Eliminating Recursion (Optional)

Tail Recursion Optimization (optional)

"TAIL RECURSION n. See TAIL RECURSION." — The Jargon File.

Suppose we have a routine of the following form

procedure $f(p)$
    var v
    if e then
        S
        $f(A)$
    else
        T
    end if
end f

Note that the recursive call is the last thing done before returning.

Such a call is called a tail call.

I'll assume two stacks are used by the machine code:

- A variable stack containing "stack frames." Stack frames contain local variables including parameters.
- A return address stack containing return addresses.

So a function f is compiled to the following

f :

push a new stack frame to hold p and v
copy $r1$ to p's location in the stack frame
Body of Function

A return is compiled to

pop stack frame
pop an address from the return address stack
branch to that address

A call to $f(A)$,assuming the parameter is passed in $r1$, is compiled to

code for A *leaving result in* r
push address b onto the return address stack
branch to f
b :

[On some architectures, return or the last two steps of the call might be represented by a single machine instruction. That doesn't matter. I'll break them into their conceptual steps.]

[On some architectures, the stack frames and return addresses go on a single stack. It is easier to understand the following sequence of transformations if we think about two stacks. In the end, it won't matter.]

So our machine code will look something like the following.

Note that I've distributed the implicit return to each branch of the if.

f :

  push a new stack frame to hold p and v
  copy $r1$ to p's location in the stack frame
  // if e
  evaluate e
  conditional branch to a
    code for S
    // call $f(A)$
      code for A *leaving result in* $r1$
      push address b onto the return address stack
      branch to f
      b :
    // return
      pop stack frame
      pop an address from the return address stack
      branch to that address
  a :
    code for T
    // return
      pop stack frame
      pop an address from the return address stack
      branch to that address

Note that there is no use of the stack frame after the recursive call to f.

So why do we need it?

Save some space by popping the stack frame before branching to f.

f :

push a new stack frame to hold p and v
copy $r1$ to p's location in the stack frame
evaluate e
conditional branch to a
code for S
code for A *leaving result in* $r1$
pop stack frame
push address b onto the return address stack
branch to f
b : ~~pop stack frame~~
pop an address from the return address stack
branch to that address

a :

code for T
pop stack frame
pop an address from the return address stack
branch to that address

But the first thing f will do is push a new stack frame

So why pop and then push? Save time by doing neither.

f :

push a new stack frame to hold p and v

$\underline{g\ :}$

copy $r1$ to p's location in the stack frame

evaluate e

conditional branch to a

code for S

code for A *leaving result in* $r1$

~~pop stack frame~~

push address b onto the return address stack

branch to $\cancel{f}$ $\underline{g}$

b :

pop an address from the return address stack

branch to that address

a :

code for T

pop stack frame

pop an address from the return address stack

branch to that address

Suppose before the recursive call to f, the top of the return address stack is x. The call pushes b.

The return from the recursive call to f will pop address b from the stack and branch to b. The code after b then pops address x from the stack and branches to that.

If we don't push b, the return from the recursive call will pop and branch straight to x. The same end result.

So why push b ?

f :
- push a new stack frame to hold p and v
- g : copy $r1$ to p's location in the stack frame
- evaluate e
- conditional branch to a
  - code for S
  - code for A leaving result in $r1$
  - ~~push address b onto the return address stack~~
  - branch to g
  - ~~b : pop an address from the return address stack~~
  - ~~branch to that address~~
- a :
  - code for T
  - pop stack frame
  - pop an address from the return address stack
  - branch to that address

This is **tail recursion optimization** or, since it can be applied to nonrecursive calls too, **tail call optimization**.

Eliminating recursion altogether (optional)

Back in the days of Fortran and Cobol, programmers were taught to eliminate recursion altogether, as these languages did not support it. I'll illustrate with an example

procedure $f(p)$
  var v
  a: if e then
    S
    $f(x)$
    b: T
    $f(y)$
    c: U
  else
    W
  end if
  rtn:
end f

Becomes

procedure $f(p)$
  type $Label = \{a, b, c, rtn\}$
  var v
  var $vStack$:= new $Stack()$
  var $pStack$:= new $Stack()$
  var $labelStack$:= new $Stack \langle Label \rangle ()$
  var $label := a$
  while $\neg(\text{empty}(labelStack) \wedge label = rtn)$ do
    if $label = a$ then
      if e then

```
                    S
                    push p onto pStack
                    push v onto vStack
                    push b onto labelStack
                    p := x
                    label := a // recursive call
                else
                    W
                    label := rtn
                end if
            elseif label = b then
                T
                push p onto pStack
                push v onto vStack
                push c onto labelStack
                p := y
                label := a // recursive call
            elseif label = c then
                U
                label := rtn
            elseif label = rtn then
                label := pop labelStack
                p := pop pStack
                v := pop vStack
            end if
        end while
    end f
```

[End of optional section.]