Compiler Construction

Intermediate Representation

University of Freiburg

Matthias Keil, Annette Bieniusa, Peter Thiemann

University of Freiburg

28. November 2016

Outline

University of Freiburg

1 Intermediate representation

2 Registers, heap and stack frames

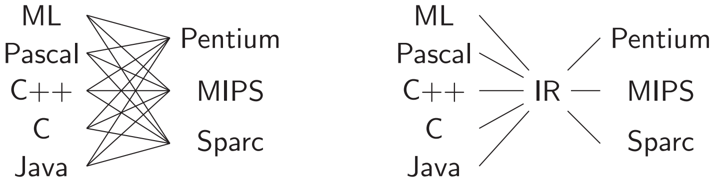
3 Memory layout

4 Contexts

5 Canonical Trees

Motivation

We could go directly from the AST to machine code, but …



Intermediate representation

- front end: lexical analysis, parsing, semantic analysis
- back end: machine specific optimization, translation to machine language
- intermediate code: machine and language independent optimization

Specifics of Intermediate Representation

A good IR is

- convenient to produce from AST
- convenient to translate into machine language
- small, with clear and simple semantics

Main differences: AST vs. IR

Conditionals	if-then-else vs. comparisons and conditional jumps
Method calls	various number of arguments vs. simple call (→ activation frames)
Memory layout	array and field deferencing vs. load/store on heap or stack

IR: Expressions

CONST(i)	integer constant i
NAME(n)	symbolic constant n [code label]
TEMP(t)	temporary t, one of arbitrary many “registers”
BINOP(o,e_1,e_2)	binary operator o with operands e_1 and e_2
MEM(e)	contents of a word of memory at address e
CALL(f,[$e_1, \ldots, e_n$])	procedure call
ESEQ(s,e)	expression sequence; evaluate statement s for side-effects, expression e for result

IR: Statements

MOVE(TEMP(t), e)	Evaluate e and move it into t.
MOVE(MEM (e_1),e_2)	Evaluate e_1 yielding address a; evaluate e_2 and move it into a.
EXP(e)	Evaluate e and discard result.
JUMP(e,[$l_1, \ldots, l_n$])	Transfer control (jump) to address e; $l_1, \ldots, l_n$ are all possible values for e. Often used: JUMP(l).
CJUMP(o,e_1,e_2,t,f)	Evaluate e_1, then e_2; compare their results using relational operator o. If true, jump to label t, else jump to label f.
SEQ(s_1,s_2)	Statement s_1 followed by statement s_2.
LABEL(n)	Define constant value of name n as current code address. NAME(n) can then be used as targets of jumps, calls, etc.

IR: Operators

Binary arithmetic and logical operators:

PLUS, MINUS, MUL, DIV	integer arithmetic operators
AND, OR, XOR	integer bitwise logical operators
LSHIFT, RSHIFT	integer logical shift operators
ARSHIFT	integer arithmetic right-shift

Relational operators:

EQ, NE	integer equality and non-equality (signed or unsigned)
LT, GT, LE, GE	integer inequalities (signed)
ULT, UGT, ULE, UGE	integer inequalities (unsigned)

Examples

Translate the following MiniJava statements to IR:

1. `if (x < y) x = y; else x = 0;`
2. `y = z[4];`

Examples

University of Freiburg

```
1 if (x < y) x = y; else x = 0;
```

- Assume, *x* corresponds to TEMP 5, y corresponds to TEMP 27.
- Define three (new) label names *L*1, *L*2, and *L*3.

```
CJUMP (LT, TEMP 5, TEMP 27, L1, L2)
L1  MOVE  (TEMP 5, TEMP 27)
JUMP L3
L2  MOVE  (TEMP 5, CONST 0)
L3  ...
```

Examples

2 `y = z[4];`

- Assume y corresponds to TEMP 27, and the array z is at memory location MEM a.
- Let w be the word size of MiniJava (e.g. 4 bytes).
- Calculate the offset for array index i.

```
MOVE (TEMP 27, +(MEM a, *(CONST 4, CONST w)))
```

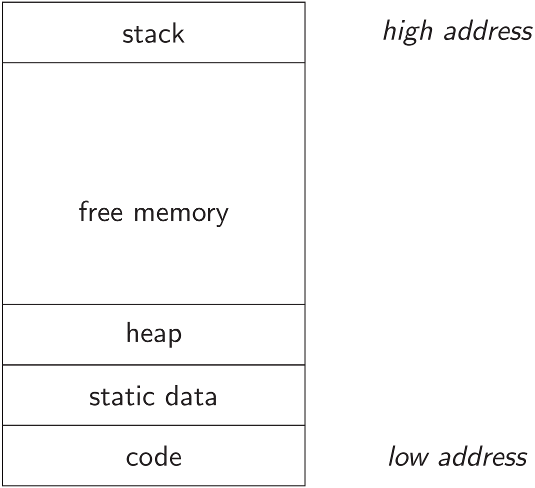
Here, we use `o(e1,e2)` as abbreviation for `BINOP(o,e1,e2)`.

Concepts of Memory layout

Registers	store local variables and temporary results; pass parameters and return results (for function calls), depending on the architecture's calling conventions.
Heap	area of memory used for dynamic memory allocation (e.g. arrays, objects)
Stack frames	maintained in program's virtual address space

Non-local data can be either referenced via static links to stack locations (also as local data of other frames), or to heap locations.

Traditional heap - stack arrangement



stack

free memory

heap

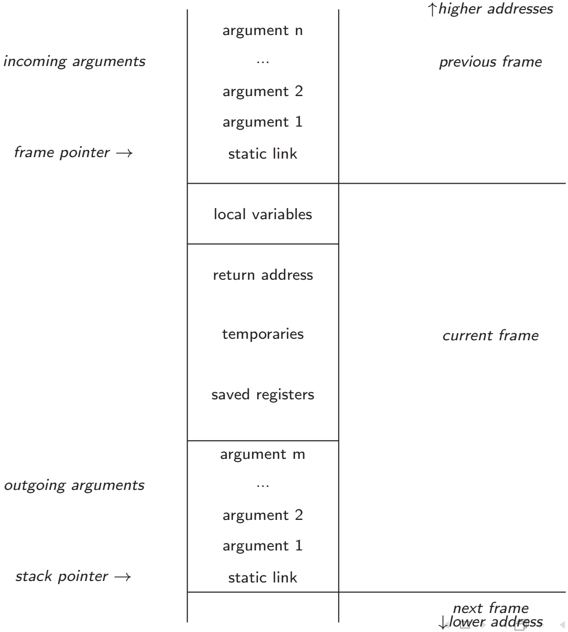
static data

code

high address

low address

Stack frames



↑*higher addresses*

argument n

incoming arguments …

previous frame

argument 2

argument 1

frame pointer → static link

local variables

return address

temporaries

current frame

saved registers

argument m

outgoing arguments …

argument 2

argument 1

stack pointer → static link

next frame

↓*lower address*

When calling a function...

University of Freiburg

The following actions are divided between the caller and the callee:

1. Evaluates actual arguments and puts values on the top of the caller's SF.
2. Stores return address in caller's SF (sometimes in the callee's SF).
3. Stores the caller's frame pointer register in callee's SF.
4. Modifies the frame pointer fp, making it point to callee's SF.
5. Modifies the stack pointer sp, making it point to the to top of the stack.
6. Go to callee's first instruction.
7. Callee begins execution.

When exciting a function...

1. Caller needs to retrieve the function return value.
2. Restores saved stack pointer for caller.
3. Restores saved register contents for caller.
4. Return to the caller.

Calling conventions

- Modern machines have a large set of registers (typically 32 registers).
- Register access is faster than memory loads and stores.
- Most functions have few parameters. Therefore, use small number of registers to pass parameters. The rest of the parameters, if any, can be passed in the stack.
- Returning function's results through registers.
- Caller-safe registers: caller is responsible to save and restore register contents.
- Callee-safe registers: callee is responsible to save and restore register contents.
- Convention is described in machine architecture manual.

When are variables written to memory?

- Variables passed by reference need to have a memory address (→ escaping vars).
- Variables accessed by a procedure nested inside the current one.
- Values which are too big to fit into a single register.
- Variable is an array (→ address arithmetic).
- Register holding the variable is needed for specific purpose.
- There are too many local variables and temporary values to fit all in registers (→ spilling).

Memory layout

Pointers/References

- Size is given by the natural word size of the given machine architecture.

Basic data types

- Integers are scalar, i.e. they occupy one word each.
- Boolean false is represented as 0, true by every non-zero value (e.g. 1).
- Other data types may be padded.

Memory layout

Strings

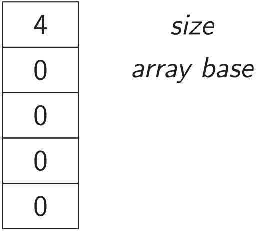
- Typically implemented statically at constant address of a segment of memory.
- In Java byte code, strings are collectively put into the constant pool.
- In assembly language, referred to by a label.
- PASCAL: fixed-length arrays of characters
- C: zero-terminated array of characters, variable length

Memory layout

Arrays (one-dimensional)

1. Size: reserve one word for the size of the array.
2. Entries: reserve space for entry of the array.

E.g. `new int[4]`



4	*size*
0	*array base*
0	
0	
0	

Memory layout

University of Freiburg

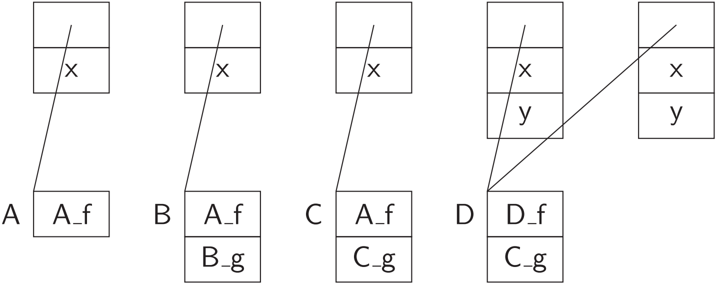
Objects

1. Methods: pointer to the *vtable* (virtual method table) of the corresponding class.
2. Fields: reserve space for fields of the class and for fields of the super classes

Memory layout

For OO languages with single-inheritance, a *prefixing* technique is used.

```
class A           {int x = 0; int f() {...} }
class B extends A {int g() {...} }
class C extends B {int g() {...} }
class D extends C {int y = 0; int f() {...} }
```



Using Expressions in different Contexts

Compare the translation for `x > 3` in

- `y = x > 3;`
- `if (x > 3) s1 else s2`

In C-like languages, what about `x = 3` in

- `x = 3;`
- `if (x = 3) s1 else s2`

Idea

Distinguish between different **contexts** of usage!

Contexts

Key ideas

You have an expression and want to use it as

- an expression: no problem
- a statement: `new EXP(...)`
- a conditional branch: create branch instruction with test against 0

You have a statement and want to use it as ...

- in MiniJava only as statement!

ExCtx(exp)	context where a value is required
NxCtx(stm)	context where no value is required
CxCtx	context with condition (abstract)
	RelCxCtx(op,left,right) relational operations
IfThenElseCtx	context of if-then-else construct

Contexts

We will keep the approach here a bit more general as there might be other kinds of ASTs. Conversion operations allow to use a form in the context of another :

unEx	converts to IR expression that evaluates inner tree and returns its value
unNx	converts to IR statement that evaluates inner tree but returns no value
unCx(t,f)	converts to IR statement that evaluates inner tree and branches to true destination if non-zero, to false destination otherwise

Translating MiniJava Expressions

Simple variables For now, we declare them as temporaries

ExCtx(TEMP t)

Arithmetic operations Choose the right binary operation!

a op b → ExCtx(BINOP (op,a.unEx,b.unEx))

Unary operations are translated with a trick:

- negation of integers → subtraction from zero
- unary complement → XOR with all ones

Translating MiniJava Expressions

University of Freiburg

Array elements Arrays are allocated on the heap.

$$e[i] \rightarrow \text{ExCtx(MEM (ADD(e.unEx(),}\\ \text{MUL(i.unEx(), CONST w))))}$$

Here, w is the target machine's word size.

In MiniJava, all values are word-sized.

Array bounds check: Check that array index i is between 0 and e.size. To this end, we will save the size in the word preceding the base.

Object fields Objects are allocated on the heap.

$$e.f \rightarrow \text{ExCtx(MEM (ADD(e.unEx(), CONST o)))}$$

where o is the byte offset of field f in the object.

Null pointer check: Check that object expression is non-null.

Translating MiniJava Expressions

Array allocation Arrays are allocated on the heap.

- Call external memory allocation function with needed size.
- Add size of array in the first memory chunk.
- Initialize then all fields with default values.
- Return address of first field as base of array.

Object allocation Objects are allocated on the heap.

- In constructor, call first external memory allocation function with needed size.
- Initialize pointer to the corresponding vtable (virtual method table).
- Initialize then all fields with default values.
- Return address of first field as base of object.

Translating MiniJava Expressions

Method call In OO language, `this` is an implicit variable. The pointer of the calling object will be added as parameter to each function!

- Fetch the class descriptor at offset 0 from object *c*.
- Fetch the method-instance pointer *p* from the (constant) offset f.
- Call *p*.

```
ExCtx(CALL(MEM( +(MEM(-(e0.unEx(), CONST(w)),
*(m.index ,CONST(w))),
e0.unEx(),e1.unEx(),...,en.unEx()))
```

Null pointer check: Check that object expression is non-null. For static methods, the function label/address can be done at compile time.

Translating MiniJava Control Structures

Code is structured into *basic blocks*:

- a maximal sequence of instructions without branches (straight-line code)
- a label starts a new basic block

For implementing control structures:

- Link up the basic blocks!
- Implementation requires bookkeeping (labels!).

While loops

```
while(c) s
```

- evaluate *c*
- if true, jump to loop body, else jump to next statement after loop
- evaluate loop body *s*
- jump to conditional
- if true, jump back to loop body

```
NxCtx(SEQ( SEQ(
LABEL(cond), c.unCx(body,done)),
SEQ( SEQ(
LABEL(body), SEQ(s.unNx(),JUMP(cond)))),
LABEL(done)))
```

For loops

University of Freiburg

```
for(i, c, u) s
```

- evaluate initialization statement *i*
- evaluate *c*
- if true, jump to loop body, else jump to next statement after loop
- evaluate loop body *s*
- evaluate update statement *u*
- jump to condition statement

```
NxCtx(SEQ( i.unNx(),
SEQ(SEQ(
LABEL(cond),c.unCx(body,done)),
SEQ(SEQ(
LABEL(body), SEQ(s.unNx(),SEQ(u.unNx(),
JUMP(cond)))),
LABEL(done))))
```

Break statement

University of Freiburg

- when translating a loop, push the done label on some stack
- `break` simply jumps to label on top of stack
- when done with translating the loop and its body, pop the label from the stack

Switch statement

`case E of` V_1`:` S_1 `...` V_n`:` S_n `end`

- evaluate the expression
- find value in case list equal to value of expression
- execute statement associated with value found
- jump to next statement after case

Key issue: finding the right case!

- sequence of conditional jumps (small case set): $O(|cases|)$
- binary search of an ordered jump table (sparse case set): $O(\log_2 |cases|)$
- hash table (dense case set): $O(1)$

Switch statement

```
            evaluate E into t
            if t != $V_1$ jump $L_1$
            code for $S_1$
            jump next
$L_1$:      if t != $V_2$ jump $L_2$
            code for $S_2$
            jump next
. . .
$L_{n-1}$:  if t != $V_n$ jump $L_n$
            code for $S_n$
            jump next
$L_n$:      code to raise run-time exception
next:
```

Switch statement

```
           evaluate E into t
           jump test
$L_1$:     code for $S_1$
           jump next
$L_2$:     code for $S_2$
           jump next
. . .
$L_n$:     code for $S_n$
           jump next
test:      if t $=$ $V_1$ jump $L_1$
           if t $=$ $V_2$ jump $L_2$
. . .
           if t $=$ $V_n$ jump $L_n$
           code to raise run-time exception
next:
```

Multi-dimensional arrays

University of Freiburg

Array allocation

- constant bounds:
    - allocate in static area, stack, or heap
    - no run-time descriptor is needed
- dynamic arrays: bounds fixed at run-time
    - allocate in stack or heap
    - descriptor is needed
- dynamic arrays: bounds can change at run-time
    - allocate in heap
    - descriptor is needed

Multi-dimensional arrays

Array layout

- Contiguous:
  - Row major: Rightmost subscript varies most quickly

    ```
    A[1,1], A[1,2], ...
    A[2,1], A[2,2], ...
    ```

    Used in PL/1, Algol, Pascal, C, Ada, Modula, Modula-2, Modula-3
  - Column major: Leftmost subscript varies most quickly

    ```
    A[1,1], A[2,1], ...
    A[1,2], A[2,2], ...
    ```

    Used in FORTRAN
- By vectors:
  - Contiguous vector of pointers to (non-contiguous) subarrays

Kinds of Contexts

ExCtx(exp)	context where a value is required
NxCtx(stm)	context where no value is required
CxCtx	context with condition (abstract)
	RelCxCtx(op,left,right) relational operations
IfThenElseCtx	context of if-then-else construct

Conversion operations allow to use a form in the context of another :

unEx	converts to IR expression that evaluates inner tree and returns its value
unNx	converts to IR statement that evaluates inner tree but returns no value
unCx(t,f)	converts to IR statement that evaluates inner tree and branches to true destination if non-zero, to false destination otherwise

Implementation

```
interface Ctx {
  Exp unEx();
  Stm unNx();
  Stm unCx(Label t, Label f);
}
```

```
class ExCtx implements Ctx {
  Exp exp;
  ExCtx (Exp e)    {exp = e;}
  Exp unEx()       {return exp;}
  Stm unNx()       {return new EXP(exp);}
  Stm unCx(Label t, Label f)
  { ... ? ... } // homework ;)
}
```

Implementation

```
class NxCtx implements Ctx {
  Stm stm;
  NxCtx (Stm s)   {stm = s;}
  Exp unEx()      { ... ? ... } // never needed in MiniJava
  Stm unNx()      {return stm;}
  Stm unCx(Label t, Label f)
  { ... ? ... } // never needed in MiniJava
}
```

Implementation

```
abstract class CxCtx implements Ctx {
  Exp unEx()      { ... ? ... }  // next slide
  Stm unNx()      { ... ? ... }  // homework ;)
  abstract Stm unCx(Label t, Label f);
}
```

Implementation

```
abstract class CxCtx implements Ctx {
  Exp unEx() {
    Temp r  = new Temp();
    Label t = new Label();
    Label f = new Label();
    return ESEQ(
    SEQ( MOVE (TEMP(r), CONST(1)),
    SEQ( this.unCx(t,f),
    SEQ( LABEL(f),
    SEQ( MOVE (TEMP(r), CONST(0)),
    LABEL(t))))),
    TEMP(r));
  }
  Stm unNx()      { ... ? ... }  // homework ;)
  abstract Stm unCx(Label t, Label f);
}
```

Implementation

For comparisons (e.g. `x < 5`):

```
class RelCxCtx extends CxCtx {
  RelOp o; Exp left; Exp right;
  RelCxCtx (RelOp o, Exp left, Exp right )
      {...}
  Stm unCx(Label t, Label f) {
    return CJUMP(o,left,right,t,f);
  }
}
```

Implementation

University of Freiburg

Translate short-circuiting boolean operators as if they were conditionals. May use if-then-else construct/conditional expression $e_1 ? e_2 : e_3$.

Example

`x < 5 && y > 0` is treated as

```
(x < 5) ? (y > 0) : 0
```

We translate $e_1 ? e_2 : e_3$ into an IfThenElseCtx(e_1,e_2,e_3) :

```
class IfThenElseCtx implements Ctx{
  Exp e1; Exp e2; Exp e3;
  IfThenElseCtx (Exp e1, Exp e2, Exp e3)
  {this.e1 = e1; this.e2 = e2; this.e3 = e3;}
  Exp unEx()        { ... ? ... }
  Stm unNx()        { ... ? ... }
  Stm unCx(Label t, Label f)
  { ... ? ... }
```

Implementation

When using a IfThenElseCtx as an expression:

```
Exp unEx(){
  Label t = new Label();
  Label f = new Label();
  Temp  r = new Temp();
  return ESEQ(
  SEQ( e1.unCx(t,f),
  SEQ( SEQ (LABEL (t),
  SEQ( MOVE ( TEMP(r), e2.unEx()),
  JUMP (j))),
  SEQ ( LABEL(f), SEQ( MOVE (TEMP(r), e3.unEx
      ()),
  JUMP (j))))),
  LABEL(j)),
  TEMP (r));
}
```

Implementation

When using a IfThenElseCtx as a conditional:

```
Stm unCx(Label t, Label f) {
  Label tt = new Label();
  Label ff = new Label();
  return SEQ ( e1.unCx(tt,ff),
  SEQ(SEQ (LABEL(tt),e2.unCx(t,f)),
  SEQ(LABEL(ff), e3.unCx(t,f))));
}
```

Motivation

Mismatches between IR and machine code

- Evaluation order of `ESEQ`'s within expressions must be made explicit, same for `CALL` nodes.
- `CALL` nodes at argument expression of other `CALL`s cause problems with registers.
- `CJUMP` may jump to either of two labels, conditional jumps of machines "fall through" if condition is false.

Motivation

Idea

Yet another tree re-writing step!

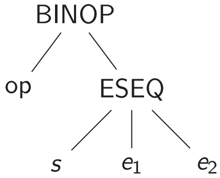
- Eliminate `SEQ` and `ESEQ` nodes ⇒ simple list of statements!
- `CALL` can only be subtree of `EXP(...)` or `MOVE(TEMP t,...)`.
- Group sequences into basic blocks without internal jumps or labels.
- Arrange basic blocks where every `CJUMP` is followed by false branch.

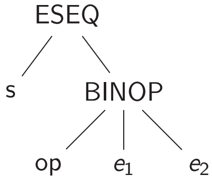
Re-writing of `ESEQ`(1)

University of Freiburg

$$\mathrm{ESEQ}(s_1,\ \mathrm{ESEQ}(s_2,e)) \quad \Rightarrow \quad \mathrm{ESEQ}(\mathrm{SEQ}(s_1,s_2),e)$$

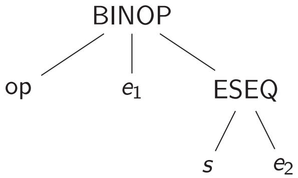
Re-writing of `ESEQ`(2)

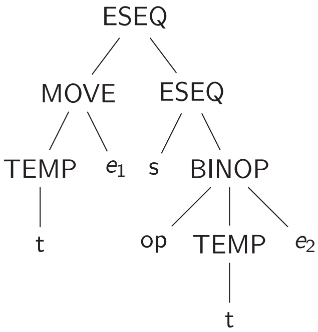




BINOP(op, ESEQ(s,e_1),e_2) ⇒ ESEQ(s,BINOP(op,e_1,e_2))
MEM(ESEQ(s,e_1)) ⇒ ESEQ(s,MEM(e_1))
JUMP(ESEQ(s,e_1)) ⇒ ESEQ(s,JUMP(e_1))
CJUMP(op,ESEQ(s,e_1),e_2,l_1,l_2) ⇒ SEQ(s,CJUMP(op,e_1,e_2,l_1,l_2))

Re-writing of `ESEQ`(3)



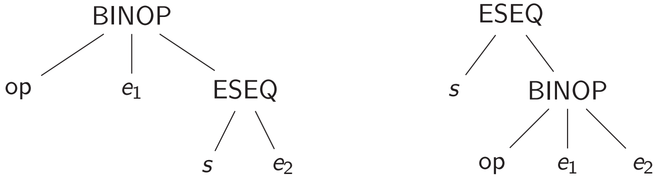


BINOP(op, e_1,ESEQ(s,e_2)) ⇒ ESEQ(MOVE(TEMP t,e_1),
ESEQ(s,
BINOP(op,TEMP t,e_2)))

CJUMP(op,e_1,ESEQ(s,e_2),l_1,l_2) ⇒ SEQ(MOVE(TEMP t,e_1),
SEQ(s,
CJUMP(op,TEMP t,e_2,l_1,l_2)))

Re-writing of `ESEQ`(4)

If s and e_1 commute, we can optimize:



$$\mathrm{BINOP}(\mathrm{op},\ e_1,\mathrm{ESEQ}(s,e_2)) \quad \Rightarrow \quad \mathrm{ESEQ}(s,\mathrm{BINOP}(\mathrm{op},e_1,e_2))$$

$$\mathrm{CJUMP}(\mathrm{op},e_1,\mathrm{ESEQ}(s,e_2),l_1,l_2) \quad \Rightarrow \quad \mathrm{SEQ}(\mathrm{s},\mathrm{CJUMP}(\mathrm{op},e_1,e_2,l_1,l_2))$$

Example

- MOVE(MEM(x),y) commutes with MEM(z) iff $x \neq z$.
- Any statement commutes with CONST(n).

General Rewriting Rules

From the examples so far, we can derive this somewhat general approach:

- Extract recursively all `ESEQ`'s out of all subexpressions.
- Generate statement sequences where sub-expressions are evaluated into temporaries.
- Rebuild original construct.

Use similar technique to eliminate nested function calls:

$$\text{CALL}(f, \mathit{args}) \Rightarrow \text{ESEQ}(\text{MOVE}(\text{TEMP t}, \text{CALL}(f, \mathit{args})), \text{TEMP t})$$

Basic Blocks and Traces

A basic block

- starts with a LABEL,
- end with a JUMP or CJUMP, and
- there are no other LABELs, JUMPs, or CJUMPs

A trace

- is a sequence of statements that could be consecutively executed in the program.

Arrange the blocks to get “optimal” traces!

Generating Traces

- Divide the list of statements of a function body into blocks.
- Put all the blocks into a list Q.
- While Q is not empty:
  - Start new (empty) trace T.
  - Remove head element b from Q.
  - While b is not marked:
    - Mark b.
    - Append b to the end of the current trace T.
    - Examine the blocks to which b branches:
      If there is any unmarked successor c, let it be the next b.
  - End the current trace T.

Finishing up

- Make sure that every CJUMP is followed by its false label.
    - If followed by true label, negate condition and swap labels.
    - If followed by neither label, insert dummy label f' and jump.

```
CJUMP(cond,a,b,t,f')
LABEL f'
JUMP(NAME f)
```

- Remove jumps that are immediately followed by their target label.

Building Traces of Basic Blocks

prologue statements	*prologue statements*	*prologue statements*
~~JUMP(NAME(test))~~	~~JUMP(NAME(test))~~	JUMP(NAME test)
LABEL(test)	LABEL(test)	LABEL(body)
CJUMP($>$,i, N,done,body)	CJUMP($\leq$,i, N,body,done)	*loop body statements*
LABEL(body)	LABEL(done)	~~JUMP(NAME(test))~~
loop body statements	*epilogue statements*	LABEL(test)
JUMP(NAME test)	LABEL(body)	CJUMP($\leq$,i, N,body,done)
LABEL(done)	*loop body statements*	LABEL(done)
epilogue statements	JUMP(NAME test)	*epilogue statements*

Alternative Intermediate Representations

- Directed acyclic graphs (DAGs): identifies common subexpression
- Three-address code: at most one operator at the right side of an instruction
- Static single assignment form (SSA): all assignments are to variables with distinct names