#420

Bruce Anderson

MULTIPLE ENVIRONMENTS FOR ECL 2/25/74

C.J. Prenner

1. New forms and primitives

a) emarked forms

QL(name, mode) $\overset{E}{<<}$ form

- the form must evaluate to an object of type mode
- the form is named by name

b) marked exprs

MEXPR [name, mode] (arglist; result-type) body;

- the value of the procedure must be of type mode
- the frame for the procedure is named name
- the name "goes on" before the actuals to the procedure are evaluated. (necessary for consistency in block-model)

c) environ (pos) - returns an ed initialized to pos

d) setenv (ed, pos) - sets ed to pos

e) enveval (form, apos, cpos) eval form in access env apos and then return to cpos

} as per B-W

pos = { NIL, ed, (ed) } as per B-W

<name,±n> - name refers to names introduced by emarked forms or mexprs

↑ nth most recent n > 0 → control env

n < 0 → access env

f) mkframe (a,c,e,b) - use extension e and params from b evaluate in a + then return to c.

2. Evaluation of new forms and primitives

a) emarked forms-QL(name, mode) $\overset{E}{<<}$ form

1. The name + mode are evaluated
2. An e-marked closure is put on the control stack (see closures)
3. A null proc frame is put on the cs contiguous with 2.
4. The form is evaluated

call to a
b) marked expr – MEXPR[name, mode] (arg list; result-type) body

1. The name + mode are evaluated
2. An emarked closure is put on the control stack
3. The procedure frame is put on the control stack (contiguous with 2)
4. Control is transferred to apply to eval args, etc.

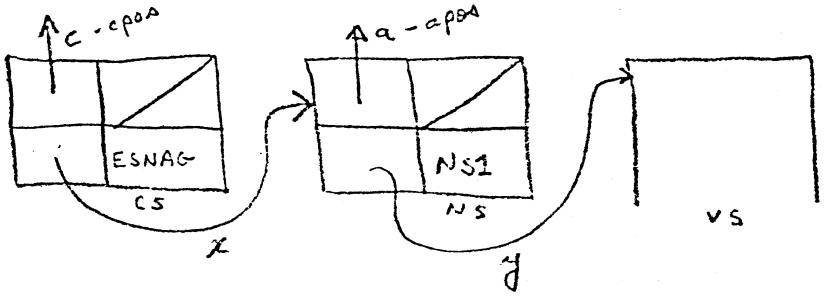
Note - for a normal procedure call a "normal" closure (see closures) is put on the control stack ("unentered" state) and the proc frame is put on after it (contiguous). Just before the proc body is entered the closure is put in the "entered" state. Note also - emarked closures are always in the entered state (i.e. for mexprs, the name "goes on" before the args are evaluated.)

c) environ - as in B-W

d) setenv - as in B-W

e) envenal (form, apos, cpos)

- given apos, cpos construct



and then evaluate the form

Note x,y are used to determine "top" of frame extension for copy.

Note - add one to use counts of a and c (c only if a=c)

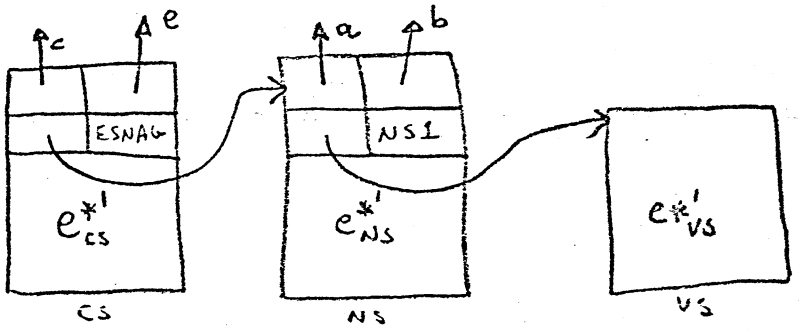
f) Mkframe (This may only be clear on second reading)

a,c,e,b

e = extension to use

b = extension whose parameters are to be used

1. Construct an enveval type extension, using copy'(e*)



2. Parameters to e and b must agree in number and mode, if so connect parameters in $e^{*'}_{ns}$ to values referenced from b^{*}_{ns}

3. The hidden parameters in $e^{*'}_{ns}$ (i.e. free vars bound by the compiler) must be updated for a. Get these from a list in e and eval in a, check for mode agreement with bindings in e*' and replace.

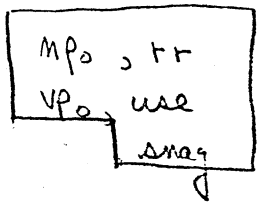
   Note - e* and b* must be "frozen" since $e^{*'}_{ns}$ has hard references to declared variables in e^{*}_{vs} and parameters in b^{*}_{vs}.

4. Add 1 To ale counts of e, b, a, c (if c ≠ a)

Closures and procedure frames

Normal closure - put on for normal procedure entry

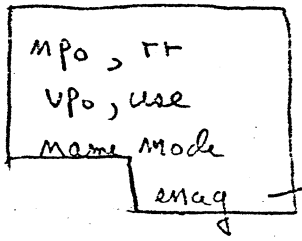
A



uncertain or entered

e-marked closure - put on for e-mark or Mexpr

B

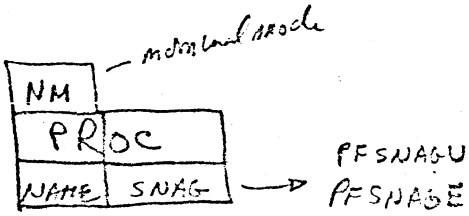


always ECLOE for "entered"

NoTe -
MPo = value of mp
VPo = value of vp
rr = result return
} when closure is made

procframe

C



normal Call - A + C (5 wds) } snags are changed To "entered" when proc is entered

Mexpr - B + C (6 wds) } snags for CLs cleared on procedure entry

e-mark - { B + C (with null fields and PFSNAGE) (6 wds)
{ and Then the form is evaluated

NT - (A,C) or (B,C) pairs are always contiguous

3. Duplication of Frame Extensions

Notation - f* = frame extension

f^*_{ns}, f^*_{vs}, f^*_{cs} - components of f* on ns, vs, cs

Note - f^*_{ns} contains parameters of basic frame as well as temps. since we want the compiler to reference all nomenclature off of np, and therefore, everything must be contiguous on the ns.

There are three cases in which duplication of a f* is required (a) stack overflow (in which case a copy is made and the original f* is deleted), (b) an attempt is made to return to a f* (use=1) and there is insufficient room beneath it on the stacks in which to run (in which case a copy is made and the original is deleted), (c) an attempt is made to return to a f* with use >1 (a copy must be made + the original retained). There are two types of copy -

(1) copy' - used for (c) above. Let f^*_{cs}', f^*_{ns}', f^*_{vs}' be the copies of f^*_{cs} f^*_{ns}, f^*_{vs}, respectively

f^*_{vs}' - only cells corresponding to temps or unnamed objects referenced from f^*_{ns} are copied

f^*_{ns}' - unnamed entries reference copies in f^*_{vs}' or heap, named entries reference objects in f^*_{vs}, heap, or other frames

f^*_{cs}' - saved stack pointers reference f^*_{ns}', f^*_{vs}', or f^*_{vs}, as appropriate

(2) copy" - used for (b) and (c) above. Let f^*_{cs}", f^*_{ns}", f^*_{vs}" be the copies of f^*_{cs}, f^*_{ns}, f^*_{vs}, respectively

f^*_{vs}" - copied in its entirety

f^*_{ns} - references to f^*_{vs} are updated to reference f^*_{vs}"

f^*_{cs} - references to f^*_{ns}, f^*_{vs} are updated to reference f^*_{ns}", f^*_{vs}"

For each of the two types of copy, there are three cases to consider.

1. copy a f* which is contiguous with its basic frame
2. copy a f* which is remote, i.e. blocked to its basic frame
3. copy a f* which has been created by enveval or mkframe

Note - In order to determine the case, the cs is scanned and the first of snags

PFSNAGE (case 1)
BLINK (case 2)
ESNAG (case 3)

seen determines the case.

For copy', f*' has "hard" references to f*, namely, name stack entries which reference objects in f^*_{vs}. Thus, once f*' is created, f* may not execute (until f*' is destroyed).*

Hence in each of the three cases, an additional link the "sib-link" is used to connect f'to f* , and the use count for f* is incremented to insure that no execution in f* will take place. As it turns out, with this method CXT counts are not required since f* will stay around until f*' is destroyed.

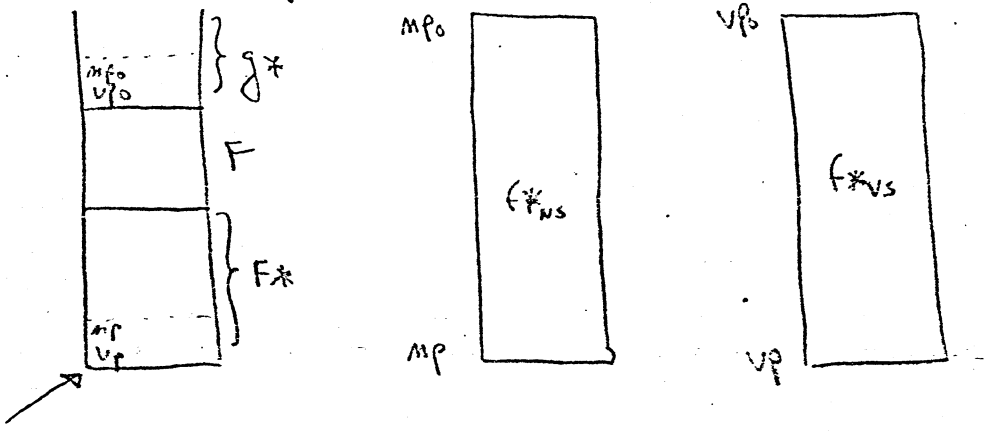
The diagrams on the following pages describe the 2 types of copies for each of the three cases. Note that f^*_{ns} contains the formals of the basic frame as well as declared variables and temps.

For example, if f is inside a block, then f*' will have references to the declared variables in f^*_{vs}. If f* executes and exits the block, then the objects referenced from f*' will be destroyed, i.e. dangling references. This seems to be a result of our unwillingness to construct frames for blocks + our desire to allow the compiler to assume contiguity of ns entries.

To copy f* (due to use>1 or overflow or no room)

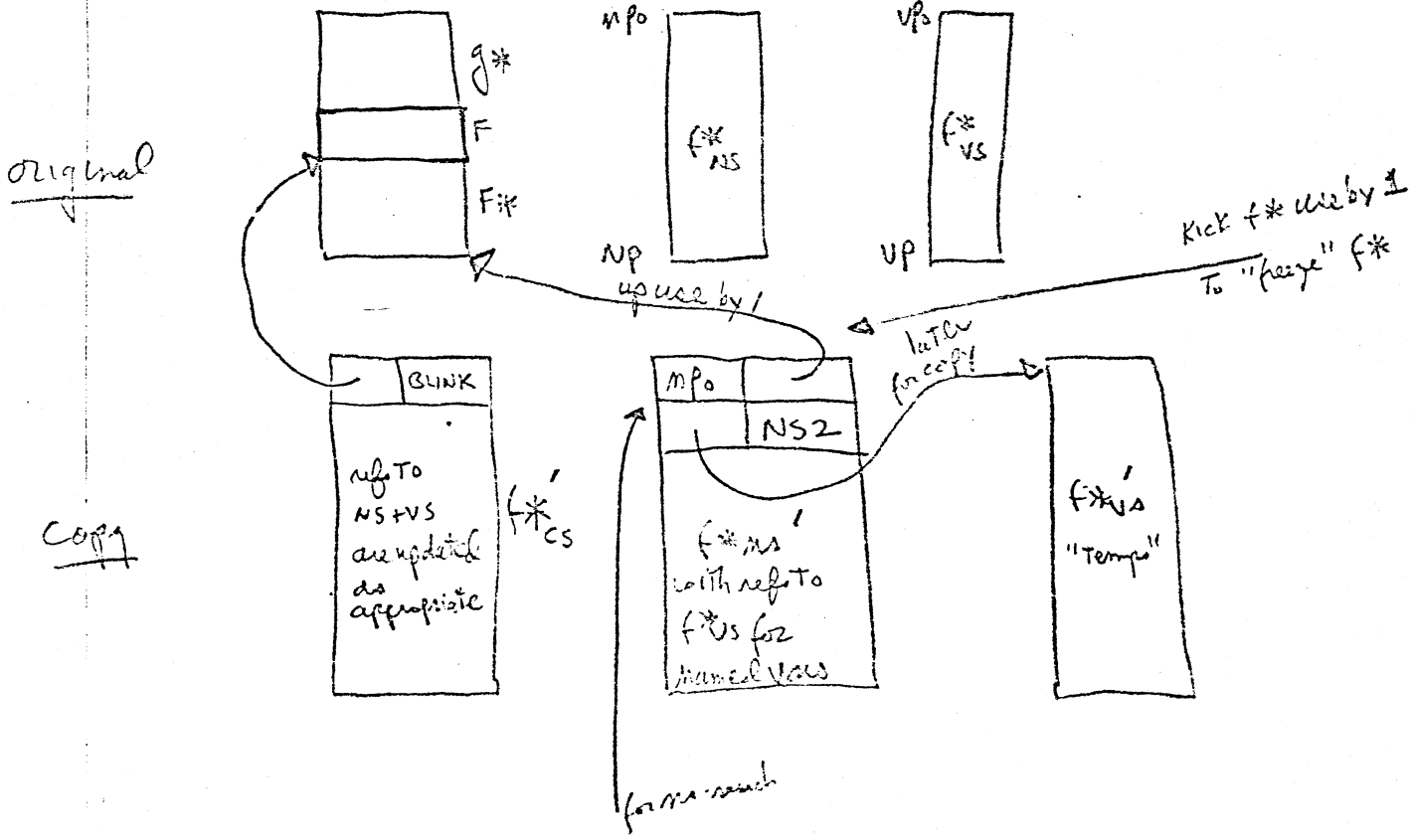
NT - if overflow - close f* with use of 1 to simplify processing
(or simply consider as being closed)

Case I - contiguous frames

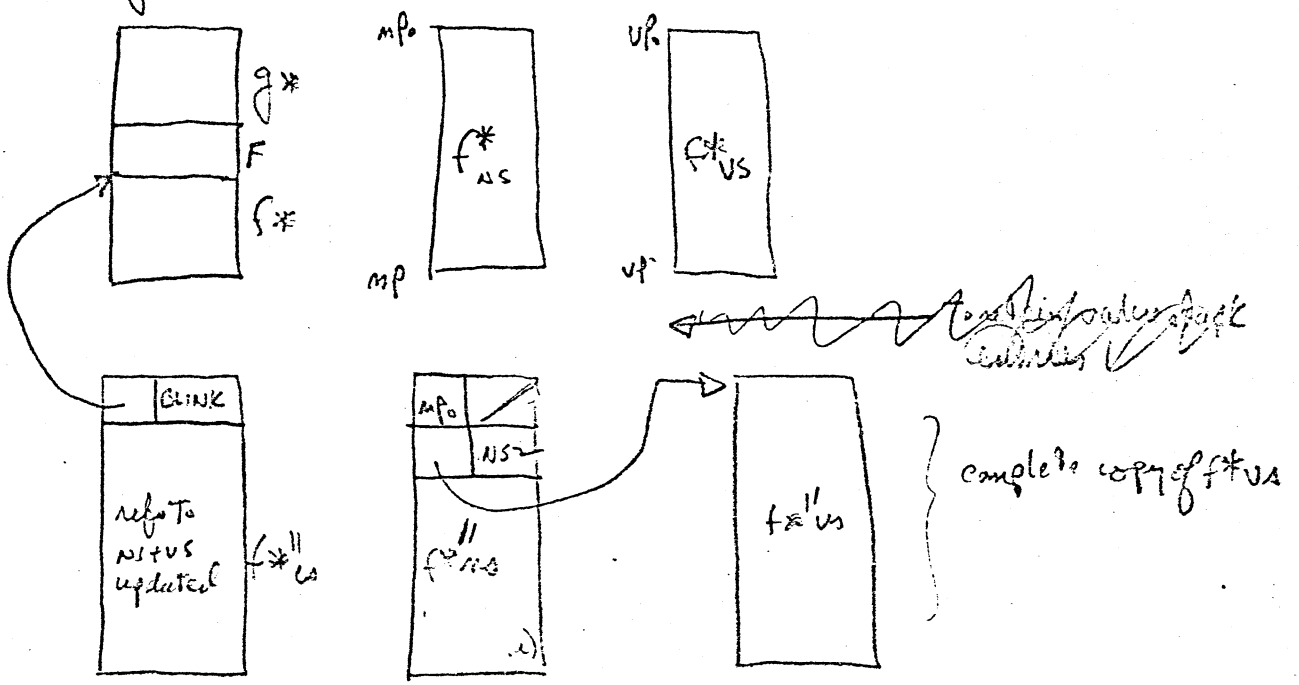


Wish to copy f* (+perhaps delete original)

(a) original is to be retained



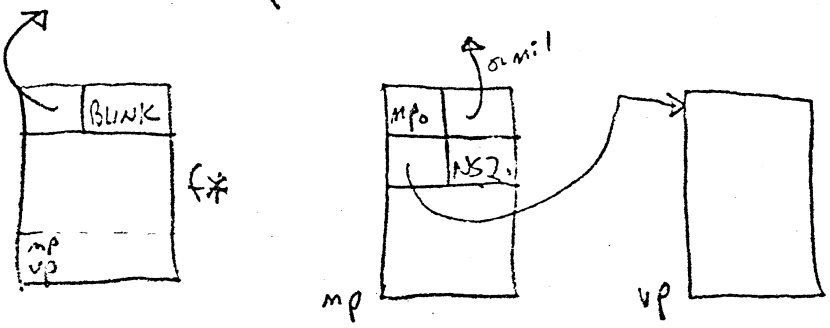
(b) original vs To be deleted



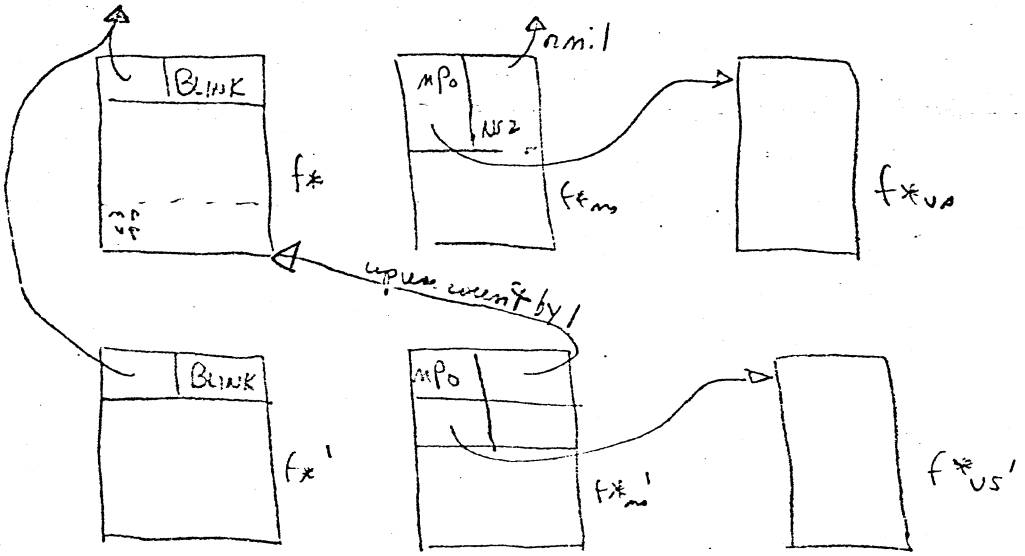
NT - f^*_{NS}, f^*_{VS} and f^*_{CS} can be deleted

~~NT alternatively f^*_{VS} should be copied~~ ~~and all 3 f^* segments deleted~~ ~~f^*_{NS} and f^*_{CS} deleted, mallfield was updated to up~~

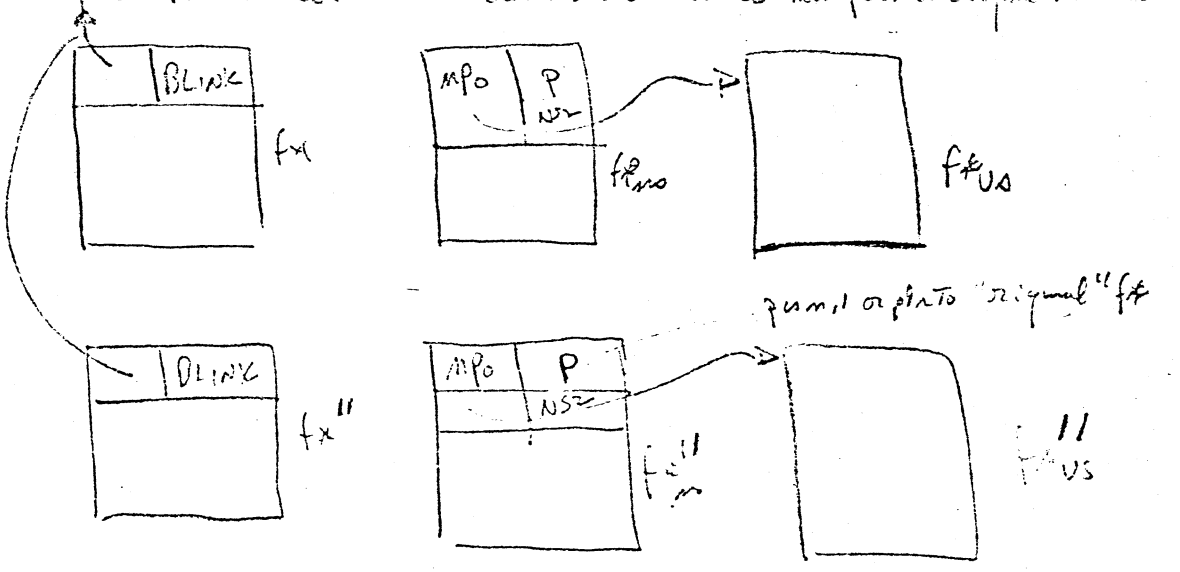
Case II remote frame



(a) original To be relocated

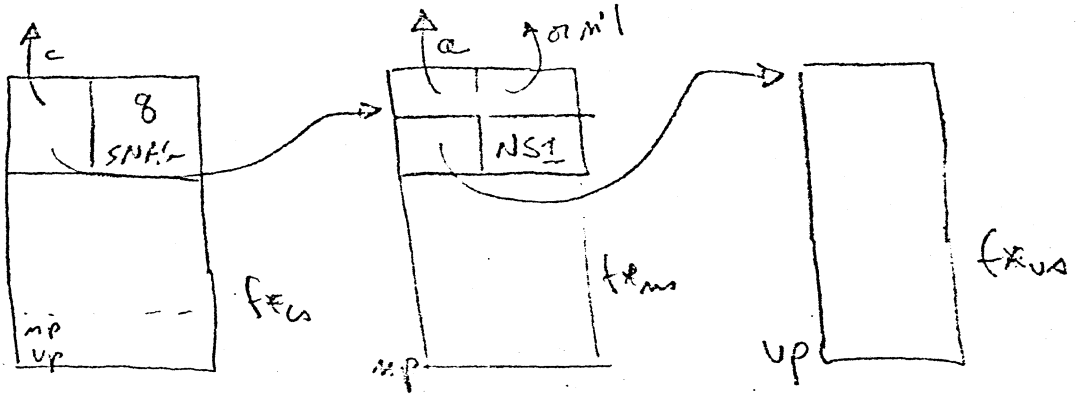


(b) original To be deleted - ~~conflict since multiple copies exist~~



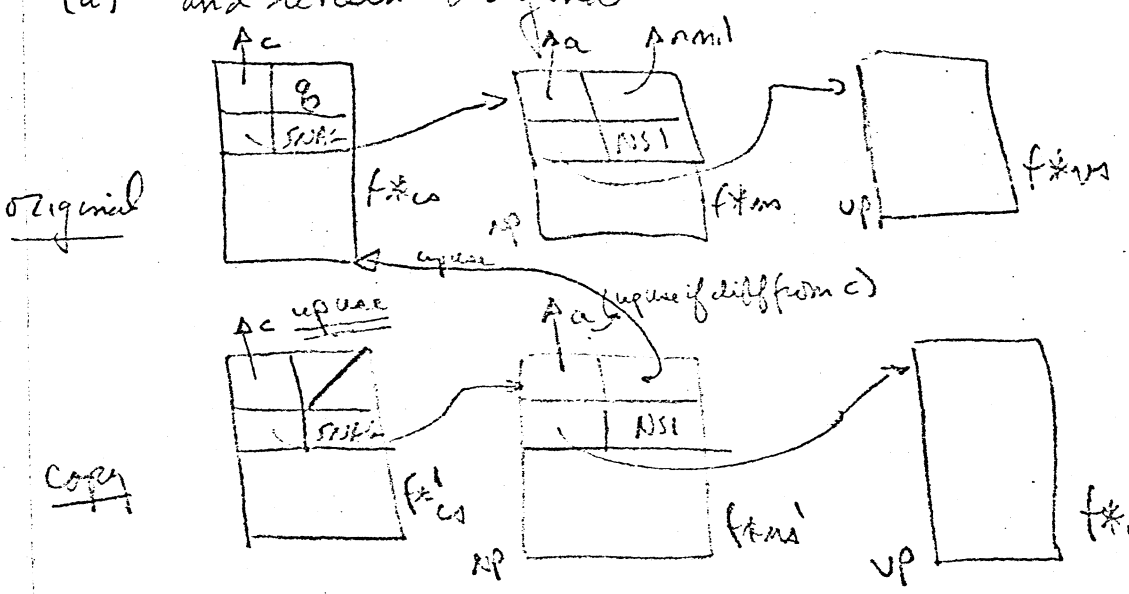
delete fx_{VS}, fx_{mp} fx

Case III - ENVEVAL + MKFRAME

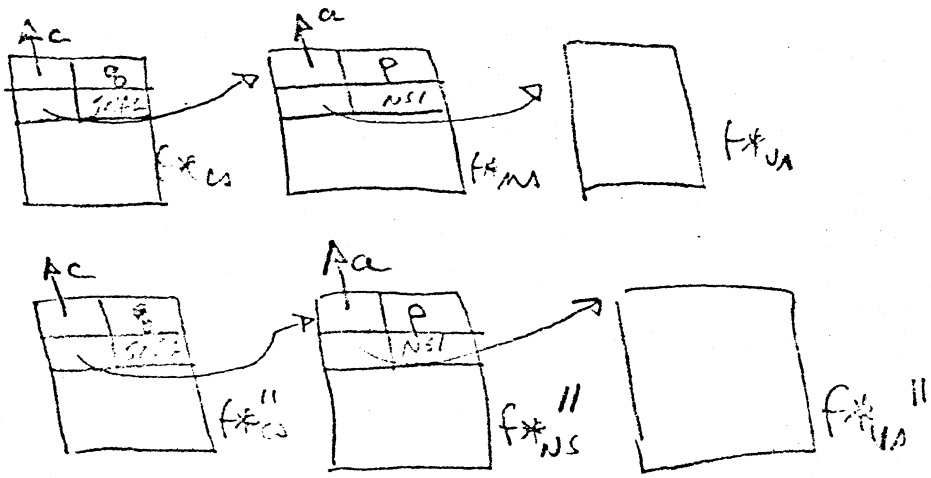


Wish To copy

(a) and retain original



(b) Delete original



and delete f*, f*ns, f*vs

4. Return from frames - execution of snag words (see closures)

1) PFSNAGE - "entered basic frame snag"

check result type against result; pop basic frame off of stack

2) ECLOE - "entered marked frame snag" (let f* be the frame extension)

- if use > 1 then use ← - 1 and copy' (f*) + execute in copy.

- check result against saved mode

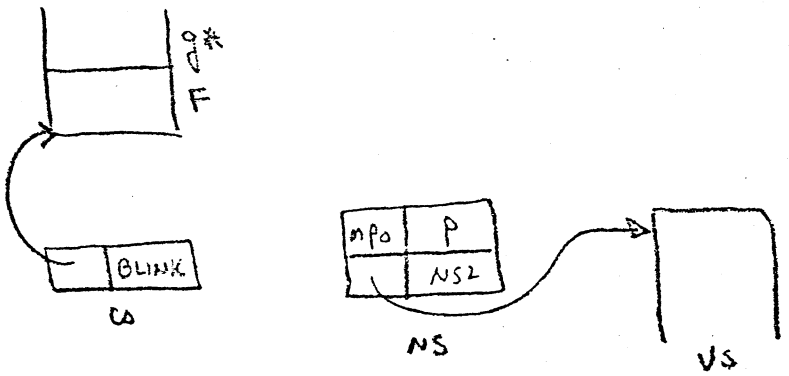
- restore stack ptrs + execute real return

3) NCLOE - "normal closure"

same as ECLOE without check on data type of result

4) BLINK - Return from remote extension

For remote f*, if the "sib-link" is nil, then no sibling is being retained in order to provide a partial environment for f*, and thus the basic frame can be deleted. If sib-link #NIL, the sib-link must be released (using release frame). However, the use count of the predecessor f* must be incremented in order to avoid its being deleted by release frame.



p = NIL => delete F; go to FOO

upuse(g*) - so it won't be deleted

releaseframe(p) - no longer need vs of sib-link

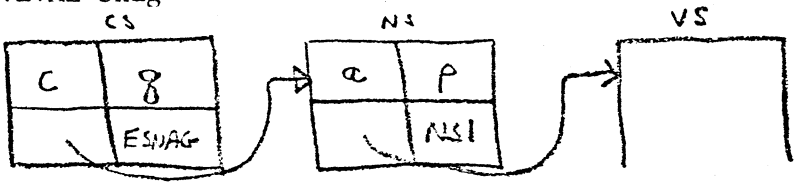
FOO: g*.use > 1 => g*.use ← - 1; copy'(g*) + run there

if room to run below g* => run there

copy"(g*) + delete original + run in copy

[NT → cs + ns info for remote f* is flushed]

5) ENVEVAL snag



p#NIL → releaseframe(p)

q#NIL → releaseframe(q)

c#a → releaseframe(a)

let g* be the extension referenced by c

goto FOO (in BLINK code)

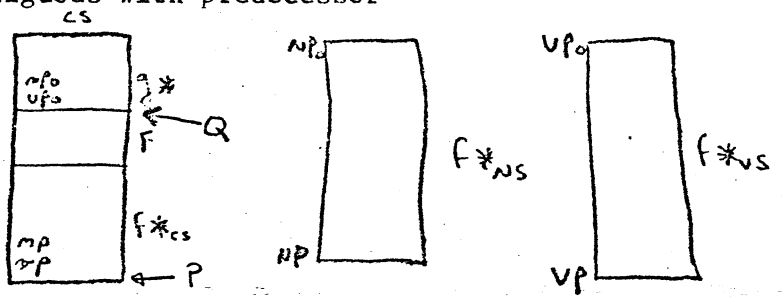
5. Release Frame

releaseframe(p) =

p references a f*

[1] P.use > 1 => P.use ←— 1 + return

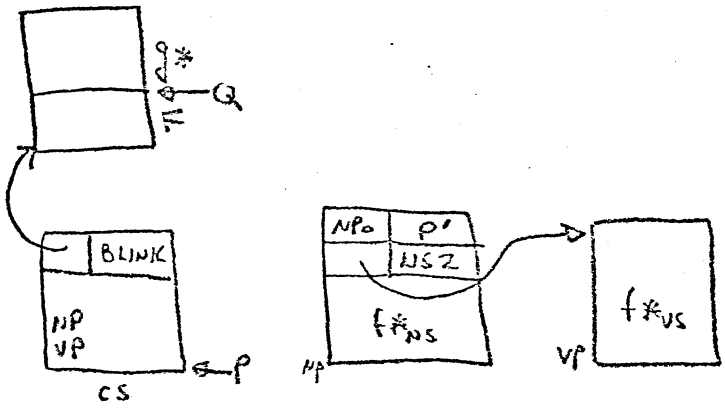
[2] if f* is contiguous with predecessor



then delete F, $f*_{cs}$, $f*_{ns}$, $f*_{vs}$

and then releaseframe(Q)

[3] if f* is remote

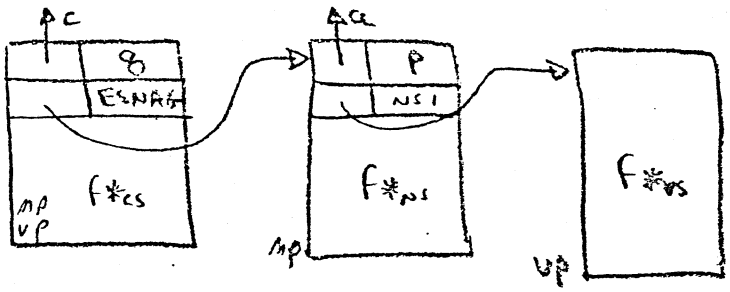


p' = NIL => delete F, $f*_{cs}$, $f*_{vs}$, $f*_{ns}$ and then releaseframe(Q)

p'≠NIL => delete $f*_{cs}$, $f*_{vs}$, $f*_{ns}$ and then releaseframe(p')

Note - F will be deleted when either an f* contiguous with F is deleted, or when a remote f* with p=NIL is deleted.

[4] enveval f*



delete f^*_{cs}, f^*_{ns}, f^*_{cs}

releaseframe(c)

a#c => releaseframe(a)

p#NIL => releaseframe(p)

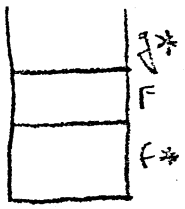
q#NIL => releaseframe(q)

6. Counting for pos

given <name, ±n>

- close current f*

- consider predecessor f* - say g*; k ← 0



FUM: g* has emarked closure => compare names

if =, and k = |n| then g* is desired extension

if =, and k # n then k ← k+1 + goto FOO

if names are # then goto FOO

FOO: g* is either contiguous, remote or envevaled extension

contiguous or remote → set g* to be predecessor extension + goto FUM

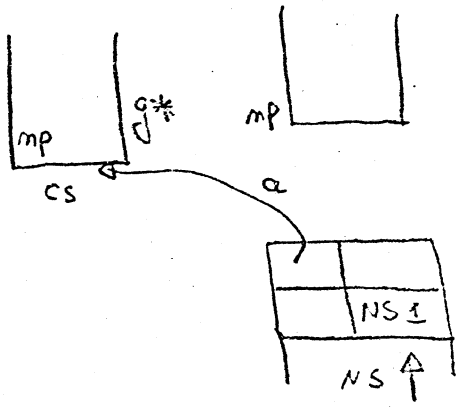
envevaled extension =>

set g* to be extension referenced by c or a as n is greater than 0 or less than 0; goto FUM

7. Name Stack search

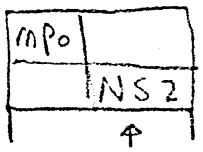
(1) search NS until NS1 or NS2 snag is seen

(2) NS1 - envend or mkframe extension



from a gel mp, continue search at mp

(3) NS2 - remote extension



Continue search at mp_0

8. Implementation Issues

(1) copy" - the entire f* is copied and the original is destroyed. Two other solutions are possible if less copying is desired.

(a) use copy' and retain original f*. This seems to be undesirable since no space is freed.

(b) use copy' and retain only f^*_{vs} (i.e. delete f^*_{ns}, f^*_{cs}). This is better than (a), but I don't know how to do it. Also, since f^*_{vs} is not deleted, chances are that one won't be able to utilize the space occupied by f^*_{ns}, f^*_{cs}, i.e. there will exist holes on the control + name stack, but no holes on the value stack.

(2) copy' - I am not sure that we all understand exactly which saved stack pointers should reference f^*_{vs} versus f^*_{vs}'. Wegbreit has suggested that any stack pointers which reference (f*vs corresponding to named objects remain unchanged. In addition, to perform fast copy' a free space is required in order to determine relocation information for each cell in the f^*_{vs} segment.

(3) saved stack pointers. A saved full work stack pointer cannot be simply put into a stack register, since the extent of the stack may change after the stack pointer is saved. Bud has suggested the following

(1) To restore full word stack ptr (eg. vp)

```
POP CP, VP
HRRI VP, @ IVPC
```

i.e. always use I?PC

(2) compiled code - only the RH of the stack ptr is restored. Thus the stacks will appear to be smaller than their actual length. The stack overflow routine will detect false overflows and fix the LH appropriately.

(4) Stack compactification and copy" can occur anytime a push is executed, therefore, any registers which reference the current f* when the push is executed will have to be updated. Hence, a push UUO or subroutine may be required in certain instances. Someone must examine the system code to determine these cases.

(5) scanning the cs - A f^*_{cs} may contain records for blocks, un-entered f*s, iterations, etc. When copied, these must be updated to point to appropriate cells in the copied segments of the name + value stacks.

(6) Shallow bindings

1) Either BUD's scheme (atom points to must recent binding which points back to the atom/validity check is made when name is accessed) or a linked-ns scheme will work.

2) In either scheme, ACD's will work or "update all names for current environment" will work.

3) ACD's can be the value of the np at the time the environment is created.

4) In all of the above, it is necessary that ns entries for stack "holes" be nulled.

(7) mkframe - problem of detecting saved references to original free vars after extension is copied and associated with a new set of free vars.