Closures — the Forth way

M. Anton Ertl, TU Wien
Bernd Paysan, net2o

Problem

Given

```
numint ( a b xt -- r )
 with xt ( x -- z )
```

which computes $r = \int_a^b \mathrm{xt}(x)\,\mathrm{d}x$, we want

```
integrate-1/x^y ( a b y -- r )
```

which computes $r = \int_a^b 1/x^y\,\mathrm{d}x$

How do we get y into the xt?

In general: How to pass extra parameters to xts executed elsewhere

Solution: Closures

```
: integrate-1/x^y ( a b y -- r )
  [{: f: y :}l ( x -- z ) y fnegate f** ;] numint ;
```

Principles:

- Explicit memory management of closures
  `:}l :}h :}d :}* :}xt`

- Explicit flat closures
  Manual closure conversion

- Assignment conversion for writable locals
  Pass the address, access with `@ !` etc.

Closures: Explicit memory management

```
: 1/x^y ( y -- xt )
  [{: f: y :}h ( x -- r ) y fnegate f** ;] ;

( a b y ) 1/x^y dup numint >addr free throw
```

Alternative: Stack underground

```
numint ( ... a b xt -- ... r )
\ with xt ( ... x -- ... z )

: integrate-1/x^y ( a b y -- r )
  frot frot ( y a b )
  [: ( y x -- y z )
    fover fnegate f** ;]
  numint fswap fdrop ;
```

Hard to follow in multi-level cases

Assignment conversion and defer-flavoured locals

Compute $\sum_{i=1}^{20} 1/i^2$

```
: for ( ... u xt -- ... )
    \ xt ( ... u1 -- ... )
    {: xt: xt :} 1+ 1 ?do i xt loop ;

: sum-series ( ... u xt -- ... r )
    \ xt ( ... u1 -- ... r1 )
    0e {: f^ ra :}
    ra [{: xt: xt ra :}l ( ... u1 -- ... )
          xt ra f@ f+ ra f! ;] for ra f@ ;

20 [: ( u1 -- r )
      dup * 1e s>f f/ ;] sum-series f.
```

Sum-series alternatives

```
: sum-series ( ... u xt -- ... r )
    \ xt ( ... u1 -- ... r1 )
    0e {: f^ ra :}
    ra [{: xt: xt ra :}l ( ... u1 -- ... )
          xt ra f@ f+ ra f! ;] for ra f@ ;
```

Stack underground instead of assignment conversion:

```
: sum-series ( ... u xt -- ... r )
  \ xt ( ... u1 -- ... r1 )
  0e [{: xt: xt :}l ( ... r1 u1 -- ... r2 )
        {: f: r :} xt r f+ ;] for ;
```

Stack underground throughout:

```
: sum-series ( ... u xt -- ... r )
  \ xt ( ... u1 -- ... r1 )
  0e swap [: ( ... xt r1 u1 -- ... xt r2 )
    {: f: r :} swap dup >r execute r> r f+
  ;] for drop ;
```

Closure conversion: `testr`

```
testr[x,p,f,u] <-
  if p[x] then f[x]
  else if atom[x] then u[]
  else testr[cdr[x],p,f,
    lambda:testr[car[x],p,f,u]].
```

```
: testr {: x p f u -- s :} recursive
  x p execute if x f execute exit then
  x atom if u execute exit then
  x cdr p f
  x p f u [{: x p f u :}l
    x car p f u testr ;] testr ;

\ Alternative:
: testr1 {: x p -- s1 f :} recursive
  x p execute if x true exit then
  x atom if nil false exit then
  x cdr p testr1 dup if exit then
  x car p testr1 ;

: testr {: x p xt: f xt: u -- s :}
  x p testr1 if f exit then
  drop u ;
```

Closure and assignment conversion: Man or boy?

```
begin
  real procedure A(k, x1, x2, x3, x4, x5);
  value k; integer k;
  real x1, x2, x3, x4, x5;
  begin
    real procedure B;
    begin k := k - 1;
      B := A := A(k, B, x1, x2, x3, x4)
    end;
    if k <= 0 then A := x4 + x5 else B
  end;
  outreal(A(10, 1, -1, -1, 1, 0))
end;
```

```
: A {: w^ k x1 x2 x3 xt: x4 xt: x5 | w^ B :}
  recursive
  k @ 0<= IF  x4 x5 f+  ELSE
    B k x1 x2 x3 action-of x4
    [{: B k x1 x2 x3 x4 :}l
      -1 k +!
      k @ B @ x1 x2 x3 x4 A ;] dup B !
      execute  THEN ;
10 [: 1e ;] [: -1e ;] 2dup swap [: 0e ;] A f.
```

Research questions

- **RQ1** How to implement access to outer locals?
  How to combine locals with quotations, **`postpone`**?

- **RQ2** Does this feature provide a significant benefit?

Research questions

- **RQ1** How to ~~implement~~ **replace** access to outer locals?
  How to combine locals with ~~quotations,~~ **`postpone`**?

- **RQ2** Does this feature provide a significant benefit?

From lexical scoping to our closures and beyond

```
: bar {: x -- xt1 xt2 :}
  [: x ;] [: to x ;] ;
```

⇒ (assignment conversion)

```
: bar {: w^ x -- xt1 xt2 :}
  [: x @ ;] [: x ! ;] ;
```

⇒ (closure conversion and explicit memory manangement)

```
: bar ( x -- xt1 xt2 )
  <{: w^ x :}d x ;> {: x :}
  x [{: x }:d x @ ;] x [{: x }:d x ! ;] ;
```

⇒ (stack closures)

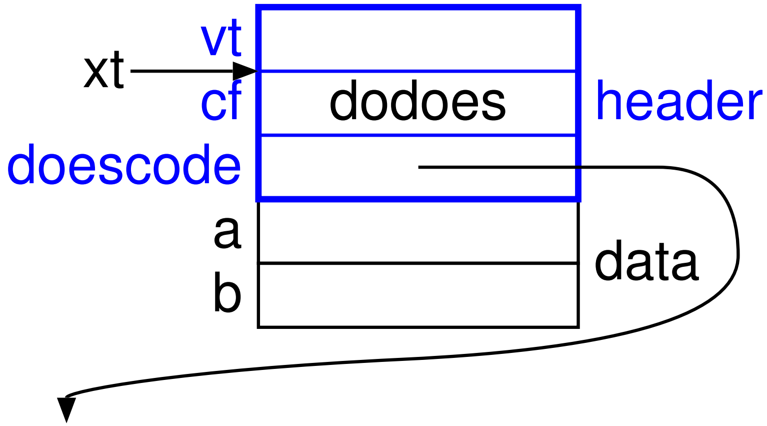
```
: bar ( x -- xt1 xt2 )
  <{: w^ x :}d x ;> {: x :}
  x 1 0 [:d {: x :} x @ ;] x 1 0 [:d {: x :} x ! ;] ;
```

⇒ (eliminate locals)

```
: bar ( x -- xt1 xt2 )
  align here swap ,
  dup 1 0 [:d @ ;] 1 0 [:d ! ;] ;
```

Implementation

: foo [{: a b :}d **a . b .** ;] ;



2@ swap >L >L
a . b . lp+2 ;s

Copy locals from closure to the locals stack

78 source lines for closures

6 source lines for home locations

25 source lines for `postpone` locals

Performance

cycles	instructions	per iteration
21.0	99.0	create `[{: x :}l x + ;]`
62.9	183.5	create `[{: x :}d x + ;]`
113.6	459.0	create and free `[{: x :}h x + ;]`
735.1	2464.7	create `noname create , [: @ + ;] set-does>`
5115.4	15159.5	create `>r :noname r> ]] literal + ; [[`
8.0	14.0	create `[: over + ;]`
7.0	43.0	run `[{: x :}l x + ;]`
21.3	85.0	run `[{: x y z :}l x + ;]`
6.0	38.0	run `noname create , [: @ + ;] set-does>`
6.2	27.0	run `>r :noname r> ]] literal + ; [[`
7.1	33.0	run `[: over + ;]`

Conclusion

- Closures allow passing data to xts executed elsewhere
- Closures are memory-managed explicitly
- Emulate lexical scoping with manual closure conversion and assignment conversion for writable locals (RQ1)
- Pure concept: Stack closure
- There are alternatives (RQ2)
- Implementation simple
- Performance competetive

```
: +field ( u1 u "name" -- u2 )
   create over , +
does> ( addr1 -- addr2 )
   @ + ;
: +field ( u1 u "name" -- u2 )
  create over , +
  here cell- 1 cells const-data
does> ( addr1 -- addr2 )
  @ + ;
: +field ( u1 u "name" -- u2 )
  create over , +
  [: @ + ;] set-does> ;
: +field ( u1 u "name" -- u2 )
  create over
  [{: u1 :}d drop u1 + ;] set-does>
  + ;
: +field ( u1 u "name" -- u2 )
  create over
  1 0 [:d nip + ;] set-does>
  + ;
```

```
: +field ( u1 u "name" -- u2 )
  over + swap ( u2 u1 )
1 0 const-does> ( addr1 -- addr2 )
  ( addr1 u1 ) + ;
: +field ( u1 u "name" -- u2 )
  over >r : r> ]] literal + ; [[ + ;
: +field {: u1 u -- u2 :}
  : ]] u1 + ; [[ u1 u + ;
: +field ( u1 u "name" -- u2 )
  create over , +
  [: @ + ;] set-does>
  [: >body @ ]] literal + [[ ;]
  set-optimizer ;
: +field ( u1 u "name" -- u2 )
   create
   over [{: u1 :}d drop u1 + ;] set-does>
   over [{: u1 :}d drop ]] u1 + [[ ;]
   set-optimizer
   + ;
```