Delimited dynamic binding

Oleg Kiselyov (FNMOC)
Chung-chieh Shan (Rutgers University)
Amr Sabry (Indiana University)

ICFP 2006

Contributions

1. The interaction of dynamic binding and delimited control was undefined and unusable
   - not undelimited control
   - not `dynamic-wind`
2. Specify delimited dynamic binding: a continuation
   - closes over part of the dynamic environment when captured
   - supplements the dynamic environment when invoked

   (like ordinary functional abstractions)
3. Translate (typed) dynamic binding to (typed) delimited control
4. Implement in Scheme, OCaml, Haskell
5. Extensions: mutable dynamic variables, stack inspection

Outline

- **Dynamic binding**

Delimited control

The problem

Delimited dynamic binding

Translation from DB to DC

Dynamic binding: pretty printing

```
let rec pretty_expr: expr -> string
     = ..........................
       ..........................
    and pretty_stmt: stmt -> string
     = ......... pretty_expr .....
       ..........................
    and pretty_prog: prog -> string
     = .......... pretty_expr ....
       ....... pretty_stmt ........;;

pretty_prog my_program
```

Dynamic binding: pretty printing

```
let rec pretty_expr: expr -> string
     = .......... 80 ..............
       ........ 80 ................
    and pretty_stmt: stmt -> string
     = .......... pretty_expr .....
       ............ 80 ............
    and pretty_prog: prog -> string
     = ............ pretty_expr ....
       ....... pretty_stmt ........;;

pretty_prog my_program
```

Dynamic binding: pretty printing

```
let rec pretty_expr: int -> expr -> string
     = ......... width ..................
       ....... width ....................
    and pretty_stmt: int -> stmt -> string
     = .......... pretty_expr width .....
       .......... width ..................
    and pretty_prog: int -> prog -> string
     = ........... pretty_expr width ....
       ....... pretty_stmt width ........;;

pretty_prog 80 my_program
```

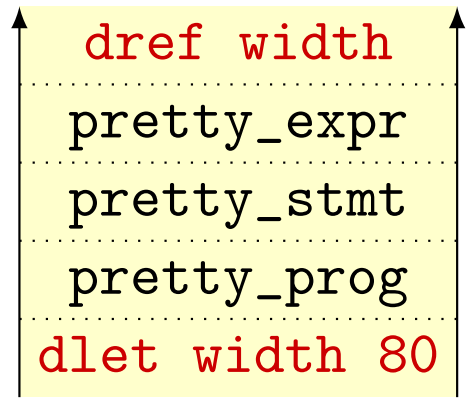
Dynamic binding: pretty printing

```
dref: 'a dynvar -> 'a
dlet: 'a dynvar -> 'a -> (unit -> 'b) -> 'b


let rec pretty_expr: expr -> string
     = .... dref width ............
       .. dref width ..............
    and pretty_stmt: stmt -> string
     = .......... pretty_expr .....
       ..... dref width ...........
    and pretty_prog: prog -> string
     = ........... pretty_expr ....
       ....... pretty_stmt ........;;

dlet width 80 (fun () -> pretty_prog my_program)
```

Dynamic binding: pretty printing

```
dref: 'a dynvar -> 'a
dlet: 'a dynvar -> 'a -> (unit -> 'b) -> 'b
```

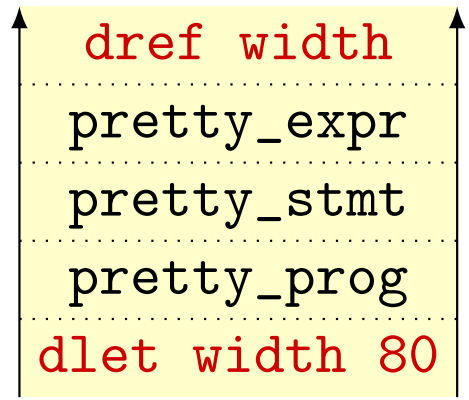
```
let rec pretty_expr: expr -> string
     = .... dref width ............
       .. dref width ..............
    and pretty_stmt: stmt -> string
     = .......... pretty_expr .....
       ..... dref width ...........
    and pretty_prog: prog -> string
     = ........... pretty_expr ....
       ....... pretty_stmt ........;;
```




```
dlet width 80 (fun () -> pretty_prog my_program)
```

Dynamic binding: pretty printing

```
dnew: unit       -> 'a dynvar
dref: 'a dynvar -> 'a
dlet: 'a dynvar -> 'a -> (unit -> 'b) -> 'b

let width = dnew ();;
let rec pretty_expr: expr -> string
     = .... dref width ............
       .. dref width ..............
    and pretty_stmt: stmt -> string
     = .......... pretty_expr .....
       ..... dref width ...........
    and pretty_prog: prog -> string
     = ........... pretty_expr ....
       ....... pretty_stmt ........;;

dlet width 80 (fun () -> pretty_prog my_program)
```



Dynamic binding: summary

Many applications

- Implicit arguments
- I/O redirection
- Exception handlers
- Mobile code
- Web applications
- ...

Dynamic binding: summary

Many applications

- Implicit arguments
- I/O redirection
- Exception handlers
- Mobile code
- Web applications
- ...

Many implementations

- Pass implicit argument (*dynamic environment*) everywhere
- Global mutable cells (*shallow binding*)
- ...

Dynamic binding: summary

Many applications

- Implicit arguments
- I/O redirection
- Exception handlers
- Mobile code
- Web applications
- ...

Many implementations

- Pass implicit argument (*dynamic environment*) everywhere
- Global mutable cells (*shallow binding*)
- ...

Outline

- Dynamic binding
- **Delimited control**
- The problem
- Delimited dynamic binding
- Translation from DB to DC

Delimited control is not call/cc

```
new_prompt : unit       -> 'a prompt
push_prompt: 'a prompt -> (unit -> 'a) -> 'a
shift      : 'a prompt -> (('b -> 'a) -> 'a) -> 'b
```

Delimited control is not call/cc

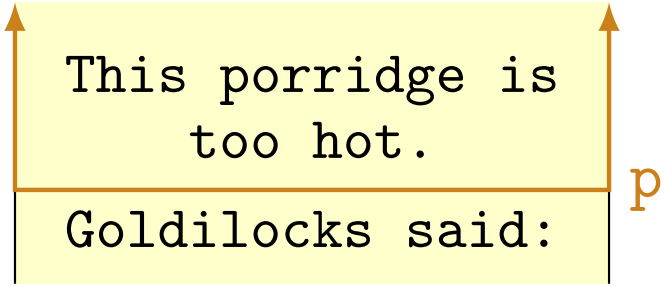
```
new_prompt : unit       -> 'a prompt
push_prompt: 'a prompt -> (unit -> 'a) -> 'a
shift      : 'a prompt -> (('b -> 'a) -> 'a) -> 'b

let p = new_prompt ()
in "Goldilocks said: " ^
   push_prompt p (fun () ->
     "This porridge is " ^
     "too hot" ^ ". ");;
```

```
Goldilocks said: This porridge is too hot.
```

Delimited control is not call/cc

```
new_prompt : unit        -> 'a prompt
push_prompt: 'a prompt -> (unit -> 'a) -> 'a
shift      : 'a prompt -> (('b -> 'a) -> 'a) -> 'b
```

```
let p = new_prompt ()
in "Goldilocks said: " ^
   push_prompt p (fun () ->
     "This porridge is " ^
     "too hot" ^ ". ");;
```

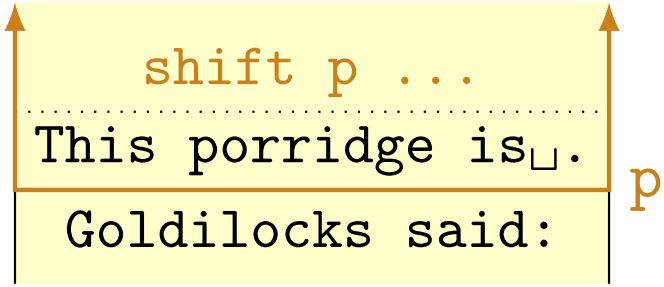



```
Goldilocks said: This porridge is too hot.
```

Delimited control is not call/cc

```
new_prompt : unit       -> 'a prompt
push_prompt: 'a prompt -> (unit -> 'a) -> 'a
shift      : 'a prompt -> (('b -> 'a) -> 'a) -> 'b
```

```
let p = new_prompt ()
in "Goldilocks said: " ^
   push_prompt p (fun () ->
     "This porridge is " ^
     (shift p (fun f -> f "too hot" ^
                        f "too cold" ^
                        f "just right")) ^ ". ");;
```




```
Goldilocks said: This porridge is too hot. This
porridge is too cold. This porridge is just right.
```

Delimited control: summary

Many applications

- Backtracking search
- Functional abstraction
- Partial evaluation
- Mobile code
- Web applications
- ...

Delimited control: summary

Many applications

- Backtracking search
- Functional abstraction
- Partial evaluation
- Mobile code
- Web applications
- ...

Many implementations

- Filinski: via call/cc and state
- CPS translation
- Native
  - Gasbichler and Sperber for Scheme 48
  - Now in OCaml

Outline

- Dynamic binding
- Delimited control
- **The problem**
- Delimited dynamic binding
- Translation from DB to DC

DB/DC interaction is undefined and unusable

Real-world examples in the paper:

- Mobile code
- Server-side Web applications
- Database cursors (iterators)

Need to *combine* dynamic environments inside and outside the control delimiter, not just *switch* among them

A cursor is a lazy list

Delay the evaluation of each cons cell.

```
type 'a cursor = Cursor of
    (unit -> ('a * 'a cursor) option);;
```

A cursor is a lazy list

Delay the evaluation of each cons cell.

```
type 'a cursor = Cursor of
    (unit -> ('a * 'a cursor) option);;

let test_cursor: char cursor
 = Cursor (fun () -> Some ('a',
   Cursor (fun () -> Some ('b',
   Cursor (fun () -> Some ('c',
   Cursor (fun () -> None))))));;
```

A cursor is a lazy list

Delay the evaluation of each cons cell.

```
type 'a cursor = Cursor of
    (unit -> ('a * 'a cursor) option);;

let test_cursor: char cursor
 = Cursor (fun () -> Some ('a',
   Cursor (fun () -> Some ('b',
   Cursor (fun () -> Some ('c',
   Cursor (fun () -> None))))));;

let next (Cursor cur: 'a cursor): 'a cursor
 = match cur ()
   with None -> failwith "next"
      | Some (head, tail) -> tail;;
```

A cursor is a lazy list

Delay the evaluation of each cons cell.

```
type 'a cursor = Cursor of
    (unit -> ('a * 'a cursor) option);;

let dump (path: string) (c: char cursor): unit
 = let channel = open_out path
   in let rec loop (Cursor cur)
       = match cur ()
         with None -> ()
            | Some (head, tail) ->
                output_char channel head; loop tail
      in try loop c; close_out channel
         with exc -> close_out channel; raise exc;;

dump "test" test_cursor;;
```

Delimited control turns an enumerator into a cursor

Suspend enumeration inside the callback for each item.

```
type 'a enum = ('a -> unit) -> unit;;
```

Delimited control turns an enumerator into a cursor

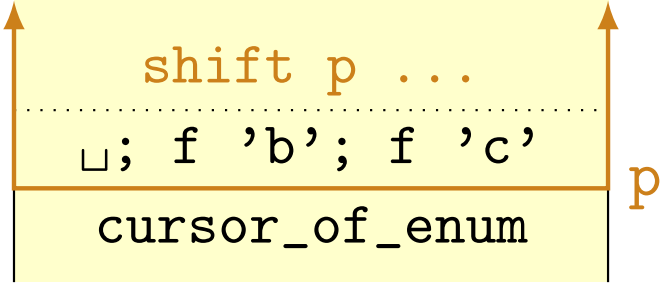
Suspend enumeration inside the callback for each item.

```
type 'a enum = ('a -> unit) -> unit;;

let test_enum f = f 'a'; f 'b'; f 'c';;
```

Delimited control turns an enumerator into a cursor

Suspend enumeration inside the callback for each item.

```
type 'a enum = ('a -> unit) -> unit;;

let test_enum f = f 'a'; f 'b'; f 'c';;

let cursor_of_enum (e: 'a enum): 'a cursor
 = Cursor (fun () ->
     let p = new_prompt ()
     in push_prompt p (fun () ->
          e (fun a -> shift p (fun k ->
                                 Some (a, Cursor k)));
          None));;

dump "test"
  (cursor_of_enum test_enum);;
```



An enumerator can handle exceptions

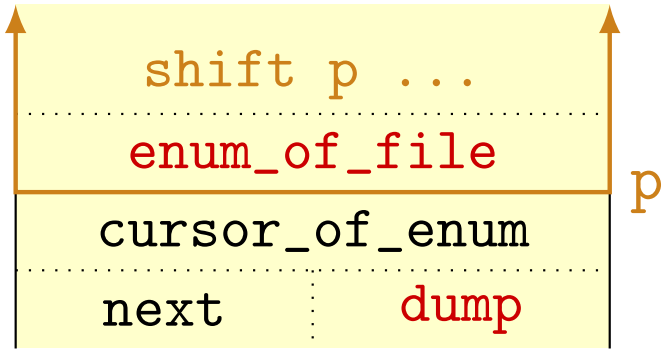
If an error occurs while reading a file, close the file before rethrowing the exception.

```
let enum_of_file (path: string): char enum
 = fun f ->
     let channel = open_in path
     in let rec loop ()
         = match try Some (input_char channel)
                 with End_of_file -> None
           with None -> ()
              | Some s -> loop (f s)
        in try loop ();
               close_in channel
           with exc -> close_in channel;
                       raise exc;;
```

Putting it all together

The cursor and its client may both handle the same exception.

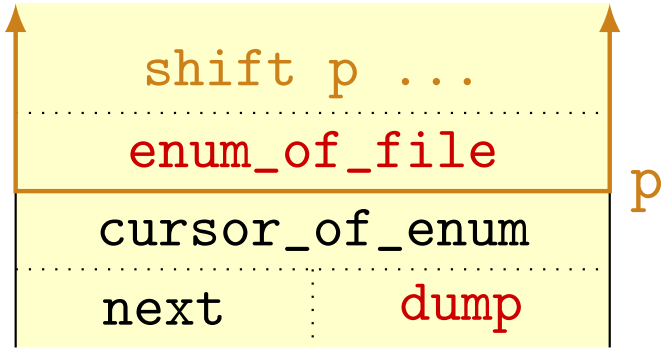
```
dump "test"
  (next
    (cursor_of_enum
      (enum_of_file
        "/dev/random")));;
```



Putting it all together

The cursor and its client may both handle the same exception.

```
dump "test"
  (next
    (cursor_of_enum
      (enum_of_file
        "/dev/random")));;
```



Dynamic binding
(of exception handlers)

Delimited control
(to suspend enumeration)

One file stays unclosed!

Contributions

1. Point out that DB/DC interaction was undefined and unusable
   - Filinski's DC in terms of call/cc and state closes over an entire dynamic environment at once
   - Filinski's layered monads forces each continuation to close over a fixed set of dynamic variables

   (see accompanying code)
2. Specify delimited dynamic binding: a continuation
   - closes over part of the dynamic environment when captured
   - supplements the dynamic environment when invoked

   (like ordinary functional abstractions)
3. Translate (typed) DB to (typed) DC
4. Implement in Scheme, OCaml, Haskell
5. Extensions: mutable dynamic variables, stack inspection

Contributions

1. Point out that DB/DC interaction was undefined and unusable
   - Filinski's DC in terms of call/cc and state closes over an entire dynamic environment at once
   - Filinski's layered monads forces each continuation to close over a fixed set of dynamic variables

   (see accompanying code)
2. Specify delimited dynamic binding: a continuation
   - closes over part of the dynamic environment when captured
   - supplements the dynamic environment when invoked

   (like ordinary functional abstractions)
3. Translate (typed) DB to (typed) DC
4. Implement in Scheme, OCaml, Haskell
5. Extensions: mutable dynamic variables, stack inspection

Contributions

1. Point out that DB/DC interaction was undefined and unusable
   - Filinski's DC in terms of call/cc and state closes over an entire dynamic environment at once
   - Filinski's layered monads forces each continuation to close over a fixed set of dynamic variables

   (see accompanying code)
2. Specify delimited dynamic binding: a continuation
   - closes over part of the dynamic environment when captured
   - supplements the dynamic environment when invoked

   (like ordinary functional abstractions)
3. Translate (typed) DB to (typed) DC
4. Implement in Scheme, OCaml, Haskell
5. Extensions: mutable dynamic variables, stack inspection

Contributions

1. Point out that DB/DC interaction was undefined and unusable
   - Filinski's DC in terms of call/cc and state closes over an entire dynamic environment at once
   - Filinski's layered monads forces each continuation to close over a fixed set of dynamic variables

   (see accompanying code)
2. Specify delimited dynamic binding: a continuation
   - closes over part of the dynamic environment when captured
   - supplements the dynamic environment when invoked

   (like ordinary functional abstractions)
3. Translate (typed) DB to (typed) DC
4. Implement in Scheme, OCaml, Haskell
5. Extensions: mutable dynamic variables, stack inspection

Contributions

1. Point out that DB/DC interaction was undefined and unusable
   - Filinski's DC in terms of call/cc and state closes over an entire dynamic environment at once
   - Filinski's layered monads forces each continuation to close over a fixed set of dynamic variables

   (see accompanying code)
2. Specify delimited dynamic binding: a continuation
   - closes over part of the dynamic environment when captured
   - supplements the dynamic environment when invoked

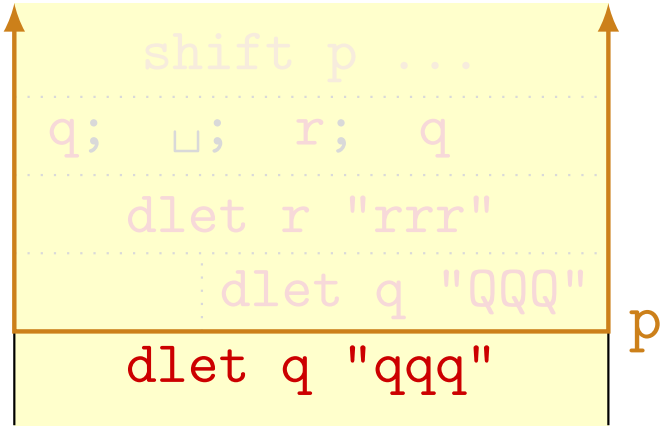
   (like ordinary functional abstractions)
3. Translate (typed) DB to (typed) DC
4. Implement in Scheme, OCaml, Haskell
5. Extensions: mutable dynamic variables, stack inspection

Outline

- Dynamic binding
- Delimited control
- The problem
- ► **Delimited dynamic binding**
- Translation from DB to DC

Delimited dynamic binding

```
let p = new_prompt ()
and q = dnew ()
and r = dnew ()
in dlet q "qqq" (fun () ->
     push_prompt p (fun () ->
       dlet r "rrr" (fun () ->
         print_endline (dref q);
         shift p (fun f -> dlet q "QQQ" f);
         print_endline (dref r);
         print_endline (dref q))))
```

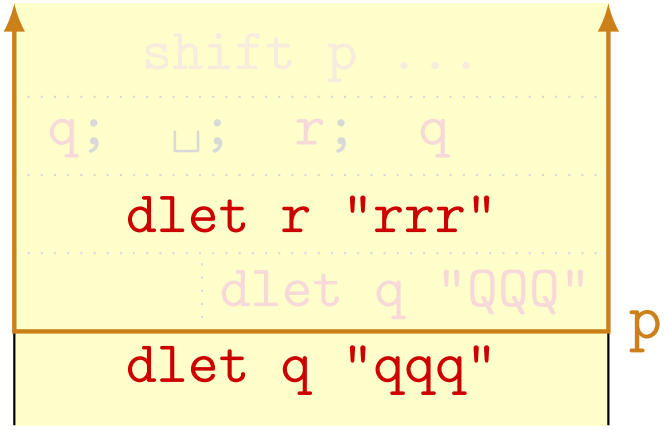



```
qqq                qqq
rrr                rrr
QQQ                qqq
```

Delimited dynamic binding

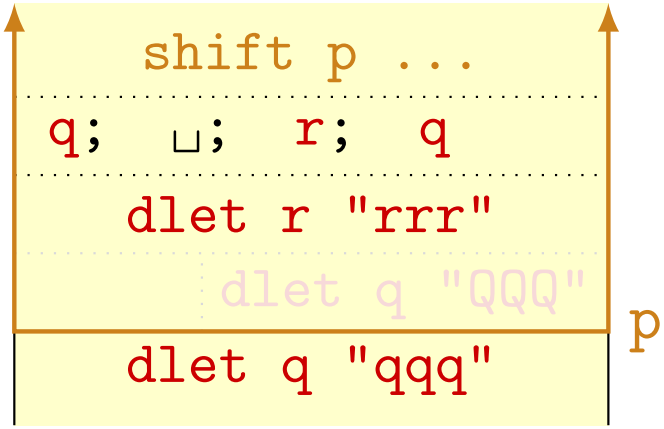
```
let p = new_prompt ()
and q = dnew ()
and r = dnew ()
in dlet q "qqq" (fun () ->
     push_prompt p (fun () ->
       dlet r "rrr" (fun () ->
         print_endline (dref q);
         shift p (fun f -> dlet q "QQQ" f);
         print_endline (dref r);
         print_endline (dref q))))
```

```
qqq          qqq
rrr          rrr
QQQ          qqq
```



Delimited dynamic binding

```
let p = new_prompt ()
and q = dnew ()
and r = dnew ()
in dlet q "qqq" (fun () ->
     push_prompt p (fun () ->
       dlet r "rrr" (fun () ->
         print_endline (dref q);
         shift p (fun f -> dlet q "QQQ" f);
         print_endline (dref r);
         print_endline (dref q))))
```



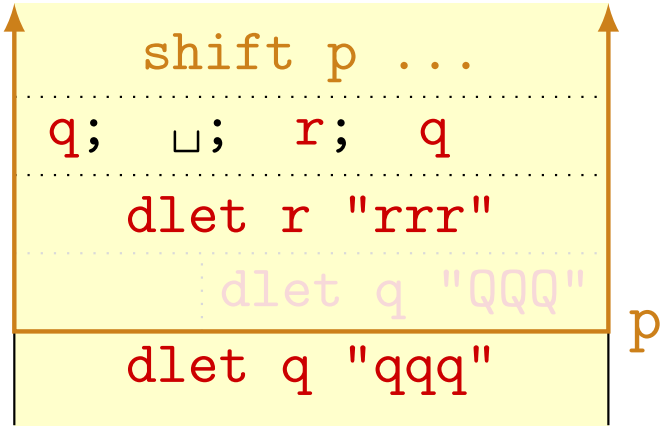

```
qqq                qqq
rrr                rrr
QQQ                qqq
```

Delimited dynamic binding

```
let p = new_prompt ()
and q = dnew ()
and r = dnew ()
in dlet q "qqq" (fun () ->
     push_prompt p (fun () ->
       dlet r "rrr" (fun () ->
         print_endline (dref q);
         shift p (fun f -> dlet q "QQQ" f);
         print_endline (dref r);
         print_endline (dref q))))
```

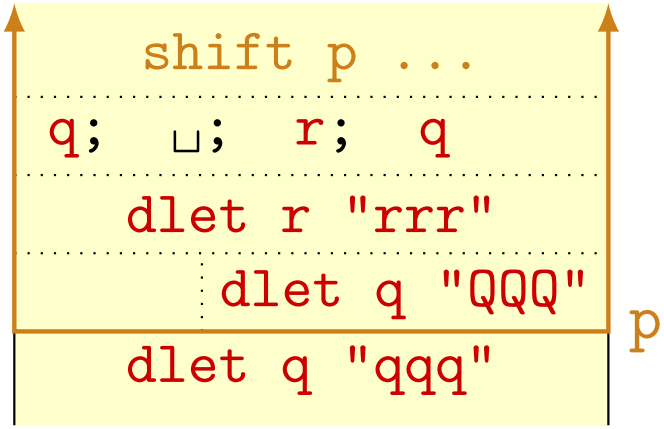
```
qqq
rrr
QQQ
```

```
qqq
rrr
qqq
```



Delimited dynamic binding

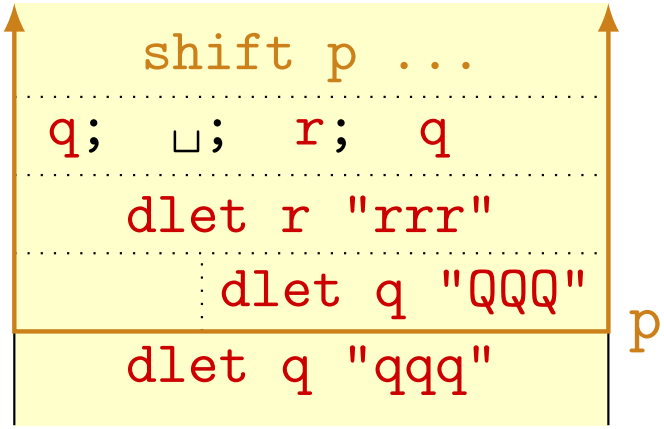
```
let p = new_prompt ()
and q = dnew ()
and r = dnew ()
in dlet q "qqq" (fun () ->
     push_prompt p (fun () ->
       dlet r "rrr" (fun () ->
         print_endline (dref q);
         shift p (fun f -> dlet q "QQQ" f);
         print_endline (dref r);
         print_endline (dref q))))
```




```
qqq                qqq
rrr                rrr
QQQ                qqq
```

Delimited dynamic binding

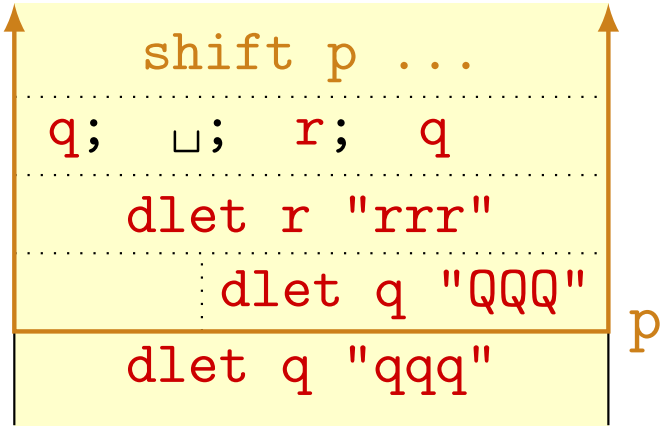
```
let p = new_prompt ()
and q = dnew ()
and r = dnew ()
in dlet q "qqq" (fun () ->
     push_prompt p (fun () ->
       dlet r "rrr" (fun () ->
         print_endline (dref q);
         shift p (fun f -> dlet q "QQQ" f);
         print_endline (dref r);
         print_endline (dref q))))
```




```
qqq                qqq
rrr                rrr
QQQ                qqq
```

Delimited dynamic binding

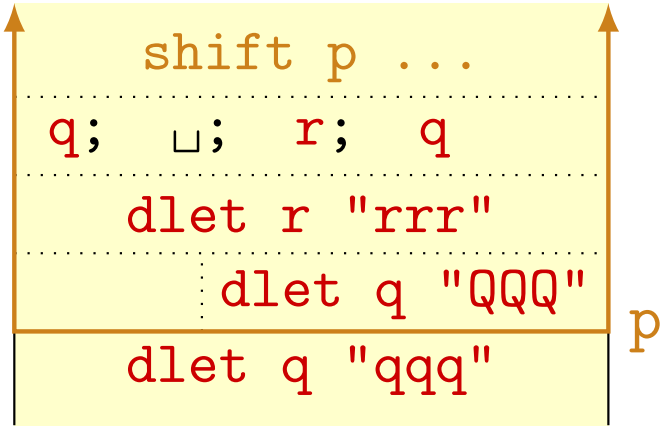
```
let p = new_prompt ()
and q = dnew ()
and r = dnew ()
in dlet q "qqq" (fun () ->
     push_prompt p (fun () ->
       dlet r "rrr" (fun () ->
         print_endline (dref q);
         shift p (fun f -> dlet q "QQQ" f);
         print_endline (dref r);
         print_endline (dref q))))
```




```
qqq                qqq
rrr                rrr
QQQ                qqq
```

Delimited dynamic binding

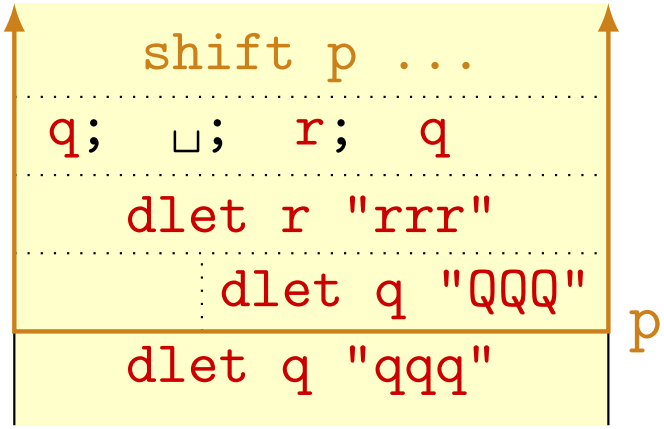
```
let p = new_prompt ()
and q = dnew ()
and r = dnew ()
in dlet q "qqq" (fun () ->
     push_prompt p (fun () ->
       dlet r "rrr" (fun () ->
         print_endline (dref q);
         shift p (fun f -> dlet q "QQQ" f);
         print_endline (dref r);
         print_endline (dref q))))
```




```
qqq                qqq
rrr                rrr
QQQ                qqq
```

Delimited dynamic binding

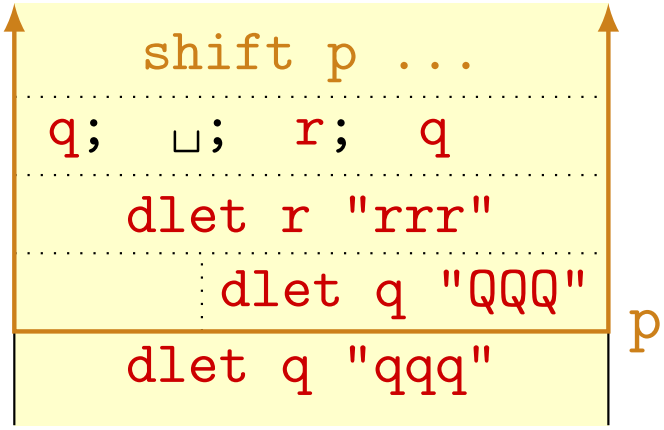
```
let p = new_prompt ()
and q = dnew ()
and r = dnew ()
in dlet q "qqq" (fun () ->
     push_prompt p (fun () ->
       dlet r "rrr" (fun () ->
         print_endline (dref q);
         shift p (fun f -> dlet q "QQQ" f);
         print_endline (dref r);
         print_endline (dref q))))
```




```
qqq             qqq       undefined
rrr             rrr       rrr
QQQ             qqq       undefined
```

Delimited dynamic binding

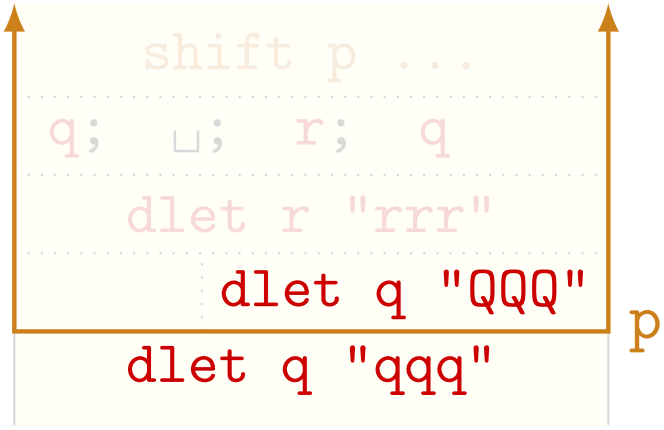
```
let p = new_prompt ()
and q = dnew ()
and r = dnew ()
in dlet q "qqq" (fun () ->
     push_prompt p (fun () ->
       dlet r "rrr" (fun () ->
         print_endline (dref q);
         shift p (fun f -> dlet q "QQQ" f);
         print_endline (dref r);
         print_endline (dref q))))
```




```
qqq                qqq        undefined
rrr                rrr        rrr
QQQ                qqq        undefined
```

Delimited dynamic binding

```
let p = new_prompt ()
and q = dnew ()
and r = dnew ()
in dlet q "qqq" (fun () ->
     push_prompt p (fun () ->
       dlet r "rrr" (fun () ->
         print_endline (dref q);
         shift p (fun f -> dlet q "QQQ" f);
         print_endline (dref r);
         print_endline (dref q))))
```




```
qqq          qqq          undefined
rrr          rrr          rrr
QQQ          qqq          undefined
```

Outline

- Dynamic binding
- Delimited control
- The problem
- Delimited dynamic binding
- **Translation from DB to DC**

DB, the language of dynamic binding (abridged)

Terms $\quad M ::= V \mid MM \mid p \mid \mathrm{dlet}\ p = V\ \mathrm{in}\ M$

Parameters $\quad p ::= p \mid q \mid r \mid \cdots$

Contexts $\quad E[\ ] ::= [\ ] \mid E[[\ ]M] \mid E[V[\ ]] \mid E[\mathrm{dlet}\ p = V\ \mathrm{in}\ [\ ]]$

$$E[(\lambda x.M)V] \mapsto E[M\{V/x\}]$$

$$E[\mathrm{dlet}\ p = V\ \mathrm{in}\ V'] \mapsto E[V']$$

$$E[\mathrm{dlet}\ p = V\ \mathrm{in}\ E'[p]] \mapsto E[\mathrm{dlet}\ p = V\ \mathrm{in}\ E'[V]] \quad \text{if } p \notin \mathrm{BP}(E')$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash_\Sigma x : \tau} \qquad \frac{\Gamma, x : \tau_1 \vdash_\Sigma M : \tau_2}{\Gamma \vdash_\Sigma \lambda x.M : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash_\Sigma M_1 : \tau_2 \to \tau \quad \Gamma \vdash_\Sigma M_2 : \tau_2}{\Gamma \vdash_\Sigma M_1 M_2 : \tau}$$

$$\frac{\Sigma(p) = \tau}{\Gamma \vdash_\Sigma p : \tau} \qquad \frac{\Sigma(p) = \tau_1 \quad \Gamma \vdash_\Sigma V : \tau_1 \quad \Gamma \vdash_\Sigma M : \tau_2}{\Gamma \vdash_\Sigma \mathrm{dlet}\ p = V\ \mathrm{in}\ M : \tau_2}$$

DB, the language of dynamic binding (abridged)

$$\begin{array}{ll} \text{Terms} & M ::= V \mid MM \mid p \mid \text{dlet } p = V \text{ in } M \\ \text{Parameters} & p ::= p \mid q \mid r \mid \cdots \\ \text{Contexts} & E[\,] ::= [\,] \mid E[[\,]M] \mid E[V[\,]] \mid E[\text{dlet } p = V \text{ in } [\,]] \end{array}$$

Type safety is mechanized in Twelf:

- An evaluation context is a function from terms to terms.
- Evaluation contexts and prevalues depend on each other.

$$E[(\lambda x.M)V] \mapsto E[M\{V/x\}]$$

$$E[\text{dlet } p = V \text{ in } V'] \mapsto E[V']$$

$$E[\text{dlet } p = V \text{ in } E'[p]] \mapsto E[\text{dlet } p = V \text{ in } E'[V]] \quad \text{if } p \notin \text{BP}(E')$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash_\Sigma x : \tau} \qquad \frac{\Gamma, x : \tau_1 \vdash_\Sigma M : \tau_2}{\Gamma \vdash_\Sigma \lambda x.M : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash_\Sigma M_1 : \tau_2 \to \tau \quad \Gamma \vdash_\Sigma M_2 : \tau_2}{\Gamma \vdash_\Sigma M_1 M_2 : \tau}$$

$$\frac{\Sigma(p) = \tau}{\Gamma \vdash_\Sigma p : \tau} \qquad \frac{\Sigma(p) = \tau_1 \quad \Gamma \vdash_\Sigma V : \tau_1 \quad \Gamma \vdash_\Sigma M : \tau_2}{\Gamma \vdash_\Sigma \text{dlet } p = V \text{ in } M : \tau_2}$$

DB, the language of dynamic binding (abridged)

Terms $\quad M ::= V \mid MM \mid p \mid \mathrm{dlet}\ p = V\ \mathrm{in}\ M$

Parameters $\quad p ::= p \mid q \mid r \mid \cdots$

Contexts $\quad E[\ ] ::= [\ ] \mid E[[\ ]M] \mid E[V[\ ]] \mid E[\mathrm{dlet}\ p = V\ \mathrm{in}\ [\ ]]$

$$E[(\lambda x.M)V] \mapsto E[M\{V/x\}]$$

$$E[\mathrm{dlet}\ p = V\ \mathrm{in}\ V'] \mapsto E[V']$$

$$E[\mathrm{dlet}\ p = V\ \mathrm{in}\ E'[p]] \mapsto E[\mathrm{dlet}\ p = V\ \mathrm{in}\ E'[V]] \quad \text{if } p \notin \mathrm{BP}(E')$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash_\Sigma x : \tau} \qquad \frac{\Gamma, x : \tau_1 \vdash_\Sigma M : \tau_2}{\Gamma \vdash_\Sigma \lambda x.M : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash_\Sigma M_1 : \tau_2 \to \tau \quad \Gamma \vdash_\Sigma M_2 : \tau_2}{\Gamma \vdash_\Sigma M_1 M_2 : \tau}$$

$$\frac{\Sigma(p) = \tau}{\Gamma \vdash_\Sigma p : \tau} \qquad \frac{\Sigma(p) = \tau_1 \quad \Gamma \vdash_\Sigma V : \tau_1 \quad \Gamma \vdash_\Sigma M : \tau_2}{\Gamma \vdash_\Sigma \mathrm{dlet}\ p = V\ \mathrm{in}\ M : \tau_2}$$

DC, the language of delimited control (abridged)

$$\begin{array}{ll} \text{Terms} & M ::= V \mid MM \mid \text{shift } p \text{ as } f \text{ in } M \mid \text{reset } p \text{ in } M \\ \text{Prompts} & p ::= p \mid q \mid r \mid \cdots \\ \text{Contexts} & E[\,] ::= [\,] \mid E\big[[\,]M\big] \mid E\big[V[\,]\big] \mid E\big[\text{reset } p \text{ in } [\,]\big] \end{array}$$

$$E\big[(\lambda x.M)V\big] \mapsto E\big[M\{V/x\}\big]$$

$$E\big[\text{reset } p \text{ in } V'\big] \mapsto E\big[V'\big]$$

$$E\big[\text{reset } p \text{ in } E'[\text{shift } p \text{ as } f \text{ in } M]\big] \mapsto E\big[\text{reset } p \text{ in } M\{V/f\}\big]$$

if $p \notin \mathrm{CP}(E')$ and $V = \lambda y.\,\text{reset } p \text{ in } E'[y]$, where y is fresh

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash_\Sigma x : \tau} \qquad \frac{\Gamma, x : \tau_1 \vdash_\Sigma M : \tau_2}{\Gamma \vdash_\Sigma \lambda x.M : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash_\Sigma M_1 : \tau_2 \to \tau \quad \Gamma \vdash_\Sigma M_2 : \tau_2}{\Gamma \vdash_\Sigma M_1 M_2 : \tau}$$

$$\frac{\Sigma(p) = \tau_2 \quad \Gamma, f : \tau \to \tau_2 \vdash_\Sigma M : \tau_2}{\Gamma \vdash_\Sigma \text{shift } p \text{ as } f \text{ in } M : \tau} \qquad \frac{\Sigma(p) = \tau \quad \Gamma \vdash_\Sigma M : \tau}{\Gamma \vdash_\Sigma \text{reset } p \text{ in } M : \tau}$$

Translation from DB to DC

Typed dynamic variables

```
let dnew () = new_prompt ()

let dlet p v body
 = let q = new_prompt ()
   in push_prompt q (fun () ->
        ignore (push_prompt p (fun () ->
                  let z = body ()
                  in shift q (fun _ -> z)) v);
        failwith "cannot happen")

let dref p    = shift p (fun f -> fun v -> f v v)
```

Translation from DB to DC

Typed, mutable dynamic variables

```
let dnew () = new_prompt ()

let dlet p v body
 = let q = new_prompt ()
   in push_prompt q (fun () ->
        ignore (push_prompt p (fun () ->
                  let z = body ()
                  in shift q (fun _ -> z)) v);
        failwith "cannot happen")

let dref p    = shift p (fun f -> fun v -> f v v)
let dset p v' = shift p (fun f -> fun v -> f v v')
```

Translation from DB to DC

Typed, mutable dynamic variables with stack inspection

```
let dnew () = new_prompt ()

let dlet p v body
 = let q = new_prompt ()
   in push_prompt q (fun () ->
        ignore (push_prompt p (fun () ->
                  let z = body ()
                  in shift q (fun _ -> z)) v);
        failwith "cannot happen")

let dref p    = shift p (fun f -> fun v -> f v v)
let dset p v' = shift p (fun f -> fun v -> f v v')
let dupp p g  = shift p (fun f -> fun v -> f (g v) v)
```

Translation from DB + DC to DC

Typed, mutable dynamic variables with stack inspection

```
let dnew () = new_prompt ()

let dlet p v body
 = let q = new_prompt ()
   in push_prompt q (fun () ->
        ignore (push_prompt p (fun () ->
                  let z = body ()
                  in shift q (fun _ -> z)) v);
        failwith "cannot happen")

let dref p    = shift p (fun f -> fun v -> f v v)
let dset p v' = shift p (fun f -> fun v -> f v v')
let dupp p g  = shift p (fun f -> fun v -> f (g v) v)
```

Summary

- Dynamic binding and delimited control belong together
- The continuation is the universal implicit argument
- Many implementation strategies
- Need delimited `dynamic-wind` (works in Scheme 48)