Basic Blocks and Traces

Mismatches between IR and assembly language
Canonical trees
Basic blocks
Traces
Our next objective is to translate IR trees into assembly language.

The operations of the Tree language are carefully chosen to match the capabilities of most machines.

However, some aspects of Tree language do not correspond exactly.

Example: It's useful to evaluate subexpressions in any order, but Tree.ESEQ can have side effects.
IR to Assembly Lang Mismatches

- **CJUMP** has true and false labels, but real conditional jump instructions fall through if condition is false.
- **ESEQ** nodes within expressions make it possible for different orders of evaluating subtrees to yield different results.
- **CALL** nodes within argument expressions of other **CALL** nodes cause problems.
Transformation of IR Trees

- We can take any IR tree and transform it into an equivalent tree without these mismatches.

Transformation stages:

1. Rewrite tree into list of canonical trees without 
   \texttt{SEQ} or \texttt{ESEQ} nodes.
2. Group list into set of basic blocks, which contain no internal jumps or labels.
3. Order basic blocks into set of traces in which every \texttt{CJUMP} is immediately followed by its false label.
Canonical Trees

Canonical trees have the following two properties:

1. They contain no \texttt{SEQ} or \texttt{ESEQ} nodes.
2. The parent of each \texttt{CALL} node is either \texttt{EXP(...)} or \texttt{MOVE(TMP(t), ...)}.

We start by enforcing property 1, with the strategy being to lift \texttt{ESEQ} nodes higher and higher in the tree (until they become \texttt{SEQ} nodes).
ESEQ Identities (1)

1. \( \text{ESEQ}(s_1, \text{ESEQ}(s_2, e)) = \text{ESEQ}(\text{SEQ}(s_1, s_2), e) \)
2. \( \text{BINOP}(\text{op}, \text{ESEQ}(s, e_1), e_2) = \text{ESEQ}(s, \text{BINOP}(\text{op}, e_1, e_2)) \)
   - \( \text{MEM}(\text{ESEQ}(s, e)) = \text{ESEQ}(s, \text{MEM}(e)) \)
   - \( \text{JUMP}(\text{ESEQ}(s, e)) = \text{SEQ}(s, \text{JUMP}(e)) \)
   - \( \text{CJUMP}(\text{op}, \text{ESEQ}(s, e_1), e_2, l_1, l_2) = \text{SEQ}(s, \text{CJUMP}(\text{op}, e_1, e_2, l_1, l_2)) \)

What about \( \text{BINOP}(\text{op}, e_1, \text{ESEQ}(s, e_2)) \)?
Be careful not to exchange $s$ and $e_1$, in case $s$ affects $e_1$.

$$a + (--a) e_1: a, e_2: a-1, s: a=a-1$$

3. $\text{BINOP}(\text{op}, e_1, \text{ESEQ}(s, e_2)) =$

$$\text{ESEQ}(\text{MOVE}(\text{TEMP}(t), e_1),$$
$$\text{ESEQ}(s, \text{BINOP}(\text{op}, \text{TEMP}(t), e_2)))$$

$\text{CJUMP}(\text{op}, e_1, \text{ESEQ}(s, e_2), l_1, l_2) =$

$$\text{SEQ}(\text{MOVE}(\text{TEMP}(t), e_1),$$
$$\text{SEQ}(s, \text{CJUMP}(\text{op}, \text{TEMP}(t), e_2, l_1, l_2)))$$
ESEQ Identities (3)

If $s$ does not affect $e_1$ (i.e., MEM or TEMP assigned by $s$ is not referenced by $e_1$),
then we do not have to be as careful.

4. $\text{BINOP}(\text{op}, e_1, \text{ESEQ}(s, e_2)) = \text{ESEQ}(s, \text{BINOP}(\text{op}, e_1, e_2))$
CJUMP($\text{op}, e_1, \text{ESEQ}(s, e_2), l_1, l_2) = \text{SEQ}(s, \text{CJUMP}(\text{op}, e_1, e_2, l_1, l_2))$

Can we always tell if two expressions commute?
Commutative Operations

- Whether $\text{MOVE}(\text{MEM}(x), y)$ commutes with $\text{MEM}(z)$ depends on if $x = z$.
- Can we determine if $x = z$ at compile time?
- We make a conservative approximation: either two expressions “definitely commute” or “possibly do not commute”.
- Example: any statement definitely commutes with expression $\text{CONST}(n)$.
In general, for each `Tree.Stm` and `Tree.Exp` we identify the subexpressions and make rewriting rules for pulling any `ESEQ` s out.

Example: to rewrite the `Tree.ExpList` 

```
[e₁, e₂, ESEQ(s, e₃)]
```

we must pull out statement s (to occur before e₁, e₂).

**HOW? What if e₁ and e₂ commute with s?** What if e₁ or e₂ does not commute with s?
abstract public class Tree.Exp {
    // subexpr-extraction method (returns
    // list containing each subexpr)
    abstract public ExpList kids();
    // subexpr-insertion method (using ESEQ-
    // free version of each subexpr, builds
    // new version of Tree.Exp)
    abstract public Exp build(ExpList kids);
}

Tree.Stm has similar methods.
General Rewriting Rules (3)

```java
public class BINOP extends Exp {
    public int binop;
    public Exp left, right;
    public BINOP(int b, Exp l, Exp r) {...}
    public final static int PLUS=0, MINUS=1, ...
    public ExpList kids() {
        return new ExpList(left,
                           new ExpList(right, null));
    }
    public Exp build(ExpList kids) {
        return new BINOP(binop, kids.head,
                         kids.tail.head);
    }
}
```
General Rewriting Rules (4)

Tree.Stm do_stm(Tree.Stm s)
1. Uses s.kids() to get subexpressions of s (an expression list l).
2. Recursively pulls all the ESEQs out of l, yielding a group of side-effecting statements $s_1$ and a ESEQ-free list $l'$.
3. $SEQ(s_1, s.build(l'))$ constructs statement equivalent to s, but ESEQ-free.
Handling Return Values

- The Tree language allows \texttt{CALL} nodes as subexpressions.
- However, the actual implementation of \texttt{CALL} will put return value in \texttt{RV} register.
- What if \texttt{BINOP(PLUS, CALL(...), CALL(...))}?
- Assign each return value immediately to a fresh temporary register. \texttt{CALL (f, a)} becomes \texttt{ESEQ(MOVE(TEMP(t), CALL(f, a)), TEMP(t))}
Once a function body has been processed to remove **ESEQ** nodes and deal with **CALL** nodes, we have a tree with all **SEQ** nodes near the top.

To linearize, repeatedly apply

\[
\text{SEQ} (\text{SEQ}(a, b), c) = \text{SEQ} (a, \text{SEQ}(b, c))
\]

Now, in **SEQ**\((s_1, \text{SEQ}(s_2, ..., \text{SEQ}(s_{n-1}, s_n)...))\) **SEQ** nodes provide no structuring information.

Make a simple list of statements \(s_1, s_2, ..., s_{n-1}, s_n\).
Conditional Branches

- Why doesn't CJUMP of the Tree language correspond exactly to assembly language?
- To make translation to assembly language easy, we can order statements such that CJUMP\((cond, l_t, l_f)\) is immediately followed by LABEL\((l_f)\).
- 2 steps:
  1. Group list of canonical trees into basic blocks.
  2. Order the basic blocks into a set of traces.
Control Flow

- **Control flow** is the sequencing of instructions, ignoring data values in registers (or memory) and arithmetic calculations.
- However, not knowing data values means not knowing where conditional jumps will go.
- We say such jumps can go either way.
- We can group together any sequence of nonbranching instructions into a basic block.
Example: Control Flow

```java
x = this.init();
if(x<0) {
    System.out.println(x);
    y = 0;
}
else
    y = x;
return y;
```
A basic block is a sequence of statements that is always entered at the beginning and exited at the end. The first statement in the block is a LABEL. The last statement in the block is a JUMP or a CJUMP. There are no other LABELs, JUMPs, or CJUMPs in the block.
Dividing into Basic Blocks

1. Scan statement sequence from beginning to end.
   *If find LABEL, start a new block (previous block is ended).*
   *If find JUMP or CJUMP, end current block (next block is started).*

2. Scan blocks.
   *If any blocks do not end with JUMP or CJUMP, add JUMP to the next block's label.*
   *If any blocks do not begin with LABEL, create new label l and add LABEL(l).*
Example: Basic Blocks

s₁: MOVE(TEMP(x), CALL(init, ExpList(MEM(objPtr), null)))

s₂: CJUMP(LT, TEMP(x), CONST(0), NAME(t), NAME(f))

s₃: LABEL(t)

s₄: EXP(CALL(print, ExpList(TEMP(x), null)))

s₅: MOVE(TEMP(y), CONST(0))

s₆: JUMP(NAME(join))

s₇: LABEL(f)

s₈: MOVE(TEMP(y), TEMP(x))

s₉: JUMP(NAME(join))

s₁₀: LABEL(join)

s₁₁: MOVE(TEMP(RV), TEMP(y))

How does this statement sequence divide into basic blocks?
Traces

- A trace is a sequence of statements that could be consecutively executed during the program.
- A program has many overlapping traces.
- We want a set of traces that exactly covers the program (i.e., each basic block is in only one trace).
- To have few JUMPs from one trace to another, we want as few traces as possible to cover the program.
Algorithm Strategy

- Start with some basic block (making it the beginning of the trace) and follow a possible execution path (making it the rest of the trace).

Example:

- Start with block $b_1$. 
  $trace = b_1$
- $b_1$ jumps to $b_4$. 
  $trace = b_1, b_4$
- $b_4$ jumps to $b_6$. 
  $trace = b_1, b_4, b_6$
- $b_6$ ends in CJUMP to $b_7$ or $b_3$. 
  (choose either) 
  $trace = b_1, b_4, b_6, b_3$
Algorithm: Generation of Traces

Put all basic blocks in list $Q$.

while $Q$ is not empty
  Start a new (empty) trace $T$.
  Remove the head element $b$ from $Q$.
  while $b$ is not marked
    Mark $b$ and append to end of $T$.
    Examine blocks to which $b$ branches (its successors).
    if there is any unmarked successor $c$
      $b \leftarrow c$
  End current trace $T$.  

Finishing Up

- Now flatten ordered list of traces into one statement list.
- Most CJUMPs will now be followed by either their true or false label.
- For any CJUMP followed by its true label, switch the true and false labels and negate the condition.
- For any CJUMP\( (\text{cond}, a, b, l_t, l_f) \) not followed by either label, create label \( l'_f \) and rewrite as
  \[
  \text{CJUMP}(\text{cond}, a, b, l_t, l'_f); \text{LABEL}(l'_f); \text{JUMP}(\text{NAME}(l_f)).
  \]
Example: Traces

\[\begin{align*}
 s_0 &: \text{LABEL(fnbody)} \\
 B_1 \ s_1 &: \text{MOVE(TEMP(x), CALL(init, ExpList(MEM(objPtr), null)))} \\
 s_2 &: \text{CJUMP(LT, TEMP(x), CONST(0), NAME(t), NAME(f))} \\
 s_3 &: \text{LABEL(t)} \\
 B_2 \ s_4 &: \text{EXP(CALL(print, ExpList(TEMP(x), null)))} \\
 s_5 &: \text{MOVE(TEMP(y), CONST(0))} \\
 s_6 &: \text{JUMP(NAME(join))} \\
 s_7 &: \text{LABEL(f)} \\
 B_3 \ s_8 &: \text{MOVE(TEMP(y), TEMP(x))} \\
 s_9 &: \text{JUMP(NAME(join))} \\
 s_{10} &: \text{LABEL(join)} \\
 B_4 \ s_{11} &: \text{MOVE(TEMP(RV), TEMP(y))} \\
 s_{12} &: \text{JUMP(NAME(done))}
\end{align*}\]

What traces are generated for these basic blocks?