Functional programming languages
Part III: program transformations

Xavier Leroy

INRIA Paris-Rocquencourt

MPRI 2-4, 2016–2017

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
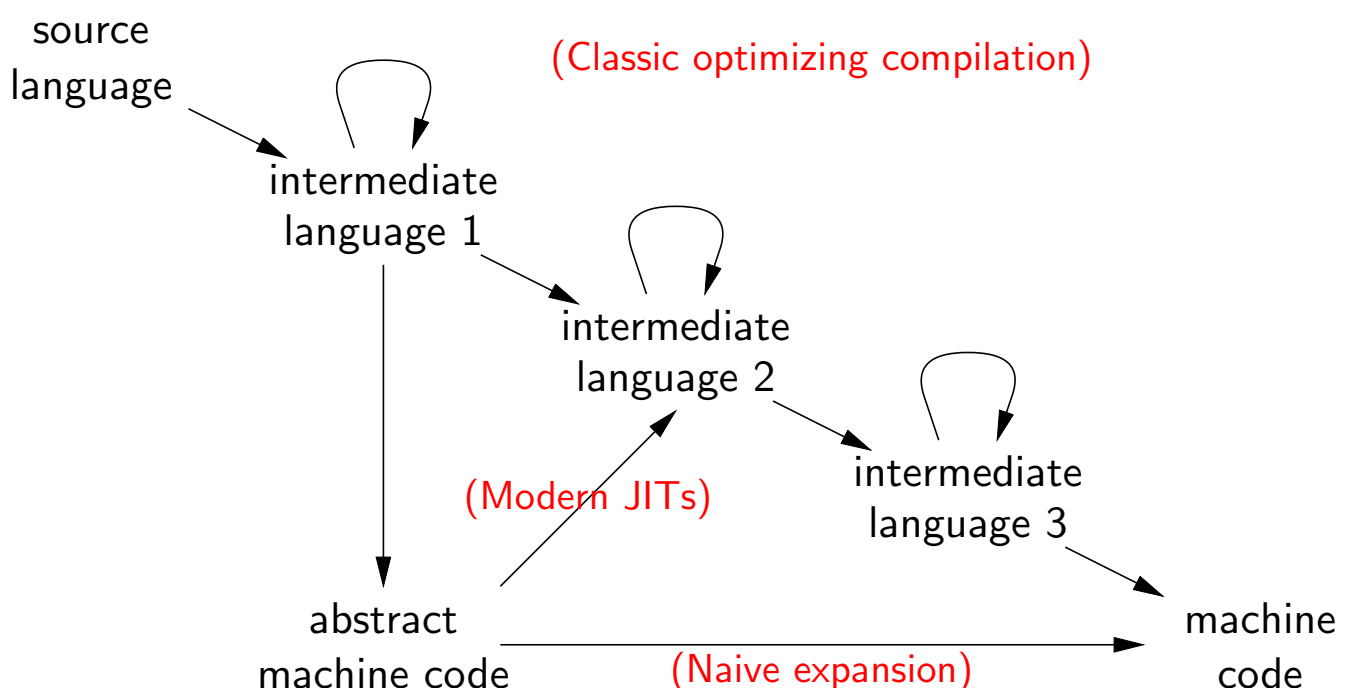
Outline

Closure conversion

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

Input: a 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

$$\llbracket x \rrbracket = x$$

$$\llbracket \lambda x.a \rrbracket = \texttt{tuple}(\lambda c, x.\ \texttt{let } x_1 = \texttt{field}_1(c) \texttt{ in}$$
$$\cdots$$
$$\texttt{let } x_n = \texttt{field}_n(c) \texttt{ in}$$
$$\llbracket a \rrbracket,$$
$$x_1, \ldots, x_n)$$

where $x_1, \ldots, x_n$ are the free variables of $\lambda x.a$

$$\llbracket a\ b \rrbracket = \texttt{let } c = \llbracket a \rrbracket \texttt{ in } \texttt{field}_0(c)(c, \llbracket b \rrbracket)$$

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

$$\llbracket \texttt{let } x = a \texttt{ in } b \rrbracket = \texttt{let } x = \llbracket a \rrbracket \texttt{ in } \llbracket b \rrbracket$$
$$\llbracket a + b \rrbracket = \llbracket a \rrbracket + \llbracket b \rrbracket$$

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.

$$
\begin{aligned}
[\![\mu f.\lambda x.a]\!] \;=\; \texttt{tuple}(\lambda f, x.\; &\texttt{let } x_1 = \texttt{field}_1(f) \texttt{ in} \\
&\cdots \\
&\texttt{let } x_n = \texttt{field}_n(f) \texttt{ in} \\
&[\![a]\!], \\
x_1, \ldots, x_n)&
\end{aligned}
$$

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

Note that regular functions $\lambda x.a$ are converted exactly like pseudo-recursive functions $\mu f.\lambda x.a$ where f 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:

```
let l = <big list> in λx. 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:

$$
\begin{aligned}
[\![x]\!] &= x \\
[\![\lambda x.a]\!] &= \texttt{new } C_{\lambda x.a}(x_1, \ldots, x_n) \\
&\qquad \text{where } x_1, \ldots, x_n \text{ are the free variables of } \lambda x.a \\
[\![a\ b]\!] &= [\![a]\!].\texttt{apply}([\![b]\!])
\end{aligned}
$$

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 Cλx.a {

    Object x1; ...; Object xn;

    Cλx.a(Object x1, ..., Object xn) {
        this.x1 = x1; ...; this.xn = xn;
    }

    Object apply(Object x) {
        return [[a]];
    }
}
```

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:

$$
\begin{aligned}
[\![fun\ arg]\!] &= \texttt{let } c = [\![fun]\!] \texttt{ in } \texttt{field}_0(c)(c, [\![arg]\!]) \\
[\![obj.meth(arg)]\!] &= \texttt{let } o = [\![obj]\!] \texttt{ in } o.meth(o, [\![arg]\!])
\end{aligned}
$$

Outline

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 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 $C(v_1, \ldots, v_n)$ where the constructor C uniquely identifies the function, and the constructor arguments $v_1, \ldots, v_n$ are the values of the free variables $x_1, \ldots, x_n$.

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

Definition of defunctionalization

$$
\begin{aligned}
[\![x]\!] &= x \\
[\![\lambda x.a]\!] &= C_{\lambda x.a}(x_1, \ldots, x_n) \\
&\quad \text{where } x_1, \ldots, x_n \text{ are the free variables of } \lambda x.a \\
[\![\mu f.\lambda x.a]\!] &= C_{\mu f.\lambda x.a}(x_1, \ldots, x_n) \\
&\quad \text{where } x_1, \ldots, x_n \text{ are the free variables of } \mu f.\lambda x.a \\
[\![a\ b]\!] &= \mathtt{apply}([\![a]\!], [\![b]\!])
\end{aligned}
$$

(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_{λx.a}(x₁, . . . , xₙ) → let x = arg in ⟦a⟧
    |  C_{μf'.λx'.a'}(x'₁, . . . , x'ₙ') → let f' = fun in let x' = arg in ⟦a'⟧
    |  . . .
in ⟦program⟧
```

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

Example

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

```
        let rec apply (fn, arg) =
            match fn 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

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` $\to b$, 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 Felleisen style, add two head reduction rules for `try...with` and a generic exception propagation rule:

$$(\texttt{try } v \texttt{ with } x \to b) \ \xrightarrow{\varepsilon} \ v$$
$$(\texttt{try raise } v \texttt{ with } x \to b) \ \xrightarrow{\varepsilon} \ b[x \leftarrow v]$$
$$F[\texttt{raise } v] \ \xrightarrow{\varepsilon} \ \texttt{raise } v \quad \text{if } F \neq []$$

Exception propagation contexts F are like reduction contexts E but do not allow crossing a `try...with`

Reduction contexts:
$$E ::= [] \mid E \ b \mid v \ E \mid \texttt{try } E \texttt{ with } x \to b \mid \ldots$$
Exception propagation contexts:
$$F ::= [] \mid F \ b \mid v \ F \mid \ldots$$

Reduction semantics for exceptions

Assume the current program is $p = E[\texttt{raise } v]$, 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

$$p = E'[\texttt{try } F[\texttt{raise } v] \texttt{ with } x \to b]$$

where F contains no `try...with` constructs.
$F[\texttt{raise } v]$ head-reduces to `raise v`, and $E'[\texttt{try } [] \texttt{ with } x \to b]$ is an evaluation context. The reduction sequence is therefore:

$$
\begin{aligned}
p = E'[\texttt{try } F[\texttt{raise } v] \texttt{ with } x \to b] \quad &\to \quad E'[\texttt{try raise } v \texttt{ with } x \to b] \\
&\to \quad E'[b[x \leftarrow v]]
\end{aligned}
$$

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

$$p = E[\texttt{raise } v] \to \texttt{raise } v$$

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: $a \xrightarrow{*} \texttt{raise } v$
- Divergence: $a \xrightarrow{*} a' \to \dots$
- Error: $a \xrightarrow{*} a' \nrightarrow$ where $a \neq v$ and $a \neq \texttt{raise } v$.

Natural semantics for exceptions

In natural semantics, the evaluation relation becomes $a \Rightarrow r$
where evaluation results are $r ::= v \mid \mathtt{raise}\ v$.
Add the following rules for `try...with`:

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

as well as exception propagation rules such as:

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

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 to a value v
- $E(v)$ 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

$$
\begin{aligned}
[\![N]\!] &= V(N) \\
[\![x]\!] &= V(x) \\
[\![\lambda x.a]\!] &= V(\lambda x.\ [\![a]\!]) \\
[\![\texttt{let } x = a \texttt{ in } b]\!] &= \texttt{match } [\![a]\!] \texttt{ with } E(x) \to E(x) \mid V(x) \to [\![b]\!] \\
[\![a\ b]\!] &= \texttt{match } [\![a]\!] \texttt{ with} \\
&\quad \mid\ E(e_a) \to E(e_a) \\
&\quad \mid\ V(v_a) \to \\
&\qquad \texttt{match } [\![b]\!] \texttt{ with } E(e_b) \to E(e_b) \mid V(v_b) \to v_a\ v_b
\end{aligned}
$$

Effect on types: if $a : \tau$ then $[\![a]\!] : [\![\tau]\!]$ outcome
where $[\![\tau_1 \to \tau_2]\!] = [\![\tau_1]\!] \to [\![\tau_2]\!]$ outcome and $[\![\tau]\!] = \tau$ for base types
and where type `'a outcome = V of 'a | E of exn.`

Definition of the conversion: Exception-specific constructs

$$
[\![\texttt{raise } a]\!] = \texttt{match } [\![a]\!] \texttt{ with } E(e_a) \to E(e_a) \mid V(v_a) \to E(v_a)
$$

$$
[\![\texttt{try } a \texttt{ with } x \to b]\!] = \texttt{match } [\![a]\!] \texttt{ with } E(x) \to [\![b]\!] \mid V(v_a) \to V(v_a)
$$

Example of conversion

```
⟦try fn arg with exn → 0⟧ =
    match
        match V(fn) 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 $V(\ldots)$ or $E(\ldots)$ is known at compile-time.

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

$$(\texttt{match } E(v) \texttt{ with } E(x) \to b \mid V(x) \to c) \quad \overset{adm}{\rightarrow} \quad b[x \leftarrow v]$$

$$(\texttt{match } V(v) \texttt{ with } E(x) \to b \mid V(x) \to c) \quad \overset{adm}{\rightarrow} \quad c[x \leftarrow v]$$

Example of conversion

After application of administrative reductions, we obtain:

```
⟦try fn arg with exn → 0⟧ =
    match fn 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 $a \Rightarrow \mathtt{raise}\ v$, 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 $\llbracket a[x \leftarrow v] \rrbracket = \llbracket a \rrbracket [x \leftarrow \llbracket v \rrbracket_v]$. $\qquad\square$

Outline

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 (recap)

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
```

Uses of references

Memoization = caching of already-computed results.

```
let fib_cache = ref map_empty

let rec fib n =
  match map_find n !fib_cache with
  | Some res -> res
  | None ->
      let res =
        if n < 2 then 1 else fib(n-1) + fib(n-2) in
      fib_cache := map_add n res !fib_cache;
      res
```

Lazy evaluation = CBV + references

Implementation of lazy evaluation:

```
type 'a state =
  | Unevaluated of unit -> 'a
  | Evaluated of 'a

type 'a lazy = ('a state) ref

let force (lz: 'a lazy) : 'a =
  match !lz with
  | Evaluated v -> v
  | Unevaluated f ->
      let v = f() in lz := Evaluated v; v
```

$+$ syntactic sugar: $\texttt{lazy}\ e \equiv \texttt{ref}(\texttt{Unevaluated}(\texttt{fun}\ () \rightarrow e))$

Semantics of references

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

$$\texttt{let}\ r = \texttt{ref}\ 1\ \texttt{in}\ r := 2; !r \quad \not\to \quad (\texttt{ref}\ 1) := 2; !(\texttt{ref}\ 1)$$

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 locations.

Reduction semantics for references

Reduction over configurations $a \, / \, s$: pairs of a term a and a store s mapping locations to values.

One-step reduction $a \, / \, s \to a' \, / \, s'$
(read: in initial store s, a reduces to a' and updates the store to s')

$$
\begin{aligned}
(\lambda x.a) \, v \, / \, s &\xrightarrow{\varepsilon} a[x \leftarrow v] \, / \, s \\
\mathtt{ref} \, v \, / \, s &\xrightarrow{\varepsilon} \ell \, / \, (s + \ell \mapsto v) \quad \text{where } \ell \notin Dom(s) \\
\mathtt{deref} \, \ell \, / \, s &\xrightarrow{\varepsilon} s(\ell) \, / \, s \\
\mathtt{assign} \, \ell \, v \, / \, s &\xrightarrow{\varepsilon} (\,) \, / \, (s + \ell \mapsto v)
\end{aligned}
$$

$$
\frac{a \, / \, s \xrightarrow{\varepsilon} a' \, / \, s'}{E(a) \, / \, s \to E(a') \, / \, s'} \quad \text{(context)}
$$

Example of reduction sequence

In red: the active redex at every step.

$$
\begin{aligned}
&\mathtt{let} \, r = \mathtt{ref} \, 3 \, \mathtt{in} \, \mathtt{let} \, x = r := \,!r + 1 \, \mathtt{in} \, !r \, / \, \emptyset \\
\to \quad &\mathtt{let} \, r = \ell \, \mathtt{in} \, \mathtt{let} \, x = r := \,!r + 1 \, \mathtt{in} \, !r \, / \, \{\ell \mapsto 3\} \\
\to \quad &\mathtt{let} \, x = \ell := \,!\ell + 1 \, \mathtt{in} \, !\ell \, / \, \{\ell \mapsto 3\} \\
\to \quad &\mathtt{let} \, x = \ell := 3 + 1 \, \mathtt{in} \, !\ell \, / \, \{\ell \mapsto 3\} \\
\to \quad &\mathtt{let} \, x = \ell := 4 \, \mathtt{in} \, !\ell \, / \, \{\ell \mapsto 3\} \\
\to \quad &\mathtt{let} \, x = (\,) \, \mathtt{in} \, !\ell \, / \, \{\ell \mapsto 4\} \\
\to \quad &!\ell \, / \, \{\ell \mapsto 4\} \\
\to \quad &4
\end{aligned}
$$

Conversion to state-passing style

Goal: get rid of state.

Input: a functional language with imperative 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

$$
\begin{aligned}
[\![N]\!] &= \lambda s.(N, s) \\
[\![x]\!] &= \lambda s.(x, s) \\
[\![\lambda x.a]\!] &= \lambda s.(\lambda x.[\![a]\!],\ s) \\
[\![\texttt{let } x = a \texttt{ in } b]\!] &= \lambda s.\ \texttt{match } [\![a]\!]\ s \texttt{ with } (x, s') \to [\![b]\!]\ s' \\
[\![a\ b]\!] &= \lambda s.\ \texttt{match } [\![a]\!]\ s \texttt{ with } (v_a, s') \to \\
&\qquad\qquad \texttt{match } [\![b]\!]\ s' \texttt{ with } (v_b, s'') \to v_a\ v_b\ s''
\end{aligned}
$$

Effect on types: if $a : \tau$ then $[\![a]\!] : \texttt{store} \to [\![\tau]\!] \times \texttt{store}$
where $[\![\tau_1 \to \tau_2]\!] = [\![\tau_1]\!] \to \texttt{store} \to [\![\tau_2]\!] \times \texttt{store}$
and $[\![\tau]\!] = \tau$ for base types.

Definition of the conversion:
Constructs specific to references

$$[\![\mathtt{ref}\ a]\!] \;=\; \lambda s.\,\mathtt{match}\ [\![a]\!]\ s\ \mathtt{with}\ (v_a, s') \to \mathtt{store\_alloc}\ v_a\ s'$$

$$[\![!a]\!] \;=\; \lambda s.\,\mathtt{match}\ [\![a]\!]\ s\ \mathtt{with}\ (v_a, s') \to (\mathtt{store\_read}\ v_a\ s', s')$$

$$[\![a := b]\!] \;=\; \lambda s.\,\mathtt{match}\ [\![a]\!]\ s\ \mathtt{with}\ (v_a, s') \to$$
$$\mathtt{match}\ [\![b]\!]\ s'\ \mathtt{with}\ (v_b, s'') \to (\varepsilon, \mathtt{store\_write}\ v_a\ v_b\ s'')$$

The operations `store_alloc`, `store_read` and `store_write` provide a concrete implementation of the store.

Example of conversion

Administrative reductions:

$$(\lambda s'.b)s \;\overset{adm}{\mapsto}\; b[s \leftarrow s'] \text{ if } s \text{ variable}$$
$$(\mathtt{match}\ (a, s)\ \mathtt{with}\ (x, s') \to b) \;\overset{adm}{\mapsto}\; \mathtt{let}\ x = a\ \mathtt{in}\ b[s' \leftarrow s]$$
$$\mathtt{let}\ x = v\ \mathtt{in}\ b \;\overset{adm}{\mapsto}\; b[x \leftarrow v]$$
$$\mathtt{let}\ x = y\ \mathtt{in}\ b \;\overset{adm}{\mapsto}\; b[x \leftarrow y]$$

Example of translation after administrative reductions:

```
[[ let r = ref 3 in let x = r := !r + 1 in !r ]] =
    λs. match store_alloc s 3 with (r, s1) ->
        let t = store_read r s1 in
        let u = t + 1 in
        match (ε, store_write r u s1) with (x, s2) ->
        (store_read r s2, s2)
```

An implementation of the store

Locations = integers.

Stores = pairs (location, finite map location $\mapsto$ value).

$$
\begin{aligned}
\texttt{store\_alloc}\ v\ (n, m) &= (n,\ (n+1, m + \{n \mapsto v\})) \\
\texttt{store\_read}\ \ell\ (n, m) &= m(\ell) \\
\texttt{store\_write}\ \ell\ v\ (n, m) &= (n, m + \{\ell \mapsto v\})
\end{aligned}
$$

Typing the store?

Easy: static typing of a monomorphic store, where all values stored in references are of the same type `sval`:

$$
\begin{aligned}
\texttt{store\_alloc} &: \texttt{sval} \to \texttt{store} \to \texttt{location} \times \texttt{store} \\
\texttt{store\_read} &: \texttt{location} \to \texttt{store} \to \texttt{sval} \\
\texttt{store\_write} &: \texttt{location} \to \texttt{sval} \to \texttt{store} \to \texttt{store}
\end{aligned}
$$

Much more challenging (programming exercise III.6): a type-safe implementation of a polymorphic store.

$$
\begin{aligned}
\texttt{store\_alloc} &: \forall \alpha.\ \alpha \to \texttt{store} \to \alpha\ \texttt{location} \times \texttt{store} \\
\texttt{store\_read} &: \forall \alpha.\ \alpha\ \texttt{location} \to \texttt{store} \to \alpha \\
\texttt{store\_write} &: \forall \alpha.\ \alpha\ \texttt{location} \to \alpha \to \texttt{store} \to \texttt{store}
\end{aligned}
$$

Outline

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 $\lambda x.\, 3 * x$.

(Not $\lambda x.\, (1 + 2) * x$ because $1 + 2$ has already been evaluated to 3.)

Continations and reduction contexts

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

If $p = E[a]$ and a can reduce, then the continuation of a is $\lambda x.\ E[x]$.

Example 3

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

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

The continuation of $1 + 2$ in p is $\lambda x.E[x] = \lambda x.\ x * (3 + 4)$.
The continuation of $3 + 4$ in p' 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 at any later time, 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.

The types are:

`callcc` $: \forall \alpha, (\alpha\ \texttt{cont} \to \alpha) \to \alpha$ `throw` $: \forall \alpha\beta, \alpha\ \texttt{cont} \to \alpha \to \beta.$

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 (exercise III.8)
- Backtracking.
- Imperative iterators (such as Python's `yield`).
- Checkpointing and replay debugging.
- Coroutines / cooperative multithreading (next slides).

Coroutines

Consider a simple subroutine (procedure) that does side-effects:

```
let task message increment =
  let rec loop n =
    if n >= 10 then () else begin
      print_string message; print_int n;
      loop(n + increment)
    end
  in loop 0


let _ = task " A" 1; task " B" 2; task " C" 3
```

Execution is purely sequential, producing

`A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 B0 B2 B4 B6 B8 C0 C3 C9`

Coroutines

Coroutines interleave their executions, explicitly yielding control to other coroutines.

```
let task message increment =
  fun () ->
  let rec loop n =
    if n >= 10 then terminate() else begin
      print_string message; print_int n;
      yield();
      loop(n + increment)
    end
  in loop 0

let _ = fork(task " A" 1); fork(task " B" 2); fork(task " C" 3)
```

Execution is interleaved, producing for instance

A0 B0 A1 C0 B2 A2 C3 B4 A3 C6 B6 A4 C9 B8 A5 A6 A7 A8 A9

Implementing coroutines with continuations

Maintain a queue of continuations corresponding to coroutines that have not started yet or have previously yielded control.

```
let queue = ref []

let enqueue x = queue := !queue @ [x]

let dequeue () =
  match !queue with x :: q' -> queue := q'; x | [] -> assert false

let queue_is_empty () =
  match !queue with [] -> true | x :: q' -> false
```

Implementing coroutines with continuations

`fork proc` executes `proc()` but enqueues a continuation corresponding to the computation following `fork proc`.

```
let fork (proc: unit -> unit) =
  callcc (fun k -> enqueue k; proc())
```

`yield()` yields control within its caller and restarts another suspended computation (if any) or the caller itself (otherwise).

```
let yield () =
  callcc (fun k -> enqueue k; throw (dequeue()) ())
```

`terminate()` terminates its caller, restarting another suspended computation (if any) or terminating the whole program (otherwise).

```
let terminate () =
  if queue_is_empty()
  then exit 0
  else throw (dequeue()) ()
```

Reduction semantics for continuations

In 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`:

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

$$E[\texttt{throw } k \ v] \quad \rightarrow \quad k \ v$$

Same evaluation contexts E as before.

Note the non-linear use of the context E:

- The rule for `callcc` duplicates the current context E.
- The rule for `throw` discards the current context E.

Example of reductions

$$E[\texttt{callcc } (\lambda k.\, 1 + \texttt{throw } k\ 0)]$$
$$\rightarrow\quad E[(\lambda k.\, 1 + \texttt{throw } k\ 0)\ (\lambda x.E[x])]$$
$$\rightarrow\quad E[1 + \texttt{throw } (\lambda x.E[x])\ 0]$$
$$\rightarrow\quad (\lambda x.E[x])\ 0$$
$$\rightarrow\quad E[0]$$

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 $\lambda k \ldots$ 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

$$
\begin{aligned}
[\![N]\!] &= \lambda k.\ k\ N \\
[\![x]\!] &= \lambda k.\ k\ x \\
[\![\lambda x.a]\!] &= \lambda k.\ k\ (\lambda x.\ [\![a]\!]) \\
[\![\texttt{let}\ x = a\ \texttt{in}\ b]\!] &= \lambda k.\ [\![a]\!]\ (\lambda x.\ [\![b]\!]\ k) \\
[\![a\ b]\!] &= \lambda k.\ [\![a]\!]\ (\lambda v_a.\ [\![b]\!]\ (\lambda v_b.\ v_a\ v_b\ k))
\end{aligned}
$$

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 $[\![a]\!] : ([\![\tau]\!] \rightarrow \texttt{answer}) \rightarrow \texttt{answer}$
where $[\![\tau]\!] = \tau$ for base types
and $[\![\tau_1 \rightarrow \tau_2]\!] = [\![\tau_1]\!] \rightarrow ([\![\tau_2]\!] \rightarrow \texttt{answer}) \rightarrow \texttt{answer}$.

CPS conversion: Continuation operators

$$
\begin{aligned}
[\![\texttt{callcc}\ a]\!] &= \lambda k.\ [\![a]\!]\ (\lambda f.\ f\ k\ k) \\
[\![\texttt{throw}\ a\ b]\!] &= \lambda k.\ [\![a]\!]\ (\lambda v_a.\ [\![b]\!]\ (\lambda v_b.\ v_a\ v_b))
\end{aligned}
$$

In `callcc` a, the function value f 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.

Note (again) the non-linear use of the continuation k: `callcc` duplicates k, and `throw` ignores k,

Administrative reductions

The CPS translation $[\![\cdots]\!]$ 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:

$$[\![f\ x]\!] = \lambda k.\ (\lambda k_1.k_1\ f)\ (\lambda v_1.\ (\lambda k_2.k_2\ x)\ (\lambda v_2.\ v_1\ v_2\ k))$$

instead of 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

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

whenever v is a value or variable.

Examples of CPS translation

$[\![f(f\ x)]\!]$
$\quad = \quad \lambda k.\ f\ x\ (\lambda v.\ f\ v\ k)))$

$[\![\mu fact.\ \lambda n.\ \texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ n * fact(n - 1)]\!]$
$\quad = \quad \lambda k_0.\ k_0($
$\qquad \mu fact.\ \lambda n.\ \lambda k.\ \texttt{if}\ n = 0\ \texttt{then}\ k\ 1\ \texttt{else}\ fact\ (n - 1)\ (\lambda v.\ k\ (n * v)))$

$[\![\texttt{callcc}\ (\lambda k.a)]\!]$
$\quad = \quad \lambda k.\ [\![a]\!]\ k$

Execution of CPS-converted programs

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

$$[\![prog]\!]\,(\lambda x.x)$$

Theorem 4

If $a \xrightarrow{*} v$, then $[\![a]\!]\,(\lambda x.x) \xrightarrow{*} [\![v]\!]_v$. If a diverges, so does $[\![a]\!]\,(\lambda x.x)$.

Where the CPS translation of values v is:

$$[\![N]\!]_v = N \qquad [\![\lambda x.a]\!]_v = \lambda x.[\![a]\!]$$

Plotkin's proof

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

A difficult proof based on the following simulation diagram:
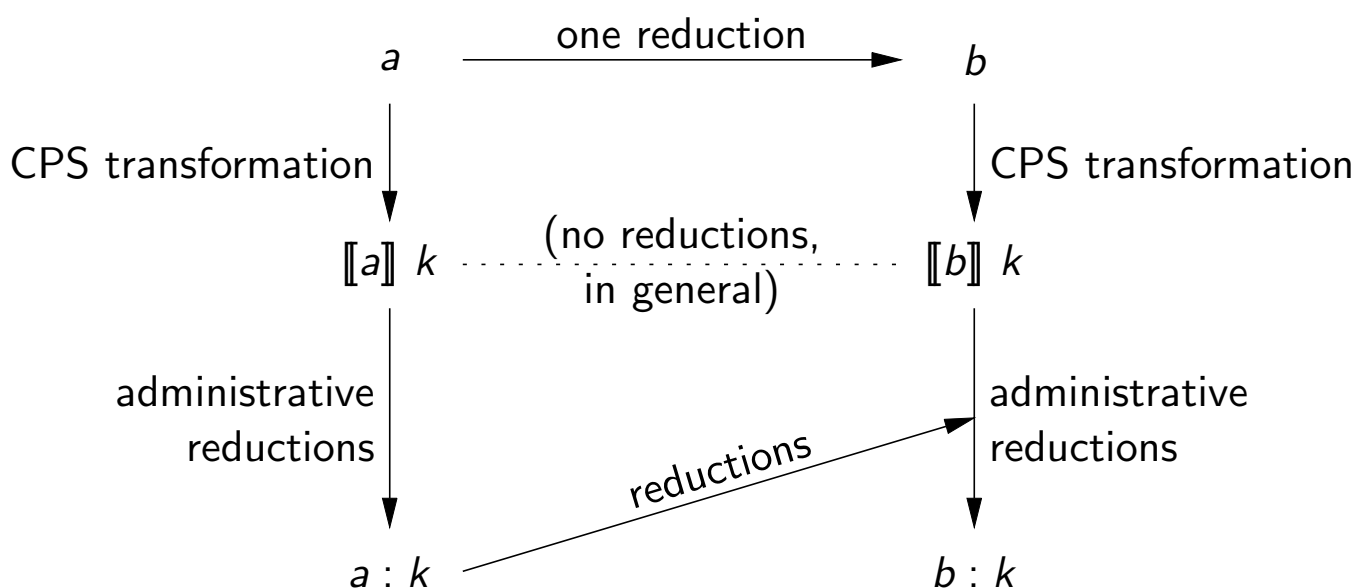
$a : k$, the "colon translation", reduces well-chosen administrative redexes.

A proof using natural semantics

These difficulties with administrative reductions can be avoided by using natural semantics in the premises:

> ## Theorem 5
>
> If $a \Rightarrow v$ and k is a value, then $[\![a]\!]\ k \xrightarrow{+} k\ [\![v]\!]_v$.
> If $a \Rightarrow \infty$ and k is a value, then $[\![a]\!]\ k$ reduces infinitely.

> ## Proof.
>
> Part 1 is by induction on a derivation of $a \Rightarrow v$. (Exercise.)
> For part 2, consider the set $X = \{[\![a]\!]\ k \mid a \Rightarrow \infty \wedge k\ \text{value}\}$ and show that
> $\forall a \in X, \exists a' \in X, a \xrightarrow{+} a'$, using part 1.
> The two proofs are suprisingly similar to the proofs of thm 10 and 17 in lecture II
> (correctness of the Modern SECD using natural semantics). $\qquad\square$

Danvy and Nielsen's one-pass CPS transformation

Split source terms a into:

Atoms: $\qquad t ::= N \mid x \mid \lambda x.a$

Serious terms: $\quad s ::= a_1\ a_2$

The one-pass CPS transformation:

$$
\begin{aligned}
[\![t]\!]\ k &= k\ \Psi(t) & \Psi(N) &= N \\
[\![t_1\ t_2]\!]\ k &= \Psi(t_1)\ \Psi(t_2)\ k & \Psi(x) &= x \\
[\![s_1\ t_2]\!]\ k &= [\![s_1]\!]\ (\lambda v_1.\ v_1\ \Psi(t_2)\ k) & \Psi(\lambda x.a) &= \lambda x.\ \lambda k.\ [\![a]\!]\ k \\
[\![t_1\ s_2]\!]\ k &= [\![s_2]\!]\ (\lambda v_2.\ \Psi(t_1)\ v_2\ k) \\
[\![s_1\ s_2]\!]\ k &= [\![s_1]\!]\ (\lambda v_1.\ [\![s_2]\!]\ (\lambda v_2.\ v_1\ v_2\ k))
\end{aligned}
$$

Danvy and Nielsen's one-pass CPS transformation

Not only does this CPS transformation produce terms free of administrative redexes, but it also enjoys a simple simulation diagram:

$$
\begin{array}{ccc}
a & \xrightarrow{\text{one reduction}} & b \\
\Big\downarrow \text{CPS transformation} & & \Big\downarrow \text{CPS transformation} \\
[\![a]\!]\ k & \xrightarrow[\text{reductions}]{\text{one or several}} & [\![b]\!]\ k
\end{array}
$$
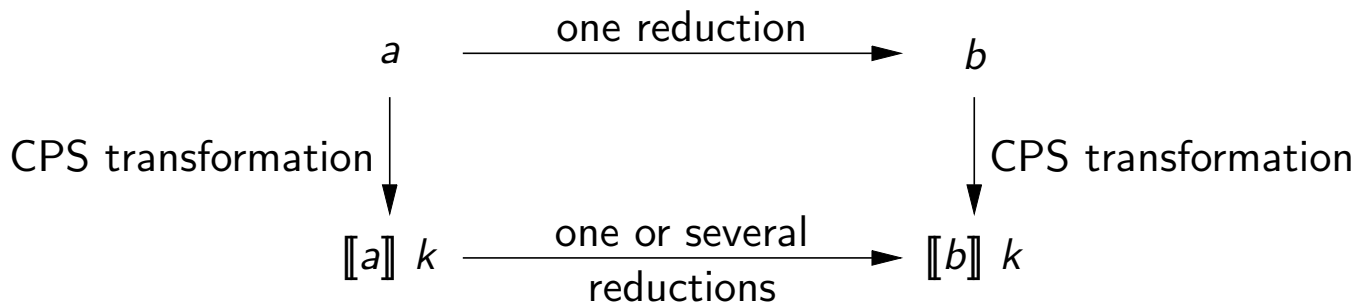
CPS terms

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

$[\![a]\!]$ is an *atom*, and $[\![a]\!]\ (\lambda x.x)$ 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, *Call-by-name, call-by-value and the lambda-calculus*, TCS 1(2), 1975

Theorem 6 (Indifference)

A closed CPS-converted program $[\![a]\!]\ (\lambda x.x)$ 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

$$[\![a\ b]\!] \quad = \quad \lambda k.\ [\![b]\!]\ (\lambda v_b.\ [\![a]\!]\ (\lambda v_a.\ v_a\ v_b\ k))$$

and call-by-name is achieved by taking

$$[\![x]\!] \quad = \quad \lambda k.\ x\ k \qquad (\text{or just } [\![x]\!] = x)$$
$$[\![a\ b]\!] \quad = \quad \lambda k.\ [\![a]\!]\ (\lambda v_a.\ v_a\ [\![b]\!]\ k)$$

Note change of viewpoint:

- in CBV, a variable is a value, so we apply the continuation to this value ($[\![x]\!] = \lambda k.\ k\ x$)

- in CBN, a variable is a suspended computation, so we apply this computation to the continuation ($[\![x]\!] = \lambda k.\ x\ k$)

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:

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

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

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

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

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
$\texttt{TAILAPPLY1}; c$	e	$c'[e']$	v	_	c'	$v.e'$	_	_	_
$\texttt{TAILAPPLY2}; c$	e	$c'[e']$	v_1	v_2	c'	$v_2.v_1.e'$	_	_	_
$\texttt{ACCESS}_1(n); c$	e	_	v_2	v_3	c	e	$e(n)$	v_2	v_3
$\texttt{CONST}_1(n); c$	e	_	v_2	v_3	c	e	N	v_2	v_3
$\texttt{CLOSURE}_1(c'); c$	e	_	v_2	v_3	c	e	$c'[e]$	v_2	v_3

(Similarly for the other $\texttt{ACCESS}$, $\texttt{CONST}$ and $\texttt{CLOSURE}$ 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.

$$\texttt{f x = 1 + g x} \qquad \texttt{g x = 2 * h x} \qquad \texttt{h x = ...}$$

Consider the execution point where $\texttt{h}$ 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

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

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 $\texttt{RETURN}$ instructions.

Stacks are more efficient in terms of GC costs and memory locality, but need to be copied in full to implement $\texttt{callcc}$.

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

Outline

Making functions tail-recursive

Contemporary OS put arbitrary, rather low limitations on stack size.

$\rightarrow$ Sometimes, programmers need to rewrite recursive functions so that they are tail recursive and run in constant stack space.

A systematic way to do so:
CPS conversion followed by defunctionalization!

Tail-recursion by hand: list concatenation

Natural definition, runs in stack space $O(\texttt{length}(x))$:

```
let rec app x y =
  match x with
  | [] -> y
  | x1 :: xs -> x1 :: app xs y
```

Tail-recursive definition, using `rev_app` (reverse & append):

```
let rec rev_app x y =
  match x with
  | [] -> y
  | x1 :: xs -> rev_app xs (x1 :: y)


let app x y = rev_app (rev_app x []) y
```

Systematic way, 1: CPS transformation

We apply the CPS transformation locally to the app function:

```
let rec cps_app x y k =
  match x with
  | [] -> k y
  | x1 :: xs -> cps_app xs y (fun res -> k (x1 :: res))
```

then pass it the initial continuation:

```
let app x y = cps_app x y (fun res -> res)
```

The result runs in constant stack space but is less readable and less efficient than the `rev_app` implementation.

Systematic way, 2: defunctionalization

Let's defunctionalize `cps_app`.

```
type 'a funval =
  | A                      (* fun res -> res *)
  | B of 'a * 'a funval    (* fun res -> k (x1 :: res) *)

let rec defun_app x y k =
  match x with
  | [] -> apply k y
  | x1 :: xs -> defun_app xs y (B(x1, k))

and apply k res =
  match k with
  | A -> res
  | B(x1, k') -> apply k' (x1 :: res)

let app x y = defun_app x y A
```

Systematic way, 3: squinting

Note that the type `funval` of function "closures" is isomorphic to `list`.

```
let rec defun_app x y k =
  match x with
  | [] -> apply k y
  | x1 :: xs -> defun_app xs y (x1 :: k)

and apply k res =
  match k with
  | [] -> res
  | x1 :: k' -> apply k' (x1 :: res)

let app x y = defun_app x y []
```

Note that `apply` is really `rev_app`,
and `defun_app x y k` is really `rev_app (rev_app x k) y` . . .

Tail-recursive interpreters
(Ager, Biernacki, Danvy and Midtgaard, 2003)

What happens if we apply the "CPS + defunctionalization" trick to an interpreter for a functional language?

Let's see on an interpreter for call-by-name, using environments and de Bruijn indices.

A CBN interpreter with environments

```
type term = Var of int | Lam of term | App of term * term

type thunk = Thunk of term * env
 and env   = thunk list
 and value = Clos of term * env

let rec eval (e: env) (a: term) : value =
  match a with
  | Var n ->
      let (Thunk(a', e')) = List.nth e n in eval e' a'
  | Lam a ->
      Clos(a, e)
  | App(b, c) ->
      let (Clos(d, e')) = eval e b in eval (Thunk(c, e) :: e') d
```

After CPS conversion . . .

```
let rec cps_eval e a k =
  match a with
  | Var n ->
      let (Thunk(a', e')) = List.nth e n in cps_eval e' a' k
  | Lam a ->
      k (Clos(a, e))
  | App(b, c) ->
      cps_eval e b (fun Clos(d, e') ->
        cps_eval (Thunk(c, e) :: e') d k)

let eval e a = cps_eval e a (fun res -> res)
```

. . . then defunctionalization . . .

```
type funval =
  | A                (* fun res -> res *)
  | B of term * env * funval
                (* fun (Clos...) -> ... c ... e ... k *)

let rec defun_eval e a k =
  match a with
  | Var n -> let (Thunk(a', e')) = List.nth e n in defun_eval e' a' k
  | Lam a -> apply k (Clos(a, e))
  | App(b, c) -> defun_eval e b (B(c, e, k))

and apply k res =
  match k with
  | A -> res
  | B(c, e, k) ->
      let (Clos(d, e')) = res in defun_eval (Thunk(c, e) :: e') d k

let eval e a = defun_eval e a A
```

. . . and a bit of cleanup . . .

(`funval` isomorphic to `thunk list`; inlining of `apply`)

```
let rec tail_eval e a k =
  match a, k with
  | Var n, _ -> let (Thunk(a', e')) = List.nth e n in tail_eval e' a' k
  | Lam a, [] -> Clos(a, e)
  | Lam a, Thunk(c, e') :: k -> tail_eval (Thunk(c, e') :: e) a k
  | App(b, c), _ -> tail_eval e b (Thunk(c, e) :: k)

let eval e a = tail_eval e a []
```

. . . we obtain Krivine's machine!

As can be seen by extracting from `tail_eval` the underlying transition function over `(e, a, k)` triples.

```
type step_result = Finished of value | Next of env * term * thunk list

let step = function
  | (e, Var n, k) -> let (Thunk(a', e')) = List.nth e n in Next(e', a', k)
  | (e, Lam a, []) -> Finished(Clos(a, e))
  | (e, Lam a, Thunk(c, e') :: k) -> Next(Thunk(c, e') :: e, a, k)
  | (e, App(b, c), k) -> Next(e, b, Thunk(c, e) :: k)

let rec tail_eval e a k =
  match step (e, a, k) with
  | Finished v -> v | Next(e',a',k') -> tail_eval e' a' k'
```

(Compare with the `ACCESS`, `GRAB` and `PUSH` transitions of Krivine's machine.)

What about call-by-value?

If we apply the same trick to the CBV interpreter from lecture I, we do not obtain the Modern SECD, but something that is very close to another CBV abstract machine: the CEK machine of Felleisen and Friedman.

A Functional Correspondence between Evaluators and Abstract Machines.
Mads Sig Ager, Dariusz Biernacki, Olivier Danvy and Jan Midtgaard. PPDP 2003.

Outline

Propositions-as-types

(The Curry-Howard isomorphism)

Type expressions look a lot like logical formulas:

Types		Logical propositions
Function type $A \to B$	$\approx$	Implication $A \to B$
Product type $A \times B$	$\approx$	Conjunction $A \wedge B$
Sum type $A + B$	$\approx$	Disjunction $A \vee B$
Polymorphic type $\forall \alpha.A$	$\approx$	Universal quantification $\forall \alpha.A$

Where product types and sum types can be defined as

```
type 'a * 'b = Pair of 'a * 'b
type 'a + 'b = Left of 'a | Right of 'b
```

Proofs-as-programs, propositions-as-types

Likewise, constructive proofs of propositions look a lot like terminating programs of the corresponding type.

A proof of . . .		is . . .
$A \to B$	$\approx$	a total function from proofs of A to proofs of B.
$A \wedge B$	$\approx$	a pair of proofs, one for A and another for B.
$A \vee B$	$\approx$	a computation that decides which of A and B holds and returns either Left with a proof of A or Right with a proof of B.
$\forall x : A.\ B(x)$	$\approx$	a total function from values $v : A$ to proofs of $B(v)$.

(Much more details in course 2-7-1, "Foundations of proof systems".)

Curry-Howard isomorphism for classical logic

Is it possible to extend this correspondence to classical (not just constructive) logic?

For example: is there a program that "implements" the law of excluded middle?

$$\forall P. \ P \vee \neg P$$

Answer: yes, but we need `callcc` or similar control operators!

A Formulæ-as-Types Notion of Control, T. Griffin, POPL 1990.

"Implementing" excluded middle

Modulo Curry-Howard, the law of excluded middle

$$\forall P. \ P \vee \neg P \ \equiv \ \forall P. \ P \vee (P \to \texttt{False})$$

corresponds to the type

$$\forall P. \ P + (P \to \texttt{False})$$

where + is the sum type (`Left` and `Right` constructors)
and `False` an empty type.

The following term "implements" excluded middle:

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

"Implementing" excluded middle

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

How does it work?

- The rest of the proof term asks "P or not P?"
- The term above returns $\text{Right}(\lambda p \ldots)$, i.e. "$P \rightarrow \text{False}$".
- The only thing the rest of the proof can do with this proof is to apply it to a proof p of P to derive a contradiction.
- ... at which time the continuation is invoked, restarting the original proof with $\text{Left}(p)$, i.e. "P is true".