Why use an Intermediate Representation (IR)?

- More portable compiler architecture
IR Trees

Exp ::= CONST( int )
| NAME( Label )
| TEMP( Temp )
| BINOP( Op, Exp, Exp)
| MEM(Exp)
| CALL( Exp fun, List<Exp> args)
| ESEQ( Stm, Exp )

Stm ::= MOVE(Exp dst, Exp src)
| EXP(Exp)
| JUMP( Label )
| CJUMP( RelOp, Exp, Exp, Label thn, Label els)
| SEQ( Stm, Stm)
| LABEL( Label )

Data ::= DATA( Label, List<Exp> values)
Basic Blocks and Traces (Chapter 8)

• The “free form” IR the translator generates is “hard” to convert to machine code.

• Problematic features:
  – CJUMP with two labels.
  – ESEQ nodes (side effects in expressions)
  – CALL nodes in expressions (side effects)
  – CALL within CALL (problematic for return value register)

• Why did we do this?
  – Allowing these makes it easy to generate IR, but hard to turn IR into executable code.

• What are we going to do about it?
  – massage the IR into a more manageable form.
Three steps

• Convert to linearized, canonical IR
  – no ESEQ nodes, all SEQ nodes clustered at the top => list
  – CALL only in restricted places:
    • EXP(CALL(...))
    • MOVE(tmp, CALL(...))

• Convert linearized IR into basic blocks
  – basic block
  – starts with a label
  – no “internal” jumps
  – ends with a jump

• Trace Scheduling:
  – analyze and optimize JUMP and CJUMP nodes.
Step 1) Linearizing the IR

- Driven by a set of rewriting rules which
  - lift all statements out of expressions
  - convert all CALLs in EXPs to MOVE(...,CALL(...))

- Let’s think about this together and write out the rewriting rules on the board.
Eliminating ESEQ: Rewriting to “Lift” statements

```
ESEQ
  <s1>
  ESEQ
    <s2>
    <e>
  <e>

rule 1

SEQ
  <s1>
  <s2>
  <e>
```

means:

do <s1>
do <s2>
return eval(<e>)

means:
do <s1>
do <s2>
return eval(<e>)
Eliminating ESEQ: Rewriting to “Lift” statements

\[ (+) \quad \text{rule 2} \quad (\text{ESEQ}) \]

means:
\[
\begin{align*}
\text{do } \langle s1 \rangle \\
\text{val1 } &\leftarrow \text{eval } \langle e1 \rangle \\
\text{val2 } &\leftarrow \text{eval } \langle e2 \rangle \\
\text{return } \text{val1 + val2}
\end{align*}
\]

means:
\[
\begin{align*}
\text{do } \langle s1 \rangle \\
\text{val1 } &\leftarrow \text{eval } \langle e1 \rangle \\
\text{val2 } &\leftarrow \text{eval } \langle e2 \rangle \\
\text{return } \text{val1 + val2}
\end{align*}
\]
Eliminating ESEQ: Rewriting to “Lift” statements

rule 3

means:
val1 ← eval <e1>
do <s1>
val2 ← eval <e2>
return val1 + val2
different execution order!

Use only if order of <e1> and <s1> does not matter!
Eliminating ESEQ: Rewriting to “Lift” statements

OK, so let's assume the order does matter!

```
val1 ← eval <e1>
do <s1>
val2 ← eval <e2>
return val1 + val2
```

means:

```
val1 ← eval <e1>
do <s1>
val2 ← eval <e2>
return val1 + val2
```
Eliminating ESEQ: Rewriting to “Lift” statements

OK, so let's assume the order does matter!

```
val1 <- eval <e1>
do <s1>
val2 <- eval <e2>
return val1 + val2
```

means:

```
val1 <- eval <e1>
do <s1>
val2 <- eval <e2>
return val1 + val2
```

means:

```
val1 <- eval <e1>
do <s1>
val2 <- eval <e2>
return val1 + val2
```
Can We Eliminate all ESEQ with these 4 rules?

• Yes!
  – assuming our IRTree is a Stm at the top
  – assuming we define similar rules for other Exp nodes as well.

• Let's think about this for a second and convince ourselves of the truth of this claim:
  – Can we eliminate all ESEQ node from an IR Exp tree?
What about rule #3?

- Do we even need this rule?
  - No, we can always use rule #4 instead.
- But it is better to use #3 if we can
  - it produces less complex IR
- So when can we use rule #3?
Eliminating ESEQ: Rewriting to “Lift” statements

\[
+ \\
ESEQ
\\
\langle e1 \rangle \\
\langle s1 \rangle \\
\langle e2 \rangle
\]

means:
val1 <- eval \langle e1 \rangle
\begin{align*}
do \langle s1 \rangle \\
val2 & <- \text{eval} \langle e2 \rangle \\
\text{return } & \text{val1 + val2}
\end{align*}

\[
ESEQ
\\
\langle s1 \rangle \\
\langle e1 \rangle \\
\langle e2 \rangle
\]

means:
\begin{align*}
do \langle s1 \rangle \\
\text{different } & \text{val1 <- eval } \langle e1 \rangle \\
\text{execution} & \text{val2 <- eval } \langle e2 \rangle \\
\text{order! } & \text{return val1 + val2}
\end{align*}

Use only if ordering of \langle e1 \rangle and \langle s1 \rangle doesn't matter!
When will the order not matter?

- If the execution order of two computations doesn’t matter, we say that these computations “commute”.
- Can we determine when a statement \(<s>\) and an expression \(<e>\) “commute”?
  - In general not decidable (i.e., like the “halting problem”).
- Do the following commute?
  \[
  <s>: \text{MOVE( MEM (TEMP } x, \text{ CONST 0) }
  <e>: \text{MEM( TEMP y )}
  \]
Undecidable is bad? What to do?

• Since deciding whether $s_1$ and $e_1$ commute is “undecidable” does this mean we can not use rule #3 ever?
• Fortunately no, it means we can’t tell in general when it is safe to switch $s_1$ and $e_1$.
• But in specific cases it may be easy to see that it is safe!
  – e.g.,: $e_1 = \text{CONST } k$
• Can you think of other cases?
Undecidable is bad? What to do?

• Since deciding whether \(<s1>\) and \(<e1>\) commute is “undecidable” does this mean we can not use rule #3 ever?
• Fortunately no, it means we can’t tell in general when it is safe to switch \(<s1>\) and \(<e1>\).
• But in specific cases it may be easy to see that it is safe!
  – e.g.: \(<e1>\) = CONST \(k\)

• Can you think of other cases?

```java
static boolean commute(IRStm a, IRExp b) {
    return a instanceof EXP
        && ((EXP) a).exp instanceof CONST;
    || b instanceof NAME
    || b instanceof CONST;
}
```
Undecidable is bad? What to do?

• What do we do if we are unsure whether some $e_1$ and $s_1$ commute (i.e., they might commute, but we can not prove it).

• We make the “conservative” choice

• This is a common pattern:
  – when faced with an undecidable problem
  – make a safe conservative approximation!
We can generalize the ESEQ lifter

- We only discussed how to lift ESEQ out of BINOP (+) expression nodes and other ESEQ nodes.
- What about other nodes?
  - MOVE, EXP, MEM, CALL, CJUMP all contain expressions!
- They are similar. You could come up with rewriting rules for each one.
- In fact, we can implement a “generic” version of the rewriting rules.
  - Identifies the nested EXPs
  - Creates subtrees with different structure but the same EXPs
- The book contains the details
What about CALL nodes?

- Convert to linearized, canonical IR
  - no ESEQ nodes, all SEQ nodes clustered at the top => list
  - CALL only in restricted places:
    - EXP(CALL(...))
    - MOVE(tmp, CALL(...))

- Convert linearized IR into basic blocks
  - basic block
  - starts with a label
  - no “internal” jumps
  - ends with a jump

- Trace Scheduling:
  - analyze and optimize JUMP and CJUMP nodes.
What about CALL nodes?

• Three cases:
  – EXP(CALL(...))
    • OK, keep as is
  – MOVE(TEMP(...), CALL)
    • OK keep as is
  – Other
    • NOT OK (what to do?)
What about CALL nodes?

- Three cases:
  - \( \text{EXP}(	ext{CALL}(...)) \)
    - OK, keep as is
  - \( \text{MOVE}(	ext{TEMP}(...), \text{CALL}) \)
    - OK keep as is
  - In any other context, translate
    \[
    \text{CALL}(\text{fun}, \text{args})
    \]
    - To:
      \[
      \text{ESEQ}( \text{MOVE}( \text{TEMP}(\text{newTemp}), \text{CALL}(\text{fun}, \text{args})), \text{TEMP}(\text{newTemp}) )
      \]
    - And then what with this new ESEQ?
• Convert to linearized, canonical IR
  – no ESEQ nodes, all SEQ nodes clustered at the top => list
  – CALL only in restricted places:
    • EXP(CALL(...))
    • MOVE(tmp, CALL(...))

• Convert linearized IR into basic blocks
  – A basic block
    • starts with a label
    • no “internal” jumps or labels
    • ends with a jump

• Trace Scheduling:
  – analyze and optimize JUMP and CJUMP nodes.
Finding basic blocks

• We now have linearized IR – a list of Stm nodes
  – Show the linearized IR from running the compiler

• We just have to divide it up into a list of basic blocks, following the rules for basic blocks:
  – starts with a label
  – no “internal” jumps
  – ends with a jump

• The basic idea is to work through the list of Stms, finding the boundaries between basic blocks
  – And then fix things up so all the rules are obeyed
private void mkBlocks(List<IRStm> l) {
    if (!(l.head() instanceof LABEL)) // Must start with a LABEL.
        l = List.cons(LABEL(Label.gen()), l);

    for (IRStm stm : l) {
        if (stm.isJump()) {
            currentBlock.add(stm);
            endCurrentBlock();
        } else if (stm instanceof LABEL) {
            startNewBlock((LABEL)stm);
        } else
            currentBlock.add(stm);
    }

    if (currentBlock!=null) { // We "fell of the end" without a JUMP.
        doneLabel = Label.generate("DONE");
        currentBlock.add(JUMP(doneLabel));
        endCurrentBlock();
    }
}
On to trace scheduling ...

- Convert to linearized, canonical IR
  - no ESEQ nodes, all SEQ nodes clustered at the top => list
  - CALL only in restricted places:
    - EXP(CALL(...))
    - MOVE(tmp, CALL(...))

- Convert linearized IR into basic blocks
  - A basic block
    - starts with a label
    - no “internal” jumps or labels
    - ends with a jump

- Trace Scheduling:
  - analyze and optimize JUMP and CJUMP nodes.
Now we have a list of basic blocks ... represented by
- List<List<IRStm>>
- an “initial” Label
- a “done” Label

The basic blocks all
- begin with a LABEL
- end with a JUMP or CJUMP

=> The basic blocks can go in any order without changing the meaning!

Q: So what order do we choose?

=> Trace Scheduling = Choose an order for the basic bocks
Terminology: What’s a Trace?

Trace: sequence of statements that may be executed consecutively in some execution of the program.

Trace Scheduling:
- computing a covering set of traces:
  - each statement in the program is part of exactly one trace.

Seems pretty easy, but there is one more rule:
- Every CJUMP must be followed by its false label
  Why?