

## Basic Blocks and Traces

- procedure fragments -> frame and body
- Translate AST – Tree language – replace language-specific complex constructs with something more general, and closer to machine
  - expressions, including assignments
  - procedure calls, returns...
  - conditions
- The final code is a sequence of instructions: labels, jump, procedure calls, and assignments
- Next step -> linearize the tree

- linearize tree into sequence of statements (not the same Tree.stm) - canonize
- define ‘uninterrupted’ subsets – blocks of statements which will always execute sequentially – basic blocks
- organize blocks so as to form traces of possible executions - traces

```
signature CANON =
sig
  val linearize : Tree.stm -> Tree.stm list

  val basicBlocks : Tree.stm list ->
    (Tree.stm list list * Tree.label)

  val traceSchedule : Tree.stm list list * Tree.label ->
    Tree.stm list
end (* signature CANON *)
```

- in general: Tree exp and stm don’t correspond exactly to machine languages:
  - conditional jumps on real machines have one target
  - unlimited number of registers
  - ‘high’-level constructs like ESEQ and CALL have side-effects -> problems...

## tree Statements and Expressions

- Here is the detail of itree statements **stm** and itree expressions **exp**

```
datatype stm = SEQ of stm * stm
| LABEL of label
| JUMP of exp * label list
| CJUMP of relop * exp * exp * label * label
| MOVE of exp * exp
| EXP of exp
and exp = BINOP of binop * exp * exp
| MEM of exp
| TEMP of Temp.temp
| ESEQ of stm * exp (!!!)
| NAME of label
| CONST of int
| CALL of exp * exp list (!!!)
```

## Side-Effects

- Side-effects means updating the contents of a memory cell or a temporary register.
    - ESEQ(s,e) where s is a list of statements that may contain MOVE statement
- BINOP(op,t1,t2)  
 instructions to compute t1 into ri ;  
 instructions to compute t2 into rj ;  
 rk <- ri op rj
- But it won’t work for this:  
 BINOP(PLUS,TEMP a,ESEQ(MOVE(TEMP a,u),v))
- CALL(e,el) by default puts the result in the return-result register.  
 BINOP(PLUS, CALL(...), CALL(...))

## Canonical trees

- want pure expressions only – why?
- pull stmt out of exp (ESEQ) – to top
- pull calls up to top level
  - sideeffects, register use
  - CALL ok in assignment MOVE (return value) or as a side-effect only call at the top
- exp':
  - no ESEQ, no CALL;
  - parent to CALL cannot be CALL

- stm – linearized into lists of stm'
  - stm' no SEQ, EXP is exp' or call
  - ok to have top call statements or to return call in move statements
- so – pull ESEQ, SEQ to the top and front -> SEQ, ESEQ chain can be simplified with lists of trees
- expressions CALL(...) ->
  - ESEQ(MOVE(TEMP t, CALL(...), TEMP t))

$ESEQ(s1, ESEQ(s2, e)) \implies ESEQ(SEQ(s1, s2), e)$

$BINOP(op, (ESEQ(s, e1), e2)) \implies ESEQ(s, (BINOP(op, e1, e2)))$

$BINOP(op, e1, (ESEQ(s, e2))) \implies ESEQ(MOVE(TEMP tnew, e1), ESEQ(s, (BINOP(op, TEMP tnew, e2)))$

$BINOP(op, e1, (ESEQ(s, e2))) \implies ESEQ(s, BINOP(op, e1, e2))$

(if s and e1 commute (i.e. are noninterfering, the effects performed by s will not change the value computed by e1))

- have a commute function
- for each stm/exp – extract expression list, and provide build functions (exp.list arg)
- check if commute true when reordering
- cannon.sml implements this, you'll need to call it

## Rearranging itree statements

- Goal: rearrange the list of canonical trees so that every CJUMP is immediately followed by its false branch LABEL(If).

*A basic block is a sequence of statements that is always entered at the beginning and exited at the end:*

- the first statement is a LABEL
- the last statement is a JUMP or CJUMP
- there are no other LABELS, JUMPs, or CJUMPs in between
- often used to analyze a program's control flow
- #1: take a list of canonical trees and form them into basic blocks
- #2: re-order the list of basic blocks into traces

## Canonical Trees => Basic Blocks

- Input: a sequence of statements (i.e., canonical trees --- the body of a function); Output: a set of basic blocks

Algorithm:

- if a new LABEL is found, end the current block and start a new block;
- if a JUMP or CJUMP is found, end the current block;
- if it results a block not ending with a JUMP or CJUMP, then a JUMP to the next block's label is appended to the block;
- if it results a block without a LABEL at the beginning, invent a new LABEL and stuck it there;
- invent a new label done for the beginning of the epilogue;
- put JUMP(NAME done) at the end of the last basic block.

## Traces

- Arrange blocks in traces
  - Trace: path in control-flow graph which is a potentially real program execution (cannot determine actual at compile time)
  - Blocks can be arranged in any order, but this is what we want (whenever possible):
    - cond jumps – followed by false state
    - follow uncond JUMP by target

```
initially all blocks in Q
while Q not empty
  Start a new (empty) trace T
  Remove b = head of Q
  while b not marked
    Mark b; append b to T
    Examine blocks to which b branches
    if there is an unmarked successor c
      b <- c
  End current trace b
```

## Traces => List of Statements

- Flatten the traces back to an ordered list of statements (canonical trees): - better for instruction selection phase
  - any CJUMP followed by its false label: do nothing ;
  - any CJUMP followed by its true label: switch its true and false label, and negate the condition;
  - remove JUMP(l) if it is followed by its target l ;
  - JUMPs to JUMPs;
  - any CJUMP(op,e1,e2,l,t,lf) followed by neither label: invent a new false label ln , rewrite it into :  
CJUMP(cond, lt, ln)  
LABEL ln  
JUMP(NAME lf)

## Optimal traces

```
<prolog-stmts>
LABEL test
CJUMP (i>n, epilg, body)
LABEL body
<body-stmts>
LABEL done
<epilog-stmts>
```

## Summary: IR -> Machine Code

- #1 : Transform the tree into a list of canonical trees
  - a. eliminate SEQ and ESEQ nodes
  - b. the arguments of a CALL node should never be other CALL nodes
- #2 : Perform various code optimizations on canonical trees
- #3 : Rearrange the canonical trees (into traces) so that every CJUMP(op, e1, e2,lt,lf) is immediately followed by LABEL(lf).
- #4 : Instruction Selection ---- generate the pseudo-assembly code from the canonical trees in the step #3.
- #5 : Perform register allocations on pseudo-assembly code