Published: 03/07/67

Identification

"Synthetic epilogue" procedure for EPL
synep_
D. B. Wagner, M. D. McIlroy and D. L. Boyd

Purpose

There is never a perfect guarantee that a call will receive a return; a block may be terminated rather abruptly because of a non-local go to from a dynamic descendant to a dynamic ancestor. Some blocks, however, may not be terminated without being given a chance to "de-initialize." Utter chaos will result, for example, if a block is terminated without the proper reversion of an on-unit established in the block.

For this reason all EPL blocks which require epilogues cooperate in building the epilogue chain described in BN.5.01. The prodedure synep_ is used to do all non-local go to's.

Usage

The call to synep_ may be made explicitly for some reason, but normally it is called where needed by the compiled code. Synep_ may be called for any of the following reasons:

a non-local go to
a go to to a label variable which could be non-local

Synep_ is called using the form

```
call synep_(v);
```

where the parameter expected is declared

```
dcl  v label;
```

Synep_ takes the label variable passed to it, which includes a location and a stack level, and carefully terminates each block between itself and the target block. For each of these blocks which has an epilogue listed in the epilogue chain, the epilogue is executed. Finally synep_ transfers to the location specified.

Implementation

See BN.5.01 for details of the epilogue chain and epilogue handlers. An epilogue handler is a structure containing a label and a pointer to the previous epilogue handler. The label gives the address of an epilogue and the stack pointer for the block it corresponds to.

To straighten out terminology: up in the stack means toward higher address locations, i.e. toward deeper nestings. Down is toward shallower nestings.

Synep_ threads down the stack from its own stack frame looking for blocks with epilogues. Since the epilogue chain has the same order as the stack, it is only necessary to compare each stack frame address with the stack pointer in the current epilogue handler. When an epilogue is to be performed, synep_ fiddles with stack pointers as described later and transfers to the epilogue. The epilogue ends with a return sequence which brings it back to synep_ (strangely enough, since it thinks it is returning to the block's caller).

The code which effects the transfer to the epilogue is better seen than believed. A version of it is reproduced here:

```
doepi:   eapbp     p,*    (p contains stack pointer for
                           block being terminated)
         ldaq      sp|18  (fiddle with "next sp" and
         staq      bp|18  "previous sp" in
         stpsp     bp|16  block being terminated)

         ldaq      epi_,* (Put location of epilogue
         staq      bp|8     into a place the condemned
                            block won't be needing
                            again)

         stb       sp|0
         stcd      sp|20
         tra       x
         tra       finished
x:       ldb       bp|0   (restore epilogue's
         tra       sp|8,* environment and go to it)
```

The effect of this code is that while the epilogue is being performed it has all of its base registers as they should be. "Next sp" in its stack frame points above the frame for synep_, and "previous sp" points to the frame for synep_. Thus the epilogue may freely call out, cause linkage faults, etc., but when it returns it ends

up back in synep_, which can then continue its work.

Interruptions in Epiloques

An epilogue should be coded to avoid any synchronous interrupts which could cause it trouble (such as overflows in an epilogue which is charged with reverting the overflow condition). However asynchronous interrupts can still cause trouble. For this reason asynchronous on conditions are not supported.

The following routine purports to do an iteration until time runs out and then return a value. However it returns successfully in spite of overflow if time runs out while processing the overflow trap:

```
iterate:  proc (fail) float;
          /*set timer*/
          on timer go to done;
          on overflow go to fail;
          do i = 1 by 1;
          /*iteration to calculate y*/
          end;
done:     return(y);
          end;
```

What would give trouble in the example above is an overflow fault followed almost immediately by a timer interrupt. The sequence of events is as follows: an overflow occurs, so the on-unit

```
go to fail;
```

is executed. This is a non-local go to, so it is executed using synep_. Synep_ invokes the epilogue for the iterate block, one of whose duties is to revert the on-unit for the timer condition. The timer condition occurs, however, before the epilogue has completed this reversion, so that the on-unit

```
go to done;
```

is executed immediately. This go to is local, but the problem would still exist if it were non-local. Control passes to the statement

```
return(y);
```

which invokes the epilogue again and returns a value of y which, because of the overflow, is probably worthless.

There seems to be nothing that can be done by EPL or PL/I to alleviate this situation. The above example is best considered bad programming.