. use higher-order functions or objects in C;
use imperative programming in Haskell or Coq.

Big picture of compilation



Outline

- 1 Closure conversion
- 2 Defunctionalization
- 3 Exception-returning style
- 4 State-passing style
- 5 Continuation-passing style

Closure conversion

Goal: make explicit the construction of closures and the accesses to the environment part of closures.

Input: a fully-fledged functional programming language, with general functions (possibly having free variables) as first-class values.

Output: the same language where only **closed** functions (without free variables) are first-class values. Such closed functions can be represented at run-time as code pointers, just as in C for instance.

Idea: every function receives its own closure as an extra argument, from which it recovers values for its free variables. Such functions are closed. Function closures are explicitly represented as a tuple (closed function, values of free variables).

Uses: compilation; functional programming in C, Java, ...

Definition of closure conversion

$$\begin{aligned}
 \llbracket x \rrbracket &= x \\
 \llbracket \lambda x. a \rrbracket &= \text{tuple}(\lambda c, x. \text{let } x_1 = \text{field}_1(c) \text{ in} \\
 &\quad \dots \\
 &\quad \text{let } x_n = \text{field}_n(c) \text{ in} \\
 &\quad \llbracket a \rrbracket, \\
 &\quad x_1, \dots, x_n) \\
 &\quad \text{where } x_1, \dots, x_n \text{ are the free variables of } \lambda x. a \\
 \llbracket a \ b \rrbracket &= \text{let } c = \llbracket a \rrbracket \text{ in } \text{field}_0(c)(c, \llbracket b \rrbracket)
 \end{aligned}$$

The translation extends isomorphically to other constructs, e.g.

$$\begin{aligned}
 \llbracket \text{let } x = a \text{ in } b \rrbracket &= \text{let } x = \llbracket a \rrbracket \text{ in } \llbracket b \rrbracket \\
 \llbracket a + b \rrbracket &= \llbracket a \rrbracket + \llbracket b \rrbracket
 \end{aligned}$$

Example of closure conversion

Source program in Caml:

```

fun x lst ->
  let rec map f lst =
    match lst with [] -> [] | hd :: tl -> f hd :: map f tl
  in
  map (fun y -> x + y) lst

```

Result of partial closure conversion for the `f` argument of `map`:

```

fun x lst ->
  let rec map f lst =
    match lst with [] -> []
                | hd :: tl -> field0(f)(f,hd) :: map f tl
  in
  map tuple(λc,y. let x = field1(c) in x + y,
            x)
  lst

```

Closure conversion for recursive functions

In a recursive function $\mu f.\lambda x.a$, the body a needs access to f , i.e. the closure for itself. This closure can be found in the extra function parameter that closure conversion introduces.

$$\llbracket \mu f.\lambda x.a \rrbracket = \text{tuple}(\lambda f, x. \text{let } x_1 = \text{field}_1(f) \text{ in} \\ \dots \\ \text{let } x_n = \text{field}_n(f) \text{ in} \\ \llbracket a \rrbracket, \\ x_1, \dots, x_n)$$

where x_1, \dots, x_n are the free variables of $\mu f.\lambda x.a$

In other terms, regular functions $\lambda x.a$ are converted exactly like pseudo-recursive functions $\mu c.\lambda x.a$ where c is a variable not free in a .

Minimal environments in closures

Closures built by closure conversions have **minimal environments**: they contain values only for variables actually free in the function.

In contrast, closures built by the abstract machines of lecture II have **full environments** containing values for all variables in scope when the function is evaluated.

Minimal closures consume less memory and enable a garbage collector to reclaim other data structures earlier. Consider:

$$\text{let } l = \langle \text{big list} \rangle \text{ in } \lambda x. x+1$$

With full closures, the list l is reachable from the closure of $\lambda x.x + 1$ and cannot be reclaimed as long as this closure is live.

With minimal closures, no reference to l is kept in the closure, enabling earlier garbage collection.

Closure conversion in object-oriented style

If the target of the conversion is an object-oriented language in the style of Java, we can use the following variant of closure conversion:

$$\begin{aligned} \llbracket x \rrbracket &= x \\ \llbracket \lambda x. a \rrbracket &= \text{new } C_{\lambda x. a}(x_1, \dots, x_n) \\ &\quad \text{where } x_1, \dots, x_n \text{ are the free variables of } \lambda x. a \\ \llbracket a \ b \rrbracket &= \llbracket a \rrbracket. \text{apply}(\llbracket b \rrbracket) \end{aligned}$$

Closure conversion in object-oriented style

The class $C_{\lambda x. a}$ (one for each λ -abstraction in the source) is defined as follows:

```
class  $C_{\lambda x. a}$  {
    Object  $x_1$ ; ...; Object  $x_n$ ;

     $C_{\lambda x. a}$ (Object  $x_1$ , ..., Object  $x_n$ ) {
        this. $x_1$  =  $x_1$ ; ...; this. $x_n$  =  $x_n$ ;
    }

    Object apply(Object  $x$ ) {
        return  $\llbracket a \rrbracket$ ;
    }
}
```

Closures and objects

In more general terms:

- Closure \approx Object with a single `apply` method
- Object \approx Closure with multiple entry points

Both function application and method invocation compile down to **self application**:

$$\begin{aligned} \llbracket \text{fun } arg \rrbracket &= \text{let } c = \llbracket \text{fun} \rrbracket \text{ in field}_0(c)(c, \llbracket arg \rrbracket) \\ \llbracket \text{obj.meth}(arg) \rrbracket &= \text{let } o = \llbracket \text{obj} \rrbracket \text{ in } o.\text{meth}(o, \llbracket arg \rrbracket) \end{aligned}$$

Outline

- 1 Closure conversion
- 2 Defunctionalization
- 3 Exception-returning style
- 4 State-passing style
- 5 Continuation-passing style

Defunctionalization

Goal: like closure conversion, make explicit the construction of closures and the accesses to the environment part of closures. Unlike closure conversion, do not use closed functions as first-class values.

Input: a fully-fledged functional programming language, with general functions (possibly having free variables) as first-class values.

Output: any first-order language (no functions as values).

Idea: represent each function value $\lambda x.a$ as a data structure $C(v_1, \dots, v_n)$ where the constructor C uniquely identifies the function, and the constructor arguments v_1, \dots, v_n are the values of the free variables x_1, \dots, x_n .

Uses: functional programming in Pascal, Ada, Basic, ...

Definition of defunctionalization

$$\begin{aligned} \llbracket x \rrbracket &= x \\ \llbracket \lambda x.a \rrbracket &= C_{\lambda x.a}(x_1, \dots, x_n) \\ &\quad \text{where } x_1, \dots, x_n \text{ are the free variables of } \lambda x.a \\ \llbracket \mu f.\lambda x.a \rrbracket &= C_{\mu f.\lambda x.a}(x_1, \dots, x_n) \\ &\quad \text{where } x_1, \dots, x_n \text{ are the free variables of } \mu f.\lambda x.a \\ \llbracket a \ b \rrbracket &= \text{apply}(\llbracket a \rrbracket, \llbracket b \rrbracket) \end{aligned}$$

(Other constructs: isomorphically.)

Definition of defunctionalization

The apply function collects the bodies of all functions and dispatches on its first argument. There is one case per function occurring in the source program.

```
let rec apply(fun, arg) =
  match fun with
  |  $C_{\lambda x.a}(x_1, \dots, x_n) \rightarrow \text{let } x = \text{arg in } \llbracket a \rrbracket$ 
  |  $C_{\mu f'.\lambda x'.a'}(x'_1, \dots, x'_{n'}) \rightarrow \text{let } f' = \text{fun in let } x' = \text{arg in } \llbracket a' \rrbracket$ 
  | ...
in  $\llbracket \text{program} \rrbracket$ 
```

Note: this is a whole-program transformation, unlike closure conversion.

Example

Defunctionalization of $(\lambda x.\lambda y.x) 1 2$:

```
let rec apply (fun, arg) =
  match fun with
  | C1()    -> let x = arg in C2(x)
  | C2(x)   -> let y = arg in x
in
  apply(apply(C1(), 1), 2)
```

We write C1 for $C_{\lambda x.\lambda y.x}$ and C2 for $C_{\lambda y.x}$.

Outline

- 1 Closure conversion
- 2 Defunctionalization
- 3 Exception-returning style
- 4 State-passing style
- 5 Continuation-passing style

Exceptions

Exceptions are a control structure useful for error reporting, error handling, and more generally all situations that need “early exit” out of a computation.

```
let f x =  
  try 1 + (if x = 0 then raise Error else 100 / x)  
  with Error -> 101
```

In `try a with Error → b`, if `a` evaluates normally without raising an exception, its value is returned as the value of the `try...with`. For instance, `f 4 = 26`.

If `a` raises the `Error` exception, control branches to `b`, which becomes the result of the `try...with`. For instance, `f 0 = 101`.

Exceptions

For a more realistic example of early exit, consider the computation of the product of a list of integers, returning 0 as soon as a list element is 0:

```
let product lst =
  let rec prod = function
    | [] -> 1
    | 0 :: tl -> raise Exit
    | hd :: tl -> hd * prod tl
  in
  try prod lst with Exit -> 0
```

Reduction semantics for exceptions

In Wright-Felleisen style, add two head reduction rules for `try...with` and a generic **exception propagation** rule:

$$\begin{aligned} (\text{try } v \text{ with } x \rightarrow b) &\xrightarrow{\varepsilon} v \\ (\text{try raise } v \text{ with } x \rightarrow b) &\xrightarrow{\varepsilon} b[x \leftarrow v] \\ X[\text{raise } v] &\xrightarrow{\varepsilon} \text{raise } v \text{ if } X \neq [] \end{aligned}$$

Exception propagation contexts X are like reduction contexts E but do not allow skipping past a `try...with`

Reduction contexts:

$$E ::= [] \mid E \ b \mid v \ E \mid \text{try } E \ \text{with } x \rightarrow b \mid \dots$$

Exception propagation contexts:

$$X ::= [] \mid X \ b \mid v \ X \mid \dots$$

Reduction semantics for exceptions

Assume the current program is $p = E[\text{raise } v]$, that is, we are about to raise an exception. If there is a `try...with` that encloses the raise, the program will be decomposed as

$$p = E'[\text{try } X[\text{raise } v] \text{ with } x \rightarrow b]$$

where X contains no `try...with` constructs.

$X[\text{raise } v]$ head-reduces to `raise v`, and $E'[\text{try } [] \text{ with } x \rightarrow b]$ is an evaluation context. The reduction sequence is therefore:

$$\begin{aligned} p = E'[\text{try } X[\text{raise } v] \text{ with } x \rightarrow b] &\rightarrow E'[\text{try } \text{raise } v \text{ with } x \rightarrow b] \\ &\rightarrow b[x \leftarrow v] \end{aligned}$$

If there are no `try...with` around the raise, E is an exception propagation context X and the reduction is therefore

$$p = E[\text{raise } v] \rightarrow \text{raise } v$$

Reduction semantics for exceptions

Considering reduction sequences, a fourth possible outcome of evaluation appears: termination on an uncaught exception.

- Termination: $a \xrightarrow{*} v$
- **Uncaught exception:** $a \xrightarrow{*} \text{raise } v$
- Divergence: $a \xrightarrow{*} a' \rightarrow \dots$
- Error: $a \xrightarrow{*} a' \not\rightarrow$ where $a \neq v$ and $a \neq \text{raise } v$.

Natural semantics for exceptions

In natural semantics, the evaluation relation becomes $a \Rightarrow r$ where evaluation results are $r ::= v \mid \text{raise } v$.

Add the following rules for `try...with`:

$$\frac{a \Rightarrow v}{\text{try } a \text{ with } x \rightarrow b \Rightarrow v} \qquad \frac{a \Rightarrow \text{raise } v' \quad b[x \leftarrow v'] \Rightarrow r}{\text{try } a \text{ with } x \rightarrow b \Rightarrow r}$$

as well as exception propagation rules such as:

$$\frac{a \Rightarrow \text{raise } v}{a \ b \Rightarrow \text{raise } v} \qquad \frac{a \Rightarrow v' \quad b \Rightarrow \text{raise } v}{a \ b \Rightarrow \text{raise } v}$$

Conversion to exception-returning style

Goal: get rid of exceptions.

Input: a functional language featuring exceptions (`raise` and `try...with`).

Output: a functional language with pattern-matching but no exceptions.

Idea: every expression a evaluates to either $V(v)$ if a evaluates normally or to $E(v)$ if a terminates early by raising exception v .
 V, E are datatype constructors.

Uses: giving semantics to exceptions; programming with exceptions in Haskell; reasoning about exceptions in Coq.

Definition of the conversion: Core constructs

$$\begin{aligned}
 \llbracket N \rrbracket &= V(N) \\
 \llbracket x \rrbracket &= V(x) \\
 \llbracket \lambda x. a \rrbracket &= V(\lambda x. \llbracket a \rrbracket) \\
 \llbracket \text{let } x = a \text{ in } b \rrbracket &= \text{match } \llbracket a \rrbracket \text{ with } E(x) \rightarrow E(x) \mid V(x) \rightarrow \llbracket b \rrbracket \\
 \llbracket a \ b \rrbracket &= \text{match } \llbracket a \rrbracket \text{ with} \\
 &\quad \mid E(e_a) \rightarrow E(e_a) \\
 &\quad \mid V(v_a) \rightarrow \\
 &\quad \quad \text{match } \llbracket b \rrbracket \text{ with } E(e_b) \rightarrow E(e_b) \mid V(v_b) \rightarrow v_a \ v_b
 \end{aligned}$$

Effect on types: if $a : \tau$ then $\llbracket a \rrbracket : \llbracket \tau \rrbracket$

where $\llbracket \tau_1 \rightarrow \tau_2 \rrbracket = (\tau_1 \rightarrow \llbracket \tau_2 \rrbracket)$ outcome and $\llbracket \tau \rrbracket = \tau$ outcome otherwise
 and where type 'a outcome = V of 'a | E of exn.

Definition of the conversion: Exception-specific constructs

$$\begin{aligned}
 \llbracket \text{raise } a \rrbracket &= \text{match } \llbracket a \rrbracket \text{ with } E(e_a) \rightarrow E(e_a) \mid V(v_a) \rightarrow E(v_a) \\
 \llbracket \text{try } a \text{ with } x \rightarrow b \rrbracket &= \text{match } \llbracket a \rrbracket \text{ with } E(x) \rightarrow \llbracket b \rrbracket \mid V(v_a) \rightarrow V(v_a)
 \end{aligned}$$

Example of conversion

```

[[try fun arg with exn → 0]] =
  match
    match V(fun) with
    | E(x) → E(x)
    | V(x) →
      match V(arg) with
      | E(y) → E(y)
      | V(y) → x y
  with
  | V(z) → V(z)
  | E(exn) → V(0)

```

Administrative reductions

The naive conversion generates many useless `match` constructs over arguments whose shape $V(\dots)$ or $E(\dots)$ is known at compile-time.

These can be eliminated by performing **administrative reductions** \xrightarrow{adm} at compile-time, just after the conversion:

$$(\text{match } E(v) \text{ with } E(x) \rightarrow b \mid V(x) \rightarrow c) \xrightarrow{adm} b[x \leftarrow v]$$

$$(\text{match } V(v) \text{ with } E(x) \rightarrow b \mid V(x) \rightarrow c) \xrightarrow{adm} c[x \leftarrow v]$$

Example of conversion

After application of administrative reductions, we obtain:

```

[[try fun arg with exn → 0]] =
  match fun arg with
  | V(z) → V(z)
  | E(exn) → V(0)

```

Correctness of the conversion

Define the conversion of a value $\llbracket v \rrbracket_v$ as $\llbracket N \rrbracket_v = N$ and $\llbracket \lambda x. a \rrbracket_v = \lambda x. \llbracket a \rrbracket$

Theorem 1

- ① *If $a \Rightarrow v$, then $\llbracket a \rrbracket \Rightarrow V(\llbracket v \rrbracket_v)$.*
- ② *If $a \Rightarrow \text{raise } v$, then $\llbracket a \rrbracket \Rightarrow E(\llbracket v \rrbracket_v)$.*
- ③ *If $a \Rightarrow \infty$, then $\llbracket a \rrbracket \Rightarrow \infty$.*

Proof.

(1) and (2) are proved simultaneously by induction on the derivation of $a \Rightarrow r$ where r is an evaluation result. (3) is by coinduction. All three proofs use the substitution lemma $\llbracket a[x \leftarrow v] \rrbracket = \llbracket a \rrbracket[x \leftarrow \llbracket v \rrbracket_v]$. □

Outline

- 1 Closure conversion
- 2 Defunctionalization
- 3 Exception-returning style
- 4 State-passing style
- 5 Continuation-passing style

State (imperative programming)

The word **state** in programming language theory refers to the distinguishing feature of imperative programming: the ability to assign (change the value of) variables after their definition, and to modify data structures in place after their construction.

References

A simple yet adequate way to model state is to introduce **references**: indirection cells / one-element arrays that can be modified in place. The basic operations over references are:

`ref a`

Create a new reference containing initially the value of a .

`deref a` also written `!a`

Return the current contents of reference a .

`assign a b` also written `a := b`

Replace the contents of reference a with the value of b .
Subsequent `deref a` operations will return this value.

Uses of references

A let-bound reference emulates an imperative variable:

```
int x = 3;           let x = ref 3 in
x = x + 1;          x := !x + 1;
return x;           !x
```

(We write $a; b$ instead of `let z = a in b` if z is not free in b .)

Such references also enable a function to maintain an internal state:

```
let make_pseudo_random_generator seed =
  let state = ref seed in
  λn. state := (!state * A + B) mod C; !state mod n
```

Uses of references

Arrays \approx list of references (or better: primitive arrays).

Records with mutable fields \approx tuples of references.

Imperative singly-linked list, with in-place concatenation:

```

type 'a mlist = 'a mlist_content ref
and 'a mlist_content = Nil | Cons of 'a * 'a mlist

let rec concat l1 l2 =
  match !l1 with
  | Nil → l1 := !l2
  | Cons(x, r) → concat !r l2

```

Semantics of references

Semantics based on substitutions fail to account for **sharing** between references:

$$\text{let } r = \text{ref } 1 \text{ in } r := 2; !r \not\rightarrow (\text{ref } 1) := 2; !(\text{ref } 1)$$

Left: the same reference r is shared between assignment and reading; result is 2.

Right: two distinct references are created, one is assigned, the other read; result is 1.

To account for sharing, we must use an additional level of indirection:

- `ref a` expressions evaluate to **locations** ℓ : a new kind of variable identifying references uniquely. (Locations ℓ are values.)
- A global environment called the **store** associates values to references.

Reduction semantics for references

The one-step reduction relation becomes $a / s \rightarrow a' / s'$
 (read: in initial store s , a reduces to a' and updates the store to s')

$$\begin{array}{l}
 (\lambda x. a) v / s \xrightarrow{\varepsilon} a[x \leftarrow v] / s \\
 \text{ref } v / s \xrightarrow{\varepsilon} l / (s + l \mapsto v) \quad \text{where } l \notin \text{Dom}(s) \\
 \text{deref } l / s \xrightarrow{\varepsilon} s(l) / s \\
 \text{assign } l v / s \xrightarrow{\varepsilon} () / (s + l \mapsto v) \\
 \\
 \frac{a / s \xrightarrow{\varepsilon} a' / s'}{E(a) / s \rightarrow E(a') / s'} \quad (\text{context})
 \end{array}$$

Example of reduction sequence

In red: the active redex at every step.

```

let r = ref 3 in let x = r := !r + 1 in !r / ∅
→ let r = l in let x = r := !r + 1 in !r / {l ↦ 3}
→ let x = l := !l + 1 in !l / {l ↦ 3}
→ let x = l := 3 + 1 in !l / {l ↦ 3}
→ let x = l := 4 in !l / {l ↦ 3}
→ let x = () in !l / {l ↦ 4}
→ !l / {l ↦ 4}
→ 4
  
```

Conversion to state-passing style

Goal: get rid of state.

Input: a functional language featuring references.

Output: a pure functional language.

Idea: every expression a becomes a function that takes a run-time representation of the current store and returns a pair (result value, updated store).

Uses: give semantics to references; program imperatively in Haskell; reason over imperative code in Coq.

Definition of the conversion: Core constructs

$$\llbracket N \rrbracket = \lambda s. (N, s)$$

$$\llbracket x \rrbracket = \lambda s. (x, s)$$

$$\llbracket \lambda x. a \rrbracket = \lambda s. (\lambda x. \llbracket a \rrbracket, s)$$

$$\llbracket \text{let } x = a \text{ in } b \rrbracket = \lambda s. \text{match } \llbracket a \rrbracket s \text{ with } (x, s') \rightarrow \llbracket b \rrbracket s'$$

$$\begin{aligned} \llbracket a \ b \rrbracket &= \lambda s. \text{match } \llbracket a \rrbracket s \text{ with } (v_a, s') \rightarrow \\ &\quad \text{match } \llbracket b \rrbracket s' \text{ with } (v_b, s'') \rightarrow v_a \ v_b \ s'' \end{aligned}$$

Definition of the conversion: Constructs specific to references

$$\llbracket \text{ref } a \rrbracket = \lambda s. \text{match } \llbracket a \rrbracket s \text{ with } (v_a, s') \rightarrow \text{store_alloc } v_a s'$$

$$\llbracket !a \rrbracket = \lambda s. \text{match } \llbracket a \rrbracket s \text{ with } (v_a, s') \rightarrow (\text{store_read } v_a s', s')$$

$$\llbracket a := b \rrbracket = \lambda s. \text{match } \llbracket a \rrbracket s \text{ with } (v_a, s') \rightarrow \\ \text{match } \llbracket b \rrbracket s' \text{ with } (v_b, s'') \rightarrow \text{store_write } v_a v_b s''$$

The operations `store_alloc`, `store_read` and `store_write` provide a concrete implementation of the store. Any implementation of the data structure known as persistent extensible arrays will do.

Example of conversion

Administrative reductions:

$$(\text{match } (a, s) \text{ with } (x, s') \rightarrow b) \xrightarrow{\text{adm}} \text{let } x = a \text{ in } b[s' \leftarrow s]$$

$$\text{let } x = v \text{ in } b \xrightarrow{\text{adm}} b[x \leftarrow v]$$

$$\text{let } x = y \text{ in } b \xrightarrow{\text{adm}} b[x \leftarrow y]$$

Example of translation after administrative reductions:

$$\llbracket \text{let } r = \text{ref } 3 \text{ in let } x = r := !r + 1 \text{ in } !r \rrbracket = \\ \lambda s. \text{match alloc } s \ 3 \text{ with } (r, s1) \rightarrow \\ \text{let } t = \text{deref } r \ s1 \text{ in} \\ \text{let } u = t + 1 \text{ in} \\ \text{match assign } r \ u \ s1 \text{ with } (x, s2) \rightarrow \\ (\text{deref } r \ s2, s2)$$

Outline

- 1 Closure conversion
- 2 Defunctionalization
- 3 Exception-returning style
- 4 State-passing style
- 5 Continuation-passing style

Notion of continuation

Given a program p and a subexpression a of p , the **continuation** of a is the computations that remain to be done once a is evaluated to obtain the result of p .

It can be viewed as a function: (value of a) \mapsto (value of p).

Example 2

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

The continuation of $a = (1 + 2)$ is $\lambda x. x * (3 + 4)$.

The continuation of $a' = (3 + 4)$ is $\lambda x. 3 * x$.

(Remember that $1 + 2$ has already been evaluated to 3.)

Continuations and reduction contexts

Continuations closely correspond with reduction contexts in Wright-Felleisen reduction semantics:

If $E[a]$ is a reduct of p , then the continuation of a is $\lambda x. E[x]$.

Example 3

Consider again $p = (1 + 2) * (3 + 4)$.

$$\begin{aligned} (1 + 2) * (3 + 4) &= E[1 + 2] \text{ with } E = [] * (3 + 4) \\ &\rightarrow 3 * (3 + 4) = E'[3 + 4] \text{ with } E' = 3 * [] \\ &\rightarrow 3 * 7 \rightarrow 21 \end{aligned}$$

The continuation of $1 + 2$ is $\lambda x. E[x] = \lambda x. x * (3 + 4)$.

The continuation of $3 + 4$ is $\lambda x. E'[x] = \lambda x. 3 * x$.

Continuations as first-class values

The Scheme language offers a primitive `callcc` (call with current continuation) that enables a subexpression a of the program to capture its continuation (as a function 'value of a ' \mapsto 'value of the program') and manipulate this continuation as a first-class value.

The expression `callcc ($\lambda k.a$)` evaluates as follows:

- The continuation of this expression is passed as argument to $\lambda k.a$.
- Evaluation of a proceeds; its value is the value of `callcc ($\lambda k.a$)`.
- If, during the evaluation of a **or later**, we evaluate `throw k v` , evaluation continues as if `callcc ($\lambda k.a$)` returned v .

That is, the continuation of the `callcc` expression is reinstalled and restarted with v as the result provided by this expression.

Using first-class continuations

Libraries for lists, sets, and other collection data types often provide an imperative iterator `iter`, e.g.

```
(* list_iter: ('a -> unit) -> 'a list -> unit *)

let rec list_iter f l =
  match l with
  | [] -> ()
  | head :: tail -> f head; list_iter f tail
```

Using first-class continuations

Using first-class continuations, an existing imperative iterator can be turned into a function that returns the first element of a collection satisfying a given predicate `pred`.

```
let find pred lst =
  callcc (λk.
    list_iter
      (λx. if pred x then throw k (Some x) else ())
    lst;
  None)
```

If an element `x` is found such that `pred x = true`, the `throw` causes `Some x` to be returned immediately as the result of `find pred lst`. If no such element exists, `list_iter` terminates normally, and `None` is returned.

Using first-class continuations

The previous example can also be implemented with exceptions. However, `callcc` adds the ability to **backtrack** the search.

```
let find pred lst =
  callcc (λk.
    list_iter
      (λx. if pred x
           then callcc (λk'. throw k (Some(x, k')))
           else ()))
    lst;
  None)
```

When `x` is found such that `pred x = true`, `find` returns not only `x` but also a continuation `k'` which, when thrown, will cause backtracking: the search in `lst` restarts at the element following `x`.

Using first-class continuations

The following use of `find` will print all list elements satisfying the predicate:

```
let printall pred lst =
  match find pred list with
  | None -> ()
  | Some(x, k) -> print_string x; throw k ()
```

The `throw k ()` restarts `find pred list` where it left the last time.

First-class continuations

`callcc` and other control operators are difficult to use directly (“the `goto` of functional languages”), but in combination with references, can implement a variety of interesting control structures:

- Exceptions.
- Backtracking.
- Imperative iterators (such as Python’s `yield`).
- Checkpointing and replay debugging.
- Coroutines / cooperative multithreading (next slide).

Implementing coroutines with continuations

```
callcc (fun init_k ->
  let curr_k = ref init_k in
  let communicate x =
    callcc (fun k ->
      let old_k = !curr_k in curr_k := k; throw old_k x) in
  let rec process1 n =
    print_string "1: received "; print_int n; print_newline();
    process1(communicate(n+1))
  and process2 n =
    print_string "2: received "; print_int n; print_newline();
    process2(communicate(n+1)) in
  process1(callcc(fun start1 ->
    process2(callcc(fun start2 ->
      curr_k := start2; throw start1 0))))))
```

Control operators and classical logic

Control operators such as `callcc` can also be used to provide a Curry-Howard correspondence for classical logic, i.e. define the computational content of the excluded middle axiom $\forall P. P \vee \neg P$.

Modulo Curry-Howard, this axiom corresponds to the type $\forall P. P + (P \rightarrow \text{False})$, where `False` is an empty type and $A + B$ is a datatype with two constructors

`Left` : $A \rightarrow A + B$ and `Right` : $B \rightarrow A + B$.

The following term “implements” excluded middle:

$$\text{callcc}(\lambda k. \text{Right}(\lambda p. \text{throw } k (\text{Left}(p))))$$

See *A Formulæ-as-Types Notion of Control*, T. Griffin, Symp. Principles of Programming Languages 1990.

Reduction semantics for continuations

In Wright and Felleisen’s style, keep the same head reductions $\xrightarrow{\varepsilon}$ and the same context rule as before, and add two whole-program reduction rules (\rightarrow) for `callcc` and `throw`:

$$E[\text{callcc } v] \rightarrow E[v (\lambda x. E[x])]$$

$$E[\text{throw } k v] \rightarrow k v$$

Same evaluation contexts E as before.

Example of reductions

$$\begin{aligned}
 & E[\text{callcc } (\lambda k. 1 + \text{throw } k \ 0)] \\
 & \rightarrow E[(\lambda k. 1 + \text{throw } k \ 0) (\lambda x. E[x])] \\
 & \rightarrow E[1 + \text{throw } (\lambda x. E[x]) \ 0] \\
 & \rightarrow (\lambda x. E[x]) \ 0 \\
 & \rightarrow E[0]
 \end{aligned}$$

Note how `throw` discards the current context $E[1 + []]$ and reinstalls the saved context E instead.

Conversion to continuation-passing style (CPS)

Goal: make explicit the handling of continuations.

Input: a call-by-value functional language with `callcc`.

Output: a call-by-value *or call-by-name*, pure functional language (no `callcc`).

Idea: every term a becomes a function $\lambda k \dots$ that receives its continuation k as an argument, computes the value v of a , and finishes by applying k to v .

Uses: compilation of `callcc`; semantics; programming with continuations in Caml, Haskell, ...

CPS conversion: Core constructs

$$\begin{aligned} \llbracket N \rrbracket &= \lambda k. k N \\ \llbracket x \rrbracket &= \lambda k. k x \\ \llbracket \lambda x. a \rrbracket &= \lambda k. k (\lambda x. \llbracket a \rrbracket) \\ \llbracket \text{let } x = a \text{ in } b \rrbracket &= \lambda k. \llbracket a \rrbracket (\lambda x. \llbracket b \rrbracket k) \\ \llbracket a b \rrbracket &= \lambda k. \llbracket a \rrbracket (\lambda v_a. \llbracket b \rrbracket (\lambda v_b. v_a v_b k)) \end{aligned}$$

A function $\lambda x. a$ becomes a function of two arguments, x and the continuation k that will receive the value of a .

Effect on types: if $a : \tau$ then $\llbracket a \rrbracket : \llbracket \tau \rrbracket$

where $\llbracket \tau \rrbracket = (\tau \rightarrow \text{answer}) \rightarrow \text{answer}$ for base types τ

and $\llbracket \tau_1 \rightarrow \tau_2 \rrbracket = ((\tau_1 \rightarrow \llbracket \tau_2 \rrbracket)) \rightarrow \text{answer} \rightarrow \text{answer}$.

CPS conversion: Continuation operators

$$\begin{aligned} \llbracket \text{callcc } a \rrbracket &= \lambda k. \llbracket a \rrbracket k k \\ \llbracket \text{throw } a b \rrbracket &= \lambda k. \llbracket a \rrbracket (\lambda v_a. \llbracket b \rrbracket (\lambda v_b. v_a v_b)) \end{aligned}$$

In `callcc a`, the function value of a receives the current continuation k both as its argument and as its continuation.

In `throw a b`, we discard the current continuation k and apply directly the value of a (which is a continuation captured by `callcc`) to the value of b .

Administrative reductions

The CPS translation $\llbracket \cdot \cdot \cdot \rrbracket$ produces terms that are more verbose than one would naturally write by hand, e.g. in the case of an application of a variable to a variable:

$$\llbracket f \ x \rrbracket = \lambda k. (\lambda k_1. k_1 \ f) (\lambda v_1. (\lambda k_2. k_2 \ x) (\lambda v_2. v_1 \ v_2 \ k))$$

instead of the more natural $\lambda k. f \ x \ k$.

This clutter can be eliminated by performing β reductions at transformation time to eliminate the “administrative redexes” introduced by the translation. In particular, we have

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

whenever v is a value or variable.

Examples of CPS translation

$$\begin{aligned} \llbracket f(f \ x) \rrbracket \\ = \lambda k. f \ x \ (\lambda v. f \ v \ k)) \end{aligned}$$

$$\begin{aligned} \llbracket \mu fact. \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n * fact(n - 1) \rrbracket \\ = \lambda k_0. k_0(\\ \mu fact. \lambda n. \lambda k. \text{if } n = 0 \text{ then } k \ 1 \text{ else } fact \ (n - 1) \ (\lambda v. k \ (n * v))) \end{aligned}$$

Execution of CPS-converted programs

Execution of a program $prog$ is achieved by applying its CPS conversion to the initial continuation $\lambda x.x$:

$$\llbracket prog \rrbracket (\lambda x.x)$$

Theorem 4

If $a \xrightarrow{*} N$, then $\llbracket a \rrbracket (\lambda x.x) \xrightarrow{*} N$.

Proof.

See lecture IV (monadic transformations). □

CPS terms

The λ -terms produced by the CPS transformation have a very specific shape, described by the following grammar:

CPS atom: $atom ::= x \mid N \mid \lambda v. body \mid \lambda x. \lambda k. body$

CPS body: $body ::= atom \mid atom_1 atom_2 \mid atom_1 atom_2 atom_3$

$\llbracket a \rrbracket$ is an *atom*, and $\llbracket a \rrbracket (\lambda x.x)$ is a *body*.

Reduction of CPS terms

CPS atom: $atom ::= x \mid N \mid \lambda v. body \mid \lambda x. \lambda k. body$

CPS body: $body ::= atom \mid atom_1 atom_2 \mid atom_1 atom_2 atom_3$

Note that all applications (unary or binary) are in tail-position and at application-time, their arguments are closed atoms, that is, values.

The following reduction rules suffice to evaluate CPS-converted programs:

$$\begin{aligned} (\lambda x. \lambda k. body) atom_1 atom_2 &\rightarrow body[x \leftarrow atom_1, k \leftarrow atom_2] \\ (\lambda v. body) atom &\rightarrow body[v \leftarrow atom] \end{aligned}$$

These reductions are always applied at the top of the program — there is no need for reduction under a context.

The Indifference theorem

(G. Plotkin, 1975)

Theorem 5 (Indifference)

A closed CPS-converted program $\llbracket a \rrbracket (\lambda x. x)$ evaluates in the same way in call-by-name, in left-to-right call-by-value, and in right-to-left call-by-value.

Proof.

Since closed atoms are values, the reduction rules

$$\begin{aligned} (\lambda x. \lambda k. body) atom_1 atom_2 &\rightarrow body[x \leftarrow atom_1, k \leftarrow atom_2] \\ (\lambda v. body) atom &\rightarrow body[v \leftarrow atom] \end{aligned}$$

are admissible both under call-by-value and call-by-name. Since we do not reduce under application nodes, left-to-right or right-to-left evaluation of application makes no difference. \square

CPS conversion and reduction strategy

CPS conversion encodes the reduction strategy in the structure of the converted terms. For instance, right-to-left call-by-value is obtained by taking

$$\llbracket a \ b \rrbracket = \lambda k. \llbracket b \rrbracket (\lambda v_b. \llbracket a \rrbracket (\lambda v_a. v_a \ v_b \ k))$$

and call-by-name is achieved by taking

$$\begin{aligned} \llbracket x \rrbracket &= \lambda k. x \ k \\ \llbracket a \ b \rrbracket &= \lambda k. \llbracket a \rrbracket (\lambda v_a. v_a \ \llbracket b \rrbracket \ k) \end{aligned}$$

Compilation of CPS terms

CPS terms can be executed by a **stackless** abstract machines with components

- a code pointer c
- an environment e
- three registers R_1, R_2, R_3 .

Instruction set:

$\text{ACCESS}_i(n)$	store n -th field of the environment in R_i
$\text{CONST}_i(N)$	store the integer N in R_i
$\text{CLOSURE}_i(c)$	store closure of c in R_i
TAILAPPLY1	apply closure in R_1 to argument R_2
TAILAPPLY2	apply closure in R_1 to arguments R_2, R_3

Compilation of CPS terms

Compilation of atoms $\mathcal{A}_i(atom)$ (leaves the value of $atom$ in R_i):

$$\begin{aligned}\mathcal{A}_i(\underline{n}) &= \text{ACCESS}_i(n) \\ \mathcal{A}_i(N) &= \text{CONST}_i(N) \\ \mathcal{A}_i(\lambda^1.a) &= \text{CLOSURE}_i(\mathcal{B}(a)) \\ \mathcal{A}_i(\lambda^2.a) &= \text{CLOSURE}_i(\mathcal{B}(a))\end{aligned}$$

Compilation of bodies $\mathcal{B}(body)$:

$$\begin{aligned}\mathcal{B}(a) &= \mathcal{A}_1(a) \\ \mathcal{B}(a_1 a_2) &= \mathcal{A}_1(a_1); \mathcal{A}_2(a_2); \text{TAILAPPLY1} \\ \mathcal{B}(a_1 a_2 a_3) &= \mathcal{A}_1(a_1); \mathcal{A}_2(a_2); \mathcal{A}_3(a_3); \text{TAILAPPLY2}\end{aligned}$$

Transitions of the CPS abstract machine

Machine state before					Machine state after				
Code	Env	R_1	R_2	R_3	Code	Env	R_1	R_2	R_3
TAILAPPLY1; c	e	$c'[e']$	v	-	c'	$v.e'$	-	-	-
TAILAPPLY2; c	e	$c'[e']$	v_1	v_2	c'	$v_2.v_1.e'$	-	-	-
ACCESS ₁ (n); c	e	-	v_2	v_3	c	e	$e(n)$	v_2	v_3
CONST ₁ (n); c	e	-	v_2	v_3	c	e	N	v_2	v_3
CLOSURE ₁ (c'); c	e	-	v_2	v_3	c	e	$c'[e]$	v_2	v_3

(Similarly for the other ACCESS, CONST and CLOSURE instructions.)

Continuations vs. stacks

That CPS terms can be executed without a stack is not surprising, given that the stack of a machine such as the Modern SECD is isomorphic to the current continuation in a CPS-based approach.

$$f\ x = 1 + g\ x \qquad g\ x = 2 * h\ x \qquad h\ x = \dots$$

Consider the execution point where `h` is entered. In the CPS model, the continuation at this point is

$$k = \lambda v. k' (2 * v) \text{ with } k' = \lambda v. k'' (1 + v) \text{ and } k'' = \lambda v. v$$

In the Modern SECD model, the stack at this point is

$$\underbrace{(\text{MUL}; \text{RETURN}).e_g.2}_{\approx k} . \underbrace{(\text{ADD}; \text{RETURN}).e_f.1}_{\approx k'} . \underbrace{\varepsilon.\varepsilon}_{\approx k''}$$

Continuations vs. stacks

At the machine level, stacks and continuations are two ways to represent the **call chain**: the chain of function calls currently active.

- Continuations: as a singly-linked list of heap-allocated closures, each closure representing a function activation. These closures are reclaimed by the garbage collector.
- Stacks: as contiguous blocks in a memory area outside the heap, each block representing a function activation. These blocks are explicitly deallocated by `RETURN` instructions.

Stacks are more efficient in terms of GC costs and memory locality, but need to be copied in full to implement `callcc`.

Compiling with continuations, A. Appel, Cambridge University Press, 1992.