Compiler Construction 2012/2013: Intermediate Representation

Annette Bieniusa Peter Thiemann

December 3, 2012

Outline

1. Contexts
2. Canonical Trees

From Parse Trees to Intermediate Representation

- Parser generates ASTs according to the underlying CFG
- AST high-level and generic
- Translation to IR already incorporates some design decisions and code improvements

Expressions Appear 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`

Contexts

Key ideas

An expression may be used in different ways...

- to further compute with it value: no problem
- as a statement: `x = 3`
- as a condition: create branch instruction with test against 0

A statement may be used 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

A general framework for conversion operations in contexts.
They enable 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 **t** if non-zero, to false destination **f**, 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 $\rightarrow$ ExCtx(BINOP (op,a.unEx(),b.unEx()))

Unary operations are translated with a trick:

- negation of integers $\rightarrow$ subtraction from zero
- ones complement $\rightarrow$ XOR with all ones

Translating MiniJava Expressions

Array elements Arrays are allocated on the heap.

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

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, the array representation stores 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 - on the heap

- Call external memory allocation function with size $+1$.
- Store size of array in the first word.
- Initialize all elements with default values.
- Return address of first element as base of array.

Object allocation - on the heap

- In constructor, call external memory allocation function with needed size.
- Store pointer to the corresponding vtable (virtual method table).
- Initialize 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 offset f.
- Call *p*.

e0.m (e1, . . . , en) $\longrightarrow$

```
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

```
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

- 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
L1:       code for S1
          jump next
L2:       code for S2
          jump next
. . .
Ln:       code for Sn
          jump next
test:     if t = V1 jump L1
          if t = V2 jump L2
. . .
          if t = Vn jump Ln
          code to raise run-time exception
next:
```

Multi-dimensional arrays

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

Multi-dimensional arrays: Row-major layout

- Array $[1..N, 1..M]$ of T corresponds to array $[1..N]$ of array $[1..M]$ of T
- For an array $[\ldots, L_j..U_j, \ldots]$, the number of elements in dimension j is: $D_j = U_j - L_j + 1$
- Position of $A[i_1, \ldots, i_n]$:

$$
\begin{array}{rl}
 & (i_n - L_n) + (i_{n-1} - L_{n-1})D_n + \cdots + (i_1 - L_1)D_nD_{n-1}\cdots D_2 \\
= & i_n + i_{n-1}D_n + \cdots + i_1 D_nD_{n-1}\cdots D_2 \\
 & -(L_n + L_{n-1}D_n + \cdots - L_1 D_nD_{n-1}\cdots D_2) \\
= & \text{variable part} \\
 & -\text{constant part}
\end{array}
$$

The vector of partial products $D_nD_{n-1}, \ldots, D_n\cdots D_2$ can be precomputed

- Address of $A[i_1, \ldots, i_n]$:

address(A) + ((variable part - constant part) * element size)

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 enable using 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 MiniJav
  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

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))));
}
```

Outline

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

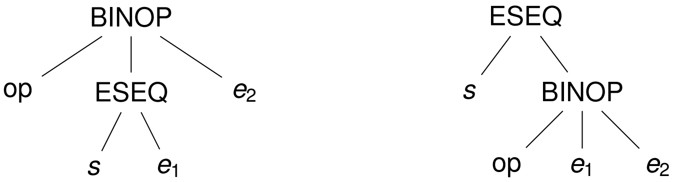
Yet another tree re-writing step!

- Eliminate `SEQ` and `ESEQ` nodes $\Rightarrow$ 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)

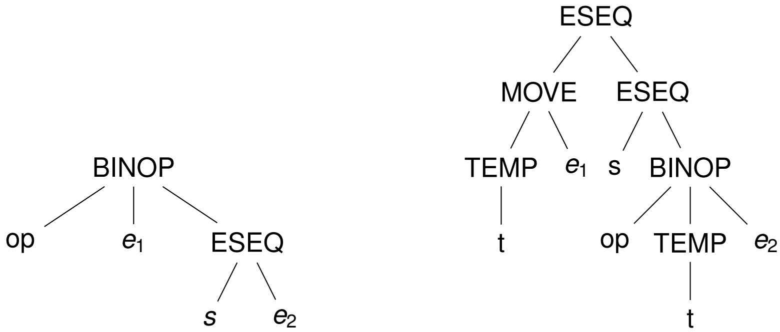
$$\text{ESEQ}(s_1, \text{ESEQ}(s_2, e)) \quad \Rightarrow \quad \text{ESEQ}(\text{SEQ}(s_1, s_2), e)$$

Re-writing of `ESEQ`(2)



BINOP(op, ESEQ(s,e_1),e_2)	$\Rightarrow$	ESEQ(s,BINOP(op,e_1,e_2))
MEM(ESEQ(s,e_1))	$\Rightarrow$	ESEQ(s,MEM(e_1))
JUMP(ESEQ(s,e_1))	$\Rightarrow$	ESEQ(s,JUMP(e_1))
CJUMP(op,ESEQ(s,e_1),e_2,l_1,l_2)	$\Rightarrow$	SEQ(s,CJUMP(op,e_1,e_2,l_1,l_2))

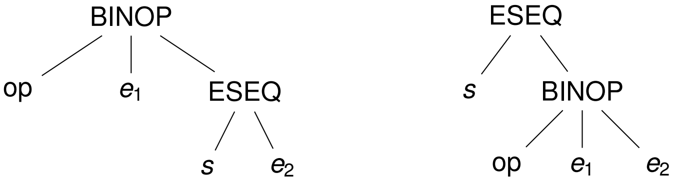
Re-writing of `ESEQ`(3)



BINOP(op, e_1,ESEQ(s,e_2))	$\Rightarrow$	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)	$\Rightarrow$	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, the move instruction is not needed:



BINOP(op, e_1,ESEQ(s,e_2))	$\Rightarrow$	ESEQ(s,BINOP(op,e_1,e_2))
CJUMP(op,e_1,ESEQ(s,e_2),l_1,l_2)	$\Rightarrow$	SEQ(s,CJUMP(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:

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

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 (later in course)