Program transformations

Outline

Outline

The fuss about purity

High level features (stores, exceptions, I/O, . . .) are essential:

- A program execution has a *raison d'être* only if it has I/O
- Processors have registers not functions
- Databases store persistent data
- Efficiency and clarity can be more easily achieved with exceptions

The fuss about purity

High level features (stores, exceptions, I/O, . . .) are essential:

- A program execution has a *raison d'être* only if it has I/O
- Processors have registers not functions
- Databases store persistent data
- Efficiency and clarity can be more easily achieved with exceptions

Question

Why some widespread languages, such as Haskell, insists on *purity*?

Advantages of a pure functional framework

- Much easier static analysis and correctness proofs
- Lazy evaluation
- Program optimizations

Static analysis: some examples

- Dependence analysis:
  - control dependencies: the evaluation of a program's expressions depends on the result of a previous expression (eg, if_then_else)
  - data dependencies: the result of a program's expressions depends on the result of a previous expression (eg, let-expression)

  Dependence analysis determines whether or not it is safe to reorder or parallelize the evaluation of expressions.

- Data-flow analysis:
  - reaching definitions: determines which definitions may reach a given point in the code (eg, registers allocation)
  - live variable analysis: calculate for each program point the variables that may be potentially read (eg, use for dead-code elimination)

  Data-flow analysis gathers information about the possible set of values calculated at various points.

- Type systems

Exercises

1. Try to imagine why the presence of impure expressions can make both dependence and data-flow analysis more difficult
2. Try to think about the problems of implementing a static type system to ensure that there won't be any uncaught exception.

Check also Appel's book

Lazy evaluation (1)

In lazy (as opposed to strict/eager) evaluation an expression passed as argument:

- is only evaluated if the result is required by the calling function (delayed evaluation)
- is only evaluated to the extent that is required by the calling function, called (short-circuit evaluation).
- is never evaluated more than once (as in applicative-order evaluation)

Lazy evaluation (1)

In lazy (as opposed to strict/eager) evaluation an expression passed as argument:

- is only evaluated if the result is required by the calling function (delayed evaluation)
- is only evaluated to the extent that is required by the calling function, called (short-circuit evaluation).
- is never evaluated more than once (as in applicative-order evaluation)

Example

$$(\lambda x.(\text{fst } x, \text{fst } x))((\lambda y.(3+y, e))5)$$

Lazy evaluation (1)

In lazy (as opposed to strict/eager) evaluation an expression passed as argument:

- is only evaluated if the result is required by the calling function (delayed evaluation)
- is only evaluated to the extent that is required by the calling function, called (short-circuit evaluation).
- is never evaluated more than once (as in applicative-order evaluation)

Example

$$
\begin{array}{rl}
& (\lambda x.(\mathrm{fst}\ x, \mathrm{fst}\ x))((\lambda y.(3+y,e))5) \\
\to & (\mathrm{fst}\ ((\lambda y.(3+y,e))5), \mathrm{fst}\ ((\lambda y.(3+y,e))5)))
\end{array}
$$

Lazy evaluation (1)

In lazy (as opposed to strict/eager) evaluation an expression passed as argument:

- is only evaluated if the result is required by the calling function (delayed evaluation)
- is only evaluated to the extent that is required by the calling function, called (short-circuit evaluation).
- is never evaluated more than once (as in applicative-order evaluation)

Example

$$(\lambda x.(\mathsf{fst}\ x, \mathsf{fst}\ x))((\lambda y.(3+y, e))5)$$

$$\begin{array}{ll} \to & (\mathsf{fst}\ ((\lambda y.(3+y, e))5), \mathsf{fst}\ ((\lambda y.(3+y, e))5))) \\ \to & (\mathsf{fst}\ (3+5, e), \mathsf{fst}\ (3+5, e)) \end{array}$$

Lazy evaluation (1)

In lazy (as opposed to strict/eager) evaluation an expression passed as argument:

- is only evaluated if the result is required by the calling function (delayed evaluation)
- is only evaluated to the extent that is required by the calling function, called (short-circuit evaluation).
- is never evaluated more than once (as in applicative-order evaluation)

Example

$$(\lambda x.(\mathrm{fst}\ x, \mathrm{fst}\ x))((\lambda y.(3+y,e))5)$$

$$\begin{array}{ll} \to & (\mathrm{fst}\ ((\lambda y.(3+y,e))5), \mathrm{fst}\ ((\lambda y.(3+y,e))5))) \\ \to & (\mathrm{fst}\ (3+5,e), \mathrm{fst}\ (3+5,e)) \\ \to & (3+5, 3+5) \end{array}$$

(The last reduction is an optimization: common subexpressions elimination)

Lazy evaluation (2)

In OCaml lazy evaluation can be implemented by *memoization*:

```
# let rec boucle = function 0 -> () | n -> boucle (n-1);;
val boucle : int -> unit = <fun>
# let gros_calcul () = boucle 100000000; 4;;
val gros_calcul : unit -> int = <fun>
# let v = gros_calcul ();;       (* it is slow *)
val v : int = 4
# v + 1;;                         (* it is fast *)
- : int = 5
#  let v () = gros_calcul ();; (* it is fast *)
val v : unit -> int = <fun>
# v () + 1;;                      (* it is slow *)
- : int = 5
# v () + 1;;                      (* it is slow *)
- : int = 5
# let v =
    let r = ref None in
      fun () -> match !r with
              | Some v -> v
              | None -> let v = (gros_calcul ()) in r := Some v; v;;
val v : unit -> int = <fun>
# v () + 1;;                       (* it is slow *)
- : int = 5
# v () + 1;;                       (* it is fast *)
- : int = 5
```

The `Lazy` module in OCaml

This is so frequent that OCaml provides this behavior natively via the special syntax `lazy` and the module `Lazy`:

```
# let v = lazy (gros_calcul ());;
val v : int lazy_t = <lazy>

# Lazy.force v;;                  (* it is slow *)
- : int = 4

# Lazy.force v;;                  (* it is fast *)
- : int = 4
```

Lazy evaluation (3)

Advantages

- Lazy data structures: possibly infinite, efficient copy, low memory footprint
- Better performance due to avoiding unnecessary calculations (?),
- Maintains purity (!)

Lazy evaluation (3)

Advantages

- Lazy data structures: possibly infinite, efficient copy, low memory footprint
- Better performance due to avoiding unnecessary calculations (?),
- Maintains purity (!)

Rationale

Since also strict languages can be endowed with laziness (see `Lazy` library in OCaml) then the clear advantage of *pervasive* lazy evaluation is to keep purity and, thus, referential transparency (not the other way round).

Optimizations

Purity makes important optimizations possible

1. Obvious program transformations. In Haskell

   ```
   map f (map g lst) = map (f.g) lst
   ```

   What if f and g had side effects?
   This is called “deforestation” and works for non-strict languages (in strict languages it may transform a function that does not terminates into one that terminates).
2. Function inlining, partial evaluation
3. Memoization
4. Common subexpressions elimination
5. Parallelization
6. Speculative evaluation
7. Other optimizations (see CPS part later on)

Program transformations

Previous optimizations are implemented by *program transformations*.

Program transformations

Previous optimizations are implemented by *program transformations*.

Meaning:

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

Program transformations

Previous optimizations are implemented by *program transformations*.

Meaning:

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

Usage:

Typically used as passes in a compiler. Progressively bridge the gap between high-level source languages and machine code.

Program transformations

Previous optimizations are implemented by *program transformations*.

Meaning:

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

Usage:

Typically used as passes in a compiler. Progressively bridge the gap between high-level source languages and machine code.

In this course:

We focus on translations between different languages. Translations within the same language are for optimization and studied in compiler courses.

Program transformations

Previous optimizations are implemented by *program transformations*.

Meaning:
In the broadest sense: all translations between programming languages that preserve the meaning of programs.
Usage:
Typically used as passes in a compiler. Progressively bridge the gap between high-level source languages and machine code.
In this course:
We focus on translations between different languages. Translations within the same language are for optimization and studied in compiler courses.
The interest is twofold:

1. Eliminate high-level features of a language and target a smaller or lower-level language.
2. To program in languages that lack a desired feature. E.g. use higher-order functions or objects in C; use imperative programming in Haskell or Coq.

Transformations

Considered transformations

We will show how to get rid of higher level features:

- High-order functions
- “Impure” features: exceptions, state, call/cc

Transformations

Considered transformations

We will show how to get rid of higher level features:

- High-order functions
- “Impure” features: exceptions, state, call/cc

Note

In order to simulate higher level features we first have to formally define their semantics.

Let us take a refresher course on operational semantics and reduction strategies

Outline

Syntax and small-step semantics

Syntax

Terms	a, b	$::=$	N	Numeric constant
		$\mid$	x	Variable
		$\mid$	$a\,b$	Application
		$\mid$	$\lambda x.a$	Abstraction
Values	v	$::=$	$\lambda x.a \mid N$	

Syntax and small-step semantics

Syntax

$$
\begin{array}{llcll}
\textit{Terms} & a, b & ::= & N & \text{Numeric constant} \\
 & & \mid & x & \text{Variable} \\
 & & \mid & a\,b & \text{Application} \\
 & & \mid & \lambda x.a & \text{Abstraction} \\
\textit{Values} & v & ::= & \lambda x.a \mid N &
\end{array}
$$

Small step semantics for strict functional languages

$$
\textit{Evaluation Contexts} \quad E \quad ::= \quad [\,] \mid E\,a \mid v\,E
$$

$$
\textsc{Beta}_v \quad (\lambda x.a)\,v \to a[x/v]
\qquad\qquad
\textsc{Context} \quad \frac{a \to b}{E[a] \to E[b]}
$$

Strategy and big-step semantics

Characteristics of the reduction strategy

Weak reduction: We cannot reduce under λ-abstractions;

Call-by-value: In an application $(\lambda x.a)\,b$, the argument b must be fully reduced to a value before β-reduction can take place.

Left-most reduction: In an application $a\,b$, we must reduce a to a value first before we can start reducing b.

Deterministic: For every term a, there is at most one b such that $a \to b$.

Strategy and big-step semantics

Characteristics of the reduction strategy

Weak reduction: We cannot reduce under λ-abstractions;

Call-by-value: In an application $(\lambda x.a)\, b$, the argument b must be fully reduced to a value before β-reduction can take place.

Left-most reduction: In an application $a\, b$, we must reduce a to a value first before we can start reducing b.

Deterministic: For every term a, there is at most one b such that $a \to b$.

Big step semantics for strict functional languages

$$N \Rightarrow N \qquad \lambda x.a \Rightarrow \lambda x.a \qquad \frac{a \Rightarrow \lambda x.c \quad b \Rightarrow v_\circ \quad c[x/v_\circ] \Rightarrow v}{a\, b \Rightarrow v}$$

Interpreter

The big step semantics induces an efficient implementation

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

exception Error

let rec subst x v = function          (* assumes v is closed *)
  | Const n -> Const n
  | Var y -> if x = y then v else Var y
  | Lam(y, b) -> if x = y then Lam(y, b) else Lam(y, subst x v b)
  | App(b, c) -> App(subst x v b, subst x v c)

let rec eval = function
  | Const n -> Const n
  | Var x -> raise Error
  | Lam(x, a) -> Lam(x, a)
  | App(a, b) ->
      match eval a with
      | Lam(x, c) -> let v = eval b in eval (subst x v c)
      | _ -> raise Error
```

Exercises

1. Define the small-step and big-step semantics for the call-by-name
2. Deduce from the latter the interpreter
3. Use the technique introduced for the type `'a delayed` earlier in the course to implement an interpreter with lazy evaluation.

Improving implementation

Environments

- Implementing textual substitution $a[x/v]$ is *inefficient*. This is why compilers and interpreters *do not* implement it.
- Alternative: record the binding $x \mapsto v$ in an *environment* e

$$\frac{e(x) = v}{e \vdash x \Rightarrow v} \qquad e \vdash N \Rightarrow N \qquad e \vdash \lambda x.a \Rightarrow \lambda x.a$$

$$\frac{e \vdash a \Rightarrow \lambda x.c \quad e \vdash b \Rightarrow v_\circ \quad e; x \mapsto v_\circ \vdash c \Rightarrow v}{e \vdash a\,b \Rightarrow v}$$

Improving implementation

Environments

- Implementing textual substitution $a[x/v]$ is *inefficient*. This is why compilers and interpreters *do not* implement it.
- Alternative: record the binding $x \mapsto v$ in an *environment* e

$$\frac{e(x) = v}{e \vdash x \Rightarrow v} \qquad e \vdash N \Rightarrow N \qquad e \vdash \lambda x.a \Rightarrow \lambda x.a$$

$$\frac{e \vdash a \Rightarrow \lambda x.c \quad e \vdash b \Rightarrow v_\circ \quad e; x \mapsto v_\circ \vdash c \Rightarrow v}{e \vdash a\,b \Rightarrow v}$$

Giving up substitutions in favor of environments does not come for free

Improving implementation

Environments

- Implementing textual substitution $a[x/v]$ is *inefficient*. This is why compilers and interpreters *do not* implement it.
- Alternative: record the binding $x \mapsto v$ in an *environment* e

$$\frac{e(x) = v}{e \vdash x \Rightarrow v} \qquad e \vdash N \Rightarrow N \qquad e \vdash \lambda x.a \Rightarrow \lambda x.a$$

$$\frac{e \vdash a \Rightarrow \lambda x.c \quad e \vdash b \Rightarrow v_\circ \quad e; x \mapsto v_\circ \vdash c \Rightarrow v}{e \vdash ab \Rightarrow v}$$

Giving up substitutions in favor of environments does not come for free

- **Lexical scoping** requires careful handling of environments

```
let x = 1 in
let f = λy.(x+1) in
let x = "foo" in
f 2
```

In the environment used to evaluate `f 2` the variable `x` is bound to 1.

Exercise

Try to evaluate

```
let x = 1 in
let f = λy.(x+1) in
let x = "foo" in
f 2
```

by the big-step semantics in the previous slide,
where `let` x `=` a `in` b is syntactic sugar for $(\lambda x.b)a$

let us outline it together

Function closures

To implement *lexical scoping in the presence of environments*, function abstractions $\lambda x.a$ must not evaluate to themselves, but to a function *closure*: a pair $(\lambda x.a)[e]$ (ie, the function and the *environment of its definition*)

Big step semantics with environments and closures

$$
\begin{array}{lrcl}
\textit{Values} & v & ::= & N \mid (\lambda x.a)[e] \\
\textit{Environments} & e & ::= & x_1 \mapsto v_1 ; \ldots ; x_n \mapsto v_n
\end{array}
$$

$$
\frac{e(x) = v}{e \vdash x \Rightarrow v} \qquad e \vdash N \Rightarrow N \qquad e \vdash \lambda x.a \Rightarrow (\lambda x.a)[e]
$$

$$
\frac{e \vdash a \Rightarrow (\lambda x.c)[e_\circ] \quad e \vdash b \Rightarrow v_\circ \quad e_\circ ; x \mapsto v_\circ \vdash c \Rightarrow v}{e \vdash a\,b \Rightarrow v}
$$

De Bruijn indexes

Identify variable not by names but by the number $\underline{n}$ of λ's that separate the variable from its binder in the syntax tree.

$$\lambda x.(\lambda y.y\,x)x \quad \text{is} \quad \lambda.(\lambda.\underline{0}\,\underline{1})\underline{0}$$

$\underline{n}$ is the variable bound by the n-th enclosing λ. Environments become sequences of values, the n-th value of the sequence being the value of variable $\underline{n-1}$.

$$\begin{array}{lrcl}
\textit{Terms} & a,b & ::= & N \mid \underline{n} \mid \lambda.a \mid a\,b \\
\textit{Values} & v & ::= & N \mid (\lambda.a)[e] \\
\textit{Environments} & e & ::= & v_0; v_1; \ldots; v_n
\end{array}$$

$$\frac{e = v_0;\ldots;v_n;\ldots;v_m}{e \vdash \underline{n} \Rightarrow v_n} \qquad e \vdash N \Rightarrow N \qquad e \vdash \lambda.a \Rightarrow (\lambda.a)[e]$$

$$\frac{e \vdash a \Rightarrow (\lambda.c)[e_\circ] \quad e \vdash b \Rightarrow v_\circ \quad {\color{red} v_\circ; e_\circ} \vdash c \Rightarrow v}{e \vdash a\,b \Rightarrow v}$$

The canonical, efficient interpreter

```
# type term = Const of int | Var of int | Lam of term | App of term * term
   and value = Vint of int | Vclos of term * environment
   and environment = value list                        (* use Vec instead *)

# exception Error

# let rec eval e a =
    match a with
     | Const n -> Vint n
     | Var n -> List.nth e n                        (* will fail for open terms *)
     | Lam a -> Vclos(Lam a, e)
     | App(a, b) ->
         match eval e a with
         | Vclos(Lam c, e') ->
             let v = eval e b in
             eval (v :: e') c
         | _ -> raise Error


# eval [] (App ( Lam (Var 0), Const (2)));;          (*  (λx.x)2 → 2  *)
- : value = Vint 2
```

Note: To obtain improved performance one should implement environments by persistent extensible arrays: for instance by the `Vec` library by Luca de Alfaro.

Outline

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

$$
\begin{array}{rcl}
[\![x]\!] & = & x \\
[\![\lambda x.a]\!] & = & \texttt{tuple}(\lambda(c,x).\texttt{let}\ \ x_1 = \texttt{field}_1(c)\ \ \texttt{in} \\
& & \qquad\qquad\qquad \vdots \\
& & \qquad\qquad\qquad \texttt{let}\ \ x_n = \texttt{field}_n(c)\ \ \texttt{in} \\
& & \qquad\qquad\qquad [\![a]\!], \\
& & \qquad\quad x_1, \ldots, x_n) \\
& & \text{where } x_1 , \ldots , x_n \text{ are the free variables of } \lambda x.a \\
[\![a\,b]\!] & = & \texttt{let}\ \ c = [\![a]\!]\ \ \texttt{in}\ \texttt{field}_0(c)(c, [\![b]\!])
\end{array}
$$

Definition of closure conversion

$$
\begin{array}{rcl}
[\![x]\!] & = & x \\
[\![\lambda x.a]\!] & = & \mathtt{tuple}(\lambda(c,x).\mathtt{let}\ \ x_1 = \mathtt{field}_1(c)\ \ \mathtt{in} \\
 & & \qquad\qquad\qquad\qquad \vdots \\
 & & \qquad\qquad\qquad\quad \mathtt{let}\ \ x_n = \mathtt{field}_n(c)\ \ \mathtt{in} \\
 & & \qquad\qquad\qquad\quad\ [\![a]\!], \\
 & & \qquad\quad x_1,\ldots,x_n) \\
 & & \text{where } x_1, \ldots, x_n \text{ are the free variables of } \lambda x.a \\
[\![a\,b]\!] & = & \mathtt{let}\ \ c = [\![a]\!]\ \ \mathtt{in}\ \ \mathtt{field}_0(c)(c,[\![b]\!])
\end{array}
$$

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

$$
\begin{array}{rcl}
[\![\mathtt{let}\ \ x = a\ \ \mathtt{in}\ \ b]\!] & = & \mathtt{let}\ \ x = [\![a]\!]\ \ \mathtt{in}\ \ [\![b]\!] \\
[\![a+b]\!] & = & [\![a]\!]+[\![b]\!]
\end{array}
$$

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

- : int -> int list -> int list = <fun>
```

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

- : int -> int list -> int list = <fun>
```

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(λ(c,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 , that is, the closure for itself. This closure can be found in the extra function parameter that closure conversion introduces.

$$
\begin{array}{rcl}
[\![\mu f.\lambda x.a]\!] & = & \texttt{tuple}(\lambda(f,x).\texttt{let}\ x_1 = \texttt{field}_1(f)\ \texttt{in} \\
 & & \qquad\qquad\qquad\vdots \\
 & & \qquad\qquad\quad \texttt{let}\ x_n = \texttt{field}_n(f)\ \texttt{in} \\
 & & \qquad\qquad\quad [\![a]\!], \\
 & & \qquad\quad x_1, \ldots, x_n)
\end{array}
$$

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

Notice that f is free in a and thus in $[\![a]\!]$, but bound in $[\![\mu f.\lambda x.a]\!]$.

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.

Closure conversion in object-oriented style

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

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

Closure conversion in object-oriented style

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

```
public class Cλx.a {
  protected internal Object x1, ..., xn;
  public Cλx.a(Object x1 , ... , Object xn ) {
    this.x1 = x1 ; ...; this.xn = xn ;
  }
  public Object apply(Object x) {
    return ⟦a⟧ ;
  }
}
```

Closure conversion in object-oriented style

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

```
public class Cλx.a {
  protected internal Object x1, ..., xn;
  public Cλx.a(Object x1 , ... , Object xn ) {
    this.x1 = x1 ; ...; this.xn = xn ;
  }
  public Object apply(Object x) {
    return ⟦a⟧ ;
  }
}
```

Typing

In order to have a more precise typing the static types of the variables and of the function should be used instead of `Object`. In particular the method `apply` should be given the same input and return types as the encoded function.

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:

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

Where an object is interpreted as a record whose fields are methods which are parametrized by `self`.

First class closure

Modern OOL such as Scala and C# (and announced for Java in the JDK 7 but currently deferred to JDK 8 or later) provide syntax to define closures, without the need to encode them.

First class closure

Modern OOL such as Scala and C# (and announced for Java in the JDK 7 but currently deferred to JDK 8 or later) provide syntax to define closures, without the need to encode them.

For instance C# provides a `delegate` modifier to define closures:

```
public delegate int DComparer (Object x, Object y)
```

Defines a new distinguished type `DComparer` whose instances are functions from two objects to `int` (i.e., `DComparer` $\equiv$ `(Object*Object)`$\rightarrow$`int`)

First class closure

Modern OOL such as Scala and C# (and announced for Java in the JDK 7 but currently deferred to JDK 8 or later) provide syntax to define closures, without the need to encode them.

For instance C# provides a `delegate` modifier to define closures:

```
public delegate int DComparer (Object x, Object y)
```

Defines a new distinguished type `DComparer` whose instances are functions from two objects to `int` (i.e., `DComparer` $\equiv$ `(Object*Object)`$\rightarrow$`int`)

Instances are created by passing to `new` a static or instance method (with compatible types):

```
DComparer mycomp = new DComparer(String.Comparer)
```

The closure `mycomp` can be passed around (wherever an argument of type `DComparer` is expected), or applied as in `mycomp("Scala","Java")`

First class closure

Actually in C# it is possible to define "lambda expressions":

Here how to write $(\lambda(x,y).x+y)$ in C#:

```
(x,y) => x + y
```

First class closure

Actually in C# it is possible to define “lambda expressions”:

Here how to write $(\lambda(x,y).x+y)$ in C#:

```
(x,y) => x + y
```

Lambda expressions can be used to instantiate closures:

```
DComparer myComp = (x,y) => x + y
```

First class closure

Actually in C# it is possible to define “lambda expressions”:

Here how to write $(\lambda(x,y).x+y)$ in C#:

```
(x,y) => x + y
```

Lambda expressions can be used to instantiate closures:

```
DComparer myComp = (x,y) => x + y
```

Delegates (roughly, function types) can be polymorphic:

```
public delegate TResult Func<TArg0, TResult>(TArg0 arg0)
```

The delegate can be instantiated as `Func<int,bool> myFunc` where `int` is an input parameter and `bool` is the return value. The return value is always specified in the last type parameter. `Func<int, string, bool>` defines a delegate with two input parameters, `int` and `string`, and a return type of `bool`.

```
Func<int, bool> myFunc = x => x == 5;
bool result = myFunc(4);          // returns false of course
```

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 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 $\mathtt{C}(\mathtt{v}_1,\ldots,\mathtt{v}_n)$ where the constructor `C` uniquely identifies the function, and the constructor arguments $\mathtt{v}_1$,..., $\mathtt{v}_n$ are the values of the variables $x_1, ..., x_n$ free in the body of the function.

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

Definition of defunctionalization

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

The difference between recursive and non-recursive functions is made in the definition of `apply`

(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.λy.b}(x1,...,xm) -> let f = fun in let y = arg in ⟦ b ⟧
    | ...
in ⟦program⟧
```

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.λy.b}(x1,...,xm) -> let f = fun in let y = arg in ⟦ b ⟧
    | ...
in ⟦program⟧
```

Note

Unlike closure conversion, this is a whole-program transformation.

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 $\mathtt{C}_{\lambda x.\lambda y.x}$ and C2 for $\mathtt{C}_{\lambda y.x}$.

Outline

Small step semantics for exceptions

$$
\begin{array}{rcll}
(\texttt{try}\ \ v\ \ \texttt{with}\ \ x \to b) & \to & v & \\
(\texttt{try}\ \ \texttt{raise}\ \ v\ \ \texttt{with}\ \ x \to b) & \to & b[x/v] & \\
P[\texttt{raise}\ \ v] & \to & \texttt{raise}\ \ v & \qquad \textit{if } P \neq [\,]
\end{array}
$$

$$
\frac{a \to b}{E[a] \to E[b]}
$$

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

Reduction contexts:

$$
E ::= [\,] \mid E\,a \mid v\,E \mid \texttt{raise}\ \ E \mid \texttt{try}\ \ E\ \ \texttt{with}\ \ x \to a \mid \ldots
$$

Exception propagation contexts: (no `try_with`)

$$
P ::= [\,] \mid P\,a \mid v\,P \mid \texttt{raise}\ \ P \mid \ldots
$$

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

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

where P does not contain any `try...with` constructs (that encloses the hole). $\mathrm{P}[\texttt{raise}\ \ v]$ head-reduces to raise v , and $E'[\texttt{try}\ \ [\,]\ \ \texttt{with}\ \ x \to b]$ is an evaluation context. The reduction sequence is therefore:

$$\begin{array}{rcl} p = E'[\texttt{try}\ \ P[\texttt{raise}\ \ v]\ \ \texttt{with}\ \ x \to b] & \to & E'[\texttt{try}\ \texttt{raise}\ \ v\ \ \texttt{with}\ \ x \to b] \\ & \to & E'[b[x/v]] \end{array}$$

If there are no `try` ... `with` around the raise, E is a *propagation context* and the reduction is therefore

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

Reduction semantics for exceptions

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

- Termination: $a \rightarrow^* v$
- Uncaught exception: $a \rightarrow \texttt{raise}\ \ v$
- Divergence: $a \rightarrow^* a' \rightarrow \ldots$
- Error: $a \rightarrow a' \not\rightarrow$ where $a \neq v$ and $a \neq \texttt{raise}\ \ v$.

Big step semantics for exception

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

$$\frac{a \Rightarrow v}{\texttt{try}\ \ a\ \ \texttt{with}\ \ x \rightarrow b \Rightarrow v} \qquad \frac{a \Rightarrow \texttt{raise}\ \ v \qquad b[x/v] \Rightarrow r}{\texttt{try}\ \ a\ \ \texttt{with}\ \ x \rightarrow b \Rightarrow r}$$

as well as exception propagation rules such as:

$$\frac{a \Rightarrow \texttt{raise}\ \ v}{a\,b \Rightarrow \texttt{raise}\ \ v} \qquad \frac{a \Rightarrow v' \qquad b \Rightarrow \texttt{raise}\ \ v}{ab \Rightarrow \texttt{raise}\ \ v}$$

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 $\mathit{Val}(v)$ if a evaluates normally, or to $\mathit{Exn}(v)$ if a terminates early by raising exception v. $\mathit{Val}, \mathit{Exn}$ are datatype constructors.

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

Definition of the transformation

$$
\begin{array}{rcl}
[\![\texttt{raise}\ a]\!] & = & \texttt{match}\ [\![a]\!]\ \texttt{with} \\
& & \mid \mathit{Exn}(x) \rightarrow \mathit{Exn}(x) \\
& & \mid \mathit{Val}(x) \rightarrow \mathit{Exn}(x) \\
[\![\texttt{try}\ a\ \texttt{with}\ x \rightarrow b]\!] & = & \texttt{match}\ [\![a]\!]\ \texttt{with} \\
& & \mid \mathit{Exn}(x) \rightarrow [\![b]\!] \\
& & \mid \mathit{Val}(x) \rightarrow \mathit{Val}(x)
\end{array}
$$

Definition of the transformation

$$
\begin{array}{rcl}
[\![N]\!] & = & Val(N) \\
[\![x]\!] & = & Val(x) \\
[\![\lambda x.a]\!] & = & Val(\lambda x.[\![a]\!]) \\
[\![\texttt{let } x = a \texttt{ in } b]\!] & = & \texttt{match } [\![a]\!] \texttt{ with } Exn(x) \to Exn(x) \mid Val(x) \to [\![b]\!] \\
[\![a\,b]\!] & = & \texttt{match } [\![a]\!] \texttt{ with} \\
& & \mid Exn(x) \to Exn(x) \\
& & \mid Val(x) \to \texttt{match } [\![b]\!] \texttt{ with} \\
& & \qquad\qquad \mid Exn(y) \to Exn(y) \\
& & \qquad\qquad \mid Val(y) \to x\,y
\end{array}
$$

Effect on types: if $a : \tau$ then $[\![a]\!] : [\![\tau]\!]$ where $[\![\tau_1 \to \tau_2]\!] = (\tau_1 \to [\![\tau_2]\!])$ `outcome` and $[\![\tau]\!] = \tau$ `outcome` otherwise and where `type 'a outcome = Val of 'a | Exn of exn`.

Example of conversion

Let fun and arg be two variables, then:

```
⟦ try fun arg with w -> 0 ⟧ =
     match
        match Val(fun) with
        | Exn(x) -> Exn(x)
        | Val(x) ->
            match Val(arg) with
            | Exn(y) -> Exn(y)
            | Val(y) -> x y
     with
     | Val(z) -> Val(z)
     | Exn(w) -> Val(0)
```

Notice that the two inner `match` can be simplified yielding

```
⟦ try fun arg with w -> 0 ⟧ =
     match fun arg with
     | Val(z) -> Val(z)
     | Exn(w) -> Val(0)
```

This transformation can be generalized by defining *administrative reductions*.

Administrative reductions

The naive conversion generates many useless match constructs over arguments whose shape *Val*(...) or *Exn*(...) is known at compile-time.

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

Administrative reduction

$$\begin{array}{rcl}
(\texttt{match}\ \ \mathit{Exn}(v)\ \ \texttt{with}\ \ \mathit{Exn}(x) \rightarrow b \mid \mathit{Val}(x) \rightarrow c) & \xrightarrow{\text{adm}} & b[x/v] \\
(\texttt{match}\ \ \mathit{Val}(v)\ \ \texttt{with}\ \ \mathit{Exn}(x) \rightarrow b \mid \mathit{Val}(x) \rightarrow c) & \xrightarrow{\text{adm}} & c[x/v]
\end{array}$$

Administrative reductions

The naive conversion generates many useless match constructs over arguments whose shape $Val(\ldots)$ or $Exn(\ldots)$ is known at compile-time.

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

Administrative reduction

$$(\texttt{match}\ \ Exn(v)\ \ \texttt{with}\ \ Exn(x) \rightarrow b \mid Val(x) \rightarrow c) \quad \xrightarrow{\text{adm}} \quad b[x/v]$$
$$(\texttt{match}\ \ Val(v)\ \ \texttt{with}\ \ Exn(x) \rightarrow b \mid Val(x) \rightarrow c) \quad \xrightarrow{\text{adm}} \quad c[x/v]$$

Correctness of the conversion

Define the conversion of a value $\mathcal{V}(v)$ as $\mathcal{V}(N) = N$ and $\mathcal{V}(\lambda x.a) = \lambda x.[\![a]\!]$.

Theorem

1. If $a \Rightarrow v$, then $[\![a]\!] \Rightarrow Val(\mathcal{V}(v))$.

2. If $a \Rightarrow \texttt{raise}\ \ v$, then $[\![a]\!] \Rightarrow Exn(\mathcal{V}(v))$.

3. If $a \Uparrow$, then $[\![a]\!] \Uparrow$.

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

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.

Semantics of references

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

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

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

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

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 $a \triangleleft s \to a' \triangleleft s'$
(read: in initial store s, a reduces to a' and updates the store to s')

$$\begin{array}{rcll} (\lambda x.a)v \triangleleft s & \to & a[x/v] \triangleleft s & \\ \mathit{ref}\ v \triangleleft s & \to & \ell \triangleleft (s + \ell \mapsto v) & \text{where } \ell \notin \mathit{Dom}(s) \\ \mathit{deref}\ \ell \triangleleft s & \to & s(\ell) \triangleleft s & \\ \mathit{assign}\ \ell\ v \triangleleft s & \to & (\,) \triangleleft (s + \ell \mapsto v) & \end{array}$$

$$\textsc{Context}\quad \frac{a \triangleleft s \to a' \triangleleft s'}{E(a) \triangleleft s \to E(a') \triangleleft s'}$$

Reduction semantics for references

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

$$
\begin{array}{rcll}
(\lambda x.a)v \triangleleft s & \to & a[x/v] \triangleleft s & \\
\mathit{ref}\ v \triangleleft s & \to & \ell \triangleleft (s + \ell \mapsto v) & \quad \text{where } \ell \notin \mathit{Dom}(s) \\
\mathit{deref}\ \ell \triangleleft s & \to & s(\ell) \triangleleft s & \\
\mathit{assign}\ \ell\ v \triangleleft s & \to & (\,) \triangleleft (s + \ell \mapsto v) &
\end{array}
$$

$$
\textsc{Context}\ \frac{a \triangleleft s \to a' \triangleleft s'}{E(a) \triangleleft s \to E(a') \triangleleft s'}
$$

Notice that we also added a new value, $(\,)$, the result of a side-effect.

Example of reduction sequence

Let us reduce the following term

$$\texttt{let } r = \texttt{ref } 3 \texttt{ in } r := !r + 1; !r$$

that is

$$\texttt{let } r = \texttt{ref } 3 \texttt{ in let } x = r := !r + 1 \texttt{ in } !r$$

(recall that `e1; e2` is syntactic sugar for `let _ = e1 in e2`)

In red: the active redex at every step.

$$\begin{array}{l}
\texttt{let } r = \texttt{ref } 3 \texttt{ in let } x = r := !r + 1 \texttt{ in } !r \lhd \varnothing \\
\quad \to \texttt{let } r = \ell \texttt{ in let } x = r := !r + 1 \texttt{ in } !r \lhd \ell \mapsto 3 \\
\quad \to \texttt{let } x = \ell := !\ell + 1 \texttt{ in } !\ell \lhd \ell \mapsto 3 \\
\quad \to \texttt{let } x = \ell := 3 + 1 \texttt{ in } !\ell \lhd \ell \mapsto 3 \\
\quad \to \texttt{let } x = \ell := 4 \texttt{ in } !\ell \lhd \ell \mapsto 3 \\
\quad \to \texttt{let } x = () \texttt{ in } !\ell \lhd \ell \mapsto 4 \\
\quad \to !\ell \lhd \ell \mapsto 4 \\
\quad \to 4
\end{array}$$

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 about imperative code in theorem provers.

Definition of the conversion

Core constructs

$$
\begin{array}{rcl}
[\![N]\!] & = & \lambda s.(N,s) \\
[\![x]\!] & = & \lambda s.(x,s) \\
[\![\lambda x.a]\!] & = & \lambda s.(\lambda x.[\![a]\!],s) \\
[\![\texttt{let}\ \ x=a\ \ \texttt{in}\ \ b]\!] & = & \lambda s.\texttt{match}\ \ [\![a]\!]s\ \ \texttt{with}\ \ (x,s') \to [\![b]\!]s' \\
[\![a\,b]\!] & = & \lambda s.\texttt{match}\ \ [\![a]\!]s\ \ \texttt{with}\ \ (x_a,s') \to \\
 & & \quad \texttt{match}\ \ [\![b]\!]s'\ \ \texttt{with}\ \ (x_b,s'') \to x_a\,x_b\,s''
\end{array}
$$

Definition of the conversion

Constructs specific to references

$$
\begin{aligned}
[\![\texttt{ref}\ a]\!] &= \lambda s.\texttt{match}\ [\![a]\!]\,s\ \texttt{with}\ (x,s') \to \texttt{store\_alloc}\,x\,s' \\
[\![!a]\!] &= \lambda s.\texttt{match}\ [\![a]\!]\,s\ \texttt{with}\ (x,s') \to (\texttt{store\_read}\,x\,s',s') \\
[\![a := b]\!] &= \lambda s.\texttt{match}\ [\![a]\!]\,s\ \texttt{with}\ (x_a,s') \to \\
&\qquad\quad \texttt{match}\ [\![b]\!]\,s'\ \texttt{with}\ (x_b,s'') \to (\varepsilon,\texttt{store\_write}\,x_a\,x_b\,s'')
\end{aligned}
$$

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.

Here ε represents the `()` value.

For instance we can use `Vec`, a library of extensible functional arrays by Luca de Alfaro. In that case we have that locations are natural numbers, a store is a vector `s` created by `Vec.empty`, a fresh location for the store s is returned by `Vec.length` s. Precisely, we have

$$\begin{aligned}
\texttt{store_alloc}\ v\, s &= (\,\texttt{Vec.length}\ s\,,\ \texttt{Vec.append}\ v\, s\,)\\
\texttt{store_read}\ \ell\, s &= \texttt{Vec.get}\ \ell\, s\\
\texttt{store_write}\ \ell\, v\, s &= \texttt{Vec.set}\ \ell\, v\, s
\end{aligned}$$

Typing (assuming all values stored in references are of the same type `sval`):

$\texttt{store\_alloc}\ : \texttt{sval} \to \texttt{store} \to \texttt{location} \times \texttt{store}$

$\texttt{store\_read}\ : \texttt{location} \to \texttt{store} \to \texttt{sval}$

$\texttt{store\_write}\ : \texttt{location} \to \texttt{sval} \to \texttt{store} \to \texttt{store}$

where `location` is `int` and `store` is `Vec.t`.

Example of conversion

Administrative reductions: (where x, y, s, and s' are variables)

$$
\begin{array}{rcl}
(\texttt{match}\ (a,s)\ \texttt{with}\ (x,s') \to b) & \xrightarrow{\text{adm}} & \texttt{let}\ x = a\ \texttt{in}\ b[s'/s] \\
(\lambda s.b)s' & \xrightarrow{\text{adm}} & b[s'/s] \\
\texttt{let}\ x = v\ \texttt{in}\ b & \xrightarrow{\text{adm}} & b[x/v] \\
\texttt{let}\ x = y\ \texttt{in}\ b & \xrightarrow{\text{adm}} & b[x/y]
\end{array}
$$

(the first reduction replaces only the store since replacing also a for x may change de evaluation order: a must be evaluated before the evaluation of b)

Example of conversion

Administrative reductions: (where x, y, s, and s' are variables)

$$\begin{aligned}
(\mathtt{match}\ (a,s)\ \mathtt{with}\ (x,s') \to b) &\xrightarrow{\text{adm}} \mathtt{let}\ x = a\ \mathtt{in}\ b[s'/s] \\
(\lambda s.b)s' &\xrightarrow{\text{adm}} b[s'/s] \\
\mathtt{let}\ x = v\ \mathtt{in}\ b &\xrightarrow{\text{adm}} b[x/v] \\
\mathtt{let}\ x = y\ \mathtt{in}\ b &\xrightarrow{\text{adm}} b[x/y]
\end{aligned}$$

(the first reduction replaces only the store since replacing also a for x may change de evaluation order: a must be evaluated before the evaluation of b)

Example of translation after administrative reductions:

Consider again the term

$$\mathtt{let}\ r = \mathtt{ref}\ 3\ \mathtt{in}\ r := !r + 1; !r$$

We have

```
⟦ let r = ref 3 in let x = r := !r + 1 in !r ⟧  =
    λs. match  store_alloc 3 s 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)
```

Outline

Notion of continuation

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

Notion of continuation

Given a program *p* and a subexpression *a* of *p*, the *continuation* of *a* is the computation that remains 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

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.)
The continuation of the whole program *p* is of course $\lambda x.x$

Continuations and reduction contexts

Continuations closely correspond with reduction contexts in small-step operational semantics:

Nota Bene

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

Continuations and reduction contexts

Continuations closely correspond with reduction contexts in small-step operational semantics:

Nota Bene

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

Example

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

$$\begin{array}{rcl} (1+2)*(3+4) & = & E_1[1+2] \text{ with } E_1 = [\,]*(3+4) \\ \to 3*(3+4) & = & E_2[3+4] \text{ with } E_2 = 3*[\,] \\ \to \quad 3*7 & = & E_3[3*7] \text{ with } E_3 = [\,] \\ \to \quad 21 & & \end{array}$$

The continuation of $1+2$ is $\lambda x.E_1[x] = \lambda x.x*(3+4)$.
The continuation of $3+4$ is $\lambda x.E_2[x] = \lambda x.3*x$.
The continuation of $3*7$ is $\lambda x.E_3[x] = \lambda x.x$.

What continuations are for?

Historically continuations where introduced to define a denotational semantics for the `goto` statement in imperative programming

- Imagine we have a pure imperative programming language.
- As suggested by the state passing translation a program $\mathtt{p}$ of this language can be interpreted as a function that transforms states into states:

$$[\![\mathtt{p}]\!] : \mathcal{S} \to \mathcal{S}$$

- This works as long as we do not have `GOTO`.

- Consider the following spaghetti code in BASIC

```
10 i = 0
20 i = i + 1
30 PRINT i; " squared = "; i * i
40 IF i >= 10 THEN GOTO 60
50 GOTO 20
60 PRINT "Program Completed."
70 END
```

- Consider the following spaghetti code in BASIC

```
10 i = 0
20 i = i + 1
30 PRINT i; " squared = "; i * i
40 IF i >= 10 THEN GOTO 60
50 GOTO 20
60 PRINT "Program Completed."
70 END
```

- Idea: add to the interpretation of programs a further parameter: a continuation.

- Consider the following spaghetti code in BASIC

```
10 i = 0
20 i = i + 1
30 PRINT i; " squared = "; i * i
40 IF i >= 10 THEN GOTO 60
50 GOTO 20
60 PRINT "Program Completed."
70 END
```

- Idea: add to the interpretation of programs a further parameter: a continuation.
- In this framework a continuation is a function of type $\mathcal{S} \to \mathcal{S}$ since it takes the result of a statement (i.e. a state) and returns a new result (new state).

$$[\![\mathrm{p}]\!] : \mathcal{S} \to (\mathcal{S} \to \mathcal{S}) \to \mathcal{S}$$

- Consider the following spaghetti code in BASIC

```
10 i = 0
20 i = i + 1
30 PRINT i; " squared = "; i * i
40 IF i >= 10 THEN GOTO 60
50 GOTO 20
60 PRINT "Program Completed."
70 END
```

- Idea: add to the interpretation of programs a further parameter: a continuation.
- In this framework a continuation is a function of type $\mathcal{S} \to \mathcal{S}$ since it takes the result of a statement (i.e. a state) and returns a new result (new state).

$$[\![\mathrm{p}]\!] : \mathcal{S} \to (\mathcal{S} \to \mathcal{S}) \to \mathcal{S}$$

- Every (interpretation of a) statement will do their usual modifications on the state they received and then will pass the resulting state to the continuations they received

- Consider the following spaghetti code in BASIC

```
10 i = 0
20 i = i + 1
30 PRINT i; " squared = "; i * i
40 IF i >= 10 THEN GOTO 60
50 GOTO 20
60 PRINT "Program Completed."
70 END
```

- Idea: add to the interpretation of programs a further parameter: a continuation.
- In this framework a continuation is a function of type $\mathcal{S} \to \mathcal{S}$ since it takes the result of a statement (i.e. a state) and returns a new result (new state).

$$[\![\mathrm{p}]\!] : \mathcal{S} \to (\mathcal{S} \to \mathcal{S}) \to \mathcal{S}$$

- Every (interpretation of a) statement will do their usual modifications on the state they received and then will pass the resulting state to the continuations they received
- Only the `GOTO` behaves differently: it throws away the received continuation and use instead the continuation of the statement to go to.

- Consider the following spaghetti code in BASIC

```
10 i = 0
20 i = i + 1
30 PRINT i; " squared = "; i * i
40 IF i >= 10 THEN GOTO 60
50 GOTO 20
60 PRINT "Program Completed."
70 END
```

- Idea: add to the interpretation of programs a further parameter: a continuation.
- In this framework a continuation is a function of type $\mathcal{S} \to \mathcal{S}$ since it takes the result of a statement (i.e. a state) and returns a new result (new state).

$$[\![\mathrm{p}]\!] : \mathcal{S} \to (\mathcal{S} \to \mathcal{S}) \to \mathcal{S}$$

- Every (interpretation of a) statement will do their usual modifications on the state they received and then will pass the resulting state to the continuations they received
- Only the `GOTO` behaves differently: it throws away the received continuation and use instead the continuation of the statement to go to.
- For instance the statement in line 50 will receive a state and a continuation and will pass the received state to the continuation of the instruction 20.

Continuations for compiler optimizations

Explicit continuations are inserted by some compiler for optimization:

```
(*       defines the product of all prime numbers <= n      *)

let rec prodprime n =                          (* bear with this       *)
    if n = 1                                   (* horrible indentation *)
      then
         1
      else if
         isprime n                             (* receives k returns b *)
      then n * prodprime (n-1)                 (* receives j returns p *)
      else prodprime (n-1);;                   (* receives h returns q *)
```

The compiler adds (at function calls) points to control the flow of this function

Continuations for compiler optimizations

Explicit continuations are inserted by some compiler for optimization:

```
(*       defines the product of all prime numbers <= n       *)

let rec prodprime n =                       (* bear with this        *)
    if n = 1                                (* horrible indentation *)
      then
         1
      else if
         isprime n                          (* receives k returns b *)
      then n * prodprime (n-1)              (* receives j returns p *)
      else prodprime (n-1);;                (* receives h returns q *)
```

The compiler adds (at function calls) points to control the flow of this function

1. `isprime` is given a return address `k` and returns a boolean `b`

Continuations for compiler optimizations

Explicit continuations are inserted by some compiler for optimization:

```
(*       defines the product of all prime numbers <= n      *)

let rec prodprime n =                   (* bear with this       *)
    if n = 1                            (* horrible indentation *)
      then
         1
      else if
         isprime n                      (* receives k returns b *)
      then n * prodprime (n-1)          (* receives j returns p *)
      else prodprime (n-1);;            (* receives h returns q *)
```

The compiler adds (at function calls) points to control the flow of this function

1. `isprime` is given a return address `k` and returns a boolean `b`
2. The first `prodprime` call will return at point `j` an integer `p`

Continuations for compiler optimizations

Explicit continuations are inserted by some compiler for optimization:

```
(*      defines the product of all prime numbers <= n      *)

let rec prodprime n =                   (* bear with this      *)
    if n = 1                            (* horrible indentation *)
      then
         1
      else if
         isprime n                      (* receives k returns b *)
      then n * prodprime (n-1)          (* receives j returns p *)
      else prodprime (n-1);;            (* receives h returns q *)
```

The compiler adds (at function calls) points to control the flow of this function

1. `isprime` is given a return address `k` and returns a boolean `b`
2. The first `prodprime` call will return at point `j` an integer `p`
3. The second `prodprime` call will return at point `h` an integer `q`

Continuations for compiler optimizations

```
let rec prodprime(n,c) =
  if n = 1
  then
    c 1                        (* pass 1 to the current continuation c *)
  else
    let k b =                              (* continuation of isprime *)
      if b
      then
        let j p =                        (* continuation of prodprime *)
          let a = n * p in c a in
          let m = n - 1
        in prodprime(m,j)  (*call prodprime(n-1) with its continuation*)
      else
        let h q =                        (* continuation of prodprime *)
          c q in
          let i = n - 1
        in prodprime(i,h)  (*call prodprime(n-1) with its continuation*)
    in isprime(n,k)        (* call isprime(n) with its continuation k *)
```

Notice that we added variables `m` and `i` to store intermediate results

Advantages

Explicit continuations bring several advantages:

- Tail recursion: `prodprime` is now tail recursive. Also the call that was already call recursive has trivial continuation (`h` is equivalent to `c`) that can be simplified:

```
let h q =
   c q in
   let i = n - 1
in prodprime(i,h)
```

$\Rightarrow$

```
let i = n - 1
in prodprime(i,c)
```

Advantages

Explicit continuations bring several advantages:

- Tail recursion: `prodprime` is now tail recursive. Also the call that was already call recursive has trivial continuation (`h` is equivalent to `c`) that can be simplified:

```
let h q =
   c q in
   let i = n - 1
in prodprime(i,h)
```

$\Rightarrow$

```
let i = n - 1
in prodprime(i,c)
```

- Inlining: In languages that are strict and/or have side effects inlining is very difficult to do directly. Explicit continuations overcome all the problems since *all actual parameters to functions are either variables or constants* (never a non-trivial sub-expression)

Advantages

Explicit continuations bring several advantages:

- Tail recursion: `prodprime` is now tail recursive. Also the call that was already call recursive has trivial continuation (`h` is equivalent to `c`) that can be simplified:

```
let h q =
   c q in
   let i = n - 1
in prodprime(i,h)
```

$\Rightarrow$

```
let i = n - 1
in prodprime(i,c)
```

- Inlining: In languages that are strict and/or have side effects inlining is very difficult to do directly. Explicit continuations overcome all the problems since *all actual parameters to functions are either variables or constants* (never a non-trivial sub-expression)
- Dataflow analysis describes static propagation of values. Continuation make this flow explicit and easy this analysis (for detection of dead-code or register allocation).

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.

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 $\texttt{callcc}(\lambda k.a)$ 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 $\texttt{callcc}(\lambda k.a)$.
- If, during the evaluation of *a or later* (if we stored k somewhere or we passed it along), we evaluate $\texttt{throw}\ k\ v$, evaluation continues as if $\texttt{callcc}(\lambda k.a)$ 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` (of type `'a -> bool`).

```
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`, then 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`, the function `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`. This is used as shown in the next function.

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 (seen)
- Backtracking (seen)
- Generators for imperative iterators such as Python's and C# `yield` (next slides).
- Coroutines / cooperative multithreading (few slides ahead).
- Checkpoint/replay debugging (in order to save the intermediate state —*ie*, a checkpoint— of a process you can save the continuation).

Python's `yield`

`yield` inside a function makes the function a *generator* that when called returns an object of type *generator*. The object has a method `next` that executes the function till the expression `yield`, returns the value of the `yield`, and at the next call of `next`, starts again right after the `yield`.

```
>>> def gen_fibonacci():                    # Generator of Fibonacci suite
...     a, b = 1, 2
...     while True:
...         yield a
...         a, b = b, a + b
...
>>> fib = gen_fibonacci()
>>> for i in range(4):
...     print fib.next()
...
1
2
3
5
>>> fib.next()
8
>>> fib.next()
13
```

Python's `yield`

Actually the argument of a `for` loop is a generator object.
At each loop the for calls the `next` method of the generator. When the generator do not find a next yield and exits, then it raises a exception that makes the for exit.

```
>>> for i in fib:
...     print i
...
21
34
55
89
144
233
377
610
987
 ⋮
 ⋮
```

Simulate `yield` by `callcc`

```
let fib () =
   let a,b = ref 1, ref 2 in
      while true do
         yield !a;
         b := !a + !b;                            (* note:  a,b ← b,a+b *)
         a := !b - !a;
      done
```

Simulate `yield` by `callcc`

```
let return = ref (Obj.magic None);;
let resume = ref (Obj.magic None);;
let fib () =
  let a,b = ref 1, ref 2 in
    while true do
      yield !a;
      b := !a + !b;
      a := !b - !a;
    done
```

1. Use two references to store addresses to resume `fib` and return from it;

Simulate `yield` by `callcc`

```
let return = ref (Obj.magic None);;
let resume = ref (Obj.magic None);;
let fib () = callcc (fun kk -> return := kk;
  let a,b = ref 1, ref 2 in
    while true do
      yield !a;
      b := !a + !b;
      a := !b - !a;
    done
  )
```

1. Use two references to store addresses to resume `fib` and return from it;
2. Save the return point in `return`

Simulate `yield` by `callcc`

```
let return = ref (Obj.magic None);;
let resume = ref (Obj.magic None);;
let fib () = callcc (fun kk -> return := kk;
   let a,b = ref 1, ref 2 in
      while true do
         callcc (fun cc -> resume := cc                    );
         b := !a + !b;
         a := !b - !a;
      done
   )
```

1. Use two references to store addresses to resume `fib` and return from it;
2. Save the return point in `return`
3. Save the resumption point in `resume`

Simulate `yield` by `callcc`

```
let return = ref (Obj.magic None);;
let resume = ref (Obj.magic None);;
let fib () = callcc (fun kk -> return := kk;
   let a,b = ref 1, ref 2 in
      while true do
         callcc (fun cc -> resume := cc; throw !return !a);
         b := !a + !b;
         a := !b - !a;
      done
   )
```

1. Use two references to store addresses to resume `fib` and return from it;
2. Save the return point in `return`
3. Save the resumption point in `resume`
4. Exit `fib()` by "going to" `return` and returning the value of `!a`

Simulate yield by callcc

```
let return = ref (Obj.magic None);;
let resume = ref (Obj.magic None);;
let fib () = callcc (fun kk -> return := kk;
   let a,b = ref 1, ref 2 in
      while true do
         callcc (fun cc -> resume := cc; throw !return !a);
         b := !a + !b;
         a := !b - !a;
      done; 0
   )
val fib :  unit -> int = <fun>
```

1. Use two references to store addresses to resume `fib` and return from it;
2. Save the return point in `return`
3. Save the resumption point in `resume`
4. Exit `fib()` by “going to” `return` and returning the value of `!a`
5. Adjust the types (the function must return an `int`)

Simulate `yield` by `callcc`

```
let return = ref (Obj.magic None);;
let resume = ref (Obj.magic None);;
let fib () = callcc (fun kk -> return := kk;
   let a,b = ref 1, ref 2 in
      while true do
         callcc (fun cc -> resume := cc; throw !return !a);
         b := !a + !b;
         a := !b - !a;
      done; 0
   )
val fib :  unit -> int = <fun>
```

1. Use two references to store addresses to resume `fib` and return from it;
2. Save the return point in `return`
3. Save the resumption point in `resume`
4. Exit `fib()` by "going to" `return` and returning the value of `!a`
5. Adjust the types (the function must return an `int`)
6. Use `callcc(fun k -> return:=k; throw  !resume ())` to resume

Example

```
# #load "callcc.cma";;
# open Callcc;;
# let return = ref (Obj.magic None);;
val return : '_a ref = contents = <poly>
# let resume = ref (Obj.magic None);;
val resume : '_a ref = contents = <poly>
# let fib() = callcc (fun kk -> return := kk;
    let a,b = ref 1, ref 2 in
        while true do
          callcc(fun cc -> (resume := cc; (throw !return !a)));
          b := !a + !b;
          a := !b - !a;
       done; 0)            ;;
val fib : unit -> int = <fun>
# fib();;
- : int = 1
# callcc (fun k -> return:=k; throw !resume ());;
- : int = 2
# callcc (fun k -> return:=k; throw !resume ());;
- : int = 3
# callcc (fun k -> return:=k; throw !resume ());;
- : int = 5
# callcc (fun k -> return:=k; throw !resume ());;
- : int = 8
# callcc (fun k -> return:=k; throw !resume ());;
- : int = 13
```

Exercise

Rewrite the previous program without the `Object.magic` so that the references contain values of type `'a Callcc.cont option` (verbose)

```
# #load "callcc.cma";;
# open Callcc;;
# let return = ref None;;
val return : '_a option ref = contents = None
# let resume = ref None;;
val resume : '_a option ref = contents = None

# let fib() = callcc (fun kk -> return := (Some kk);
  let a,b = ref 1, ref 2 in
  while true do
  callcc(fun cc -> (
           resume := (Some cc);
           let Some k = !return in (throw k !a)));
  b := !a + !b;
  a := !b - !a;
  done; 0);;
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
None
val fib : unit -> int = <fun>

# fib();;
- : int = 1
# callcc (fun k -> return:= Some k; let Some k = !resume in throw k ());;
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
None
- : int = 2
# callcc (fun k -> return:= Some k; let Some k = !resume in throw k ());;
```

Loop and tail-recursion can be encoded by `callcc`

Loop and tail-recursion can be encoded by `callcc`

```
let fib () = callcc (fun kk ->
    return := kk;
    let a,b = ref 1, ref 2 in
    callcc(fun cc -> resume := cc);
    b := !a + !b;
    a := !b - !a;
    throw !return !a)
```

Loop and tail-recursion can be encoded by `callcc`

```
let fib () = callcc (fun kk ->
    return := kk;
    let a,b = ref 1, ref 2 in
    callcc(fun cc -> resume := cc);
    b := !a + !b;
    a := !b - !a;
    throw !return !a)
```

So for instance we can avoid to call multiple times the throw ... just do not modify the return address

```
# let x = fib () in
    if x < 100 then  (
       print_int x; print_newline();
       throw !resume ())
    else ();;
1
2
3
5
8
13
21
```

Let us do it in a more functional way by using variables for `a` and `b`

```
# let resume = ref (Obj.magic None);;
val resume : '_a ref = contents = <poly>
# let fib () = callcc (fun kk ->
    let a,b = callcc(fun cc -> resume := cc ; (1,1) ) in
    throw kk (b,a+b) );;
val fib : unit -> int * int = <fun>
# let x,y = fib () in
     if x < 100 then  (
        print_int x; print_newline();
        throw !resume (x,y))
     else ();;
1
2
3
5
8
13
21
34
55
89
- : unit = ()
```

Let us do it in a more functional way by using variables for `a` and `b`

```
# let resume = ref (Obj.magic None);;
val resume : '_a ref = contents = <poly>
# let fib () = callcc (fun kk ->
    let a,b = callcc(fun cc -> resume := cc ; (1,1) ) in
    throw kk (b,a+b) );;
val fib : unit -> int * int = <fun>
# let x,y = fib () in
     if x < 100 then  (
        print_int x; print_newline();
        throw !resume (x,y))
     else ();;
1
2
3
5
8
13
21
34
55
89
- : unit = ()
```

Exercise

Modify `fib()` so as it does not need the reference `resume` for the continuation.

Coroutines

Coroutines are more generic than subroutines.

Subroutines can return only once; coroutines can return (yield) several times. Next time the coroutine is called, the execution just after the yield call.

Coroutines

Coroutines are more generic than subroutines.

Subroutines can return only once; coroutines can return (yield) several times. Next time the coroutine is called, the execution just after the yield call.

An example in pseudo-code

```
var q :=  new queue

coroutine produce
    loop
        while q is not full
           create some new items
           add the items to q
        yield to consume

coroutine consume
    loop
        while q is not empty
           remove some items from q
           use the items
        yield to produce
```

Implementing coroutines with continuations

```
coroutine process1 n =
   loop
      print "1: received "; print_ln n
      yield n+1 to process2
coroutine process2 n =
   loop
      print "2: received "; print_ln n
      yield n+1 to process1
in process1 0
```

Implementing coroutines with continuations

```
coroutine process1 n =
   loop
      print "1: received "; print_ln n
      yield n+1 to process2
coroutine process2 n =
   loop
      print "2: received "; print_ln n
      yield n+1 to process1
in process1 0
```

In OCaml with `callcc`

```
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)))))
```

Coroutines and generators

Generators are also a generalization of subroutines to define iterators

They look less expressive since the yield statement in a generator does not specify a coroutine to jump to: this is not the case:

```
generator produce
    loop
        while q is not full
           create some new items
           add the items to q
        yield consume

generator consume
    loop
        while q is not empty
           remove some items from q
           use the items
        yield produce

subroutine dispatcher
   var d := new dictionary ⟨ generator → iterator⟩
    d[produce] := start produce
    d[consume] := start consume
   var current := produce
   loop current := d[current].next()
```

Rationale

It is possible to implement coroutines on top of a generator facility, with the aid of a top-level dispatcher routine that passes control explicitly to child generators identified by tokens passed back from the generators

Generators are a much more commonly found language feature

A number of implementations of coroutines for languages with generator support but no native coroutines use this or a similar model: e.g. Perl 6, C#, Ruby, Python (prior to 2.5), ...

In OCaml there is Jérôme Vouillon's lightweight thread library (`Lwt`) that provides cooperative multi-threading. This can be implemented by coroutines (see the concurrency part of the course).

Reduction semantics for continuations

Keep the same reductions "$\rightarrow$" and the same context rules as before, and add the following rules for `callcc` and `throw`:

$$
\begin{array}{rcl}
E[\texttt{callcc}\ v] & \rightarrow & E[v(\lambda x.E[x])] \\
E[\texttt{throw}\ k\ v] & \rightarrow & kv
\end{array}
$$

(recall: the v argument of the `callcc` is a function that expects a continuation)

Same evaluation contexts E as before.

Example of reductions

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

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

Outline

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

$$
\begin{array}{rcl}
[\![N]\!] & = & \lambda k.kN \\
[\![x]\!] & = & \lambda k.kx \\
[\![\lambda x.a]\!] & = & \lambda k.k(\lambda x.[\![a]\!]) \\
[\![\texttt{let}\ \ x = a\ \ \texttt{in}\ \ b]\!] & = & \lambda k.[\![a]\!](\lambda x.[\![b]\!]k) \\
[\![a\,b]\!] & = & \lambda k.[\![a]\!](\lambda x.[\![b]\!](\lambda y.x\ y\ k))
\end{array}
$$

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

In $[\![a\,b]\!]$, the variable x (which must not be free in b) will be bound to the value returned by a and y to the value of b.

Effect on types:

if $a : \tau$ then $[\![a]\!] : ([\![\tau]\!] \to \texttt{answer}) \to \texttt{answer}$ where

$$
\begin{array}{rcll}
[\![b]\!] & = & (b \to \texttt{answer}) \to \texttt{answer} & \text{for base types } b \\
[\![\tau_1 \to \tau_2]\!] & = & [\![\tau_1]\!] \to ([\![\tau_2]\!] \to \texttt{answer}) \to \texttt{answer} &
\end{array}
$$

CPS conversion: Continuation operators

$$
\begin{array}{rcl}
[\![\texttt{callcc}\ a]\!] & = & \lambda k.[\![a]\!]\,k\,k \\
[\![\texttt{throw}\ a\ b]\!] & = & \lambda k.[\![a]\!](\lambda x.[\![b]\!](\lambda y.x\,y))
\end{array}
$$

In `callcc` a, the function value returned by $[\![a]\!]$ 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 $[\![\ldots]\!]$ produces terms that are more verbose than one would naturally write by hand. For instance, in the case of an application of a variable f to a variable x:

$$[\![f\ x]\!] = \lambda k.(\lambda k_1.k_1 f)(\lambda y_1.(\lambda k_2.k_2 x)(\lambda y_2.y_1 y_2 k))$$

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

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

whenever v is a value or variable.

Examples of CPS translation

$$[\![f(f\,x)]\!]$$
$$= \lambda k.fx(\lambda y.f\,y\,k)$$

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

Examples of CPS translation

$$[\![f(f\,x)]\!]$$
$$= \lambda k.fx(\lambda y.f\,y\,k)$$

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

Notice that the factorial function has become tail-recursive

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 (Soundness)

If $a \rightarrow^* N$, *then* $[\![a]\!](\lambda x.x) \rightarrow^* N$.

CPS terms

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

$$\begin{array}{llll} \mathit{atom} & ::= & x \mid N \mid \lambda x.\mathit{body} \mid \lambda x.\lambda k.\mathit{body} & \textbf{CPS atom} \\ \mathit{body} & ::= & \mathit{atom} \mid \mathit{atom}_1\ \mathit{atom}_2 \mid \mathit{atom}_1\ \mathit{atom}_2\ \mathit{atom}_3 & \textbf{CPS body} \end{array}$$

$[\![a]\!]$ is an atom, and $[\![a]\!](\lambda x.x)$ is a body .

Reduction of CPS terms

$$\begin{array}{rcll} atom & ::= & x \mid N \mid \lambda v.body \mid \lambda x.\lambda k.body & \textbf{CPS atom} \\ body & ::= & atom \mid atom_1\ atom_2 \mid atom_1\ atom_2\ atom_3 & \textbf{CPS body} \end{array}$$

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

Reduction of CPS terms

$$\begin{array}{rcll} atom & ::= & x \mid N \mid \lambda v.body \mid \lambda x.\lambda k.body & \textbf{CPS atom} \\ body & ::= & atom \mid atom_1\ atom_2 \mid atom_1\ atom_2\ atom_3 & \textbf{CPS body} \end{array}$$

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:

$$\begin{array}{rcl} (\lambda x.\lambda k.body)atom_1\ atom_2 & \rightarrow & body[x/atom_1, k/atom_2] \\ (\lambda x.body)atom & \rightarrow & body[x/atom] \end{array}$$

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

Reduction of CPS terms

$$\begin{array}{lcll} atom & ::= & x \mid N \mid \lambda v.body \mid \lambda x.\lambda k.body & \textbf{CPS atom} \\ body & ::= & atom \mid atom_1\ atom_2 \mid atom_1\ atom_2\ atom_3 & \textbf{CPS body} \end{array}$$

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:

$$\begin{array}{rcl} (\lambda x.\lambda k.body)atom_1\ atom_2 & \rightarrow & body[x/atom_1, k/atom_2] \\ (\lambda x.body)atom & \rightarrow & body[x/atom] \end{array}$$

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

CPS terms can be executed by a stackless abstract machine with three registers, an environment and a code pointer.

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].

CPS conversion and reduction strategy

Theorem (Indifference (Plotkin 1975))

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.

CPS conversion and reduction strategy

Theorem (Indifference (Plotkin 1975))

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.*

CPS conversion encodes the reduction strategy in the structure of the converted terms. The one we gave corresponds to left-to-right call-by-value.

$$[\![a\,b]\!] \quad = \quad \lambda k.[\![a]\!](\lambda x_a.[\![b]\!](\lambda x_b.x_a\,x_b\,k))$$

Right-to-left call-by-value is obtained by taking

$$[\![a\,b]\!] \quad = \quad \lambda k.[\![b]\!](\lambda x_b.[\![a]\!](\lambda x_a.x_a\,x_b\,k))$$

while call-by-name is achieved by taking

$$\begin{array}{rcl} [\![x]\!] & = & \lambda k.x\,k \\ [\![a\,b]\!] & = & \lambda k.[\![a]\!](\lambda x_a.x_a[\![b]\!]k) \end{array}$$

Control operators and classical logic

Control operators such as `callcc` extend the Curry-Howard correspondence from *intuitionistic logic* to *classical logic*.

Control operators and classical logic

Control operators such as `callcc` extend the Curry-Howard correspondence from *intuitionistic logic* to *classical logic*.

The *Pierce's law* $((P \to Q) \to P) \to P$ is not derivable in the intuitionistic logic while it is true in classical logic (in particular if we take $Q \equiv \bot$ then it becomes $((\neg P \to P) \to P$: if from $\neg P$ we can deduce P, then P must be true).

In terms of Curry-Howard it means that no term of the simply-typed λ-calculus has type $((P \to Q) \to P) \to P$.

Control operators and classical logic

Control operators such as `callcc` extend the Curry-Howard correspondence from *intuitionistic logic* to *classical logic*.

The *Pierce's law* $((P \to Q) \to P) \to P$ is not derivable in the intuitionistic logic while it is true in classical logic (in particular if we take $Q \equiv \bot$ then it becomes $((\neg P \to P) \to P$: if from $\neg P$ we can deduce P, then P must be true).

In terms of Curry-Howard it means that no term of the simply-typed λ-calculus has type $((P \to Q) \to P) \to P$.

But notice that

$$\texttt{callcc} : ((\alpha \to \beta) \to \alpha) \to \alpha$$

`callcc` takes as argument a function f of type $((\alpha \to \beta) \to \alpha)$ which can either return a value of type α directly or apply an argument of type α to the continuation of type $(\alpha \to \beta)$. Since the existing context is deleted when the continuation is applied, the type β is never used and may be taken to be $\bot$.

`callcc` is a proof for Pierce's law. It extends the Curry-Howard correspondence from intuitionistic logic to classical logic

`callcc` is a proof for Pierce's law. It extends the Curry-Howard correspondence from intuitionistic logic to classical logic

It is therefore possible to "prove" the excluded middle axiom $\forall P.P \vee \neg P$.

Modulo Curry-Howard, this axiom corresponds to the type $\forall P.P + (P \to \mathit{False})$, where *False* is an empty type and $A + B$ is a datatype with two constructors
$\mathit{Left} : A \to A + B$ and $\mathit{Right} : B \to A + B$.
The following term "implements" (ie, it proves) excluded middle:

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

`callcc` is a proof for Pierce's law. It extends the Curry-Howard correspondence from intuitionistic logic to classical logic

It is therefore possible to "prove" the excluded middle axiom $\forall P.P \vee \neg P$.

Modulo Curry-Howard, this axiom corresponds to the type $\forall P.P + (P \to \mathit{False})$, where *False* is an empty type and $A + B$ is a datatype with two constructors
$\mathit{Left} : A \to A + B$ and $\mathit{Right} : B \to A + B$.
The following term "implements" (ie, it proves) excluded middle:

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

What about the CPS translation?

CPS and double negation

Let $\neg A = (A \to \bot)$ where $\bot$ represent "false". In intuitionistic logic

$$\vdash A \to \neg\neg A$$

whose proof is $\lambda x{:}A.\lambda f{:}\neg A.fx$.

CPS and double negation

Let $\neg A = (A \to \bot)$ where $\bot$ represent "false". In intuitionistic logic

$$\vdash A \to \neg\neg A$$

whose proof is $\lambda x{:}A.\lambda f{:}\neg A.fx$. On the other hand:

$$\not\vdash \neg\neg A \to A$$

It is not possible to define a closed λ-term of the type above.
However

$$\vdash \neg\neg\neg A \to \neg A$$

whose proof is: $\lambda f : \neg\neg\neg A.\lambda x : A.f(\lambda g : \neg A.gx)$.
This suggests a *double negation* translation from classical to intuitionistic logic:

- $[\![\phi]\!] = \neg\neg\phi$ if ϕ is *atomic* (ie, a basic type)
- $[\![A \to B]\!] = [\![A]\!] \to [\![B]\!]$

CPS and double negation

Theorem (Glivenko 1929)

$$\vdash_{classic} A \quad \textit{iff} \quad \vdash_{intuitionistic} [A]$$

CPS and double negation

Theorem (Glivenko 1929)

$$\vdash_{classic} A \quad \textit{iff} \quad \vdash_{intuitionistic} [\![A]\!]$$

In terms of the Curry Howard isomorphism

$$\vdash_{classic} M : A \quad \text{iff} \quad \vdash_{intuitionistic} [\![M]\!] : [\![A]\!]$$

where $[\![M]\!]$ is (essentially) the CPS translation of M.

CPS and double negation

Theorem (Glivenko 1929)

$$\vdash_{classic} A \quad \textit{iff} \quad \vdash_{intuitionistic} [\![A]\!]$$

In terms of the Curry Howard isomorphism

$$\vdash_{classic} M : A \quad \text{iff} \quad \vdash_{intuitionistic} [\![M]\!] : [\![A]\!]$$

where $[\![M]\!]$ is (essentially) the CPS translation of M.

So the CPS translation extends the Curry-Howard isomorphism to the "double negation *encoding*" of the classical propositional logic

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

References

- A. Appel. Programming with continuations.
- Slides of the course *Functional Programming Languages* by Xavier Leroy (from which the slides of this and the following part heavily borrowed) available on the web:
  `http://cristal.inria.fr/~xleroy/mpri/progfunc`