1. Contexts

2. Canonical Trees
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`
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!

**Key ideas**

- `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
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

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, 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.
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.
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 \( p \).

\[
\text{ExCtx} (\text{CALL} (\text{MEM} \left( + (\text{MEM} \left( -(e0.\text{unEx}()), \text{CONST}(w)) \right) \right) \star (m.\text{index}, \text{CONST}(w))),
\]
\[
e0.\text{unEx}(), e1.\text{unEx}(), ..., en.\text{unEx}()
\]

**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

\[ \text{NxCtx}(\text{SEQ}(\text{SEQ}(\text{LABEL}(\text{cond}), c.\text{unCx}(\text{body}, \text{done})),\text{SEQ}(\text{SEQ}(\text{LABEL}(\text{body}), \text{SEQ}(s.\text{unNx}(), \text{JUMP}(\text{cond}))),\text{LABEL}(\text{done})))) \]
For loops

\[
\text{for}(i, c, u) \ s
\]

- 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

\[
NxCtx(SEQ(\ i.unNsx()),
        SEQ(SEQ(\ LABEL(cond),c.unCx(body,done)),
            SEQ(SEQ(\ LABEL(body), SEQ(s.unNsx(),SEQ(u.unNsx(),
                  JUMP(cond)))))
            LABEL(done))))
\]
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  \( V_1: \  S_1 \  \ldots \  V_n: \  S_n \)  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 != V₁ jump L₁
code for S₁
jump next

L₁: if t != V₂ jump L₂
code for S₂
jump next

... Ln⁻₁: if t != Vₙ jump Lₙ
code for Sₙ
jump next

Ln: code to raise run-time exception
next:
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

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
    
    \[
    A[1,1], A[1,2], \ldots \\
    A[2,1], A[2,2], \ldots
    \]
    
    Used in PL/1, Algol, Pascal, C, Ada, Modula, Modula-2, Modula-3
  
  - Column major: Leftmost subscript varies most quickly
    
    \[
    A[1,1], A[2,1], \ldots \\
    A[1,2], A[2,2], \ldots
    \]
    
    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 \(T\)

- Number of elt’s in dim \(j\):

\[
D_j = U_j - L_j + 1
\]

- Position of \(A[i_1, \ldots, i_n]\):

\[
(i_n - L_n) + (i_{n-1} - L_{n-1})D_n + \cdots + (i_1 - L_1)D_nD_{n-1} \cdots D_2
\]

\[
= i_n + i_{n-1}D_n + \cdots + i_1D_nD_{n-1} \cdots D_2
\]

\[
- (L_n + L_{n-1}D_n + \cdots - L_1D_nD_{n-1} \cdots D_2)
\]

\[
= \text{variable part}
\]

\[
- \text{constant part}
\]

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

\[
\text{address}(A) + ((\text{variable part} - \text{constant part}) \times \text{element size})
\]
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 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
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 ;)
}
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
}
abstract class CxCtx implements Ctx {
    Exp unEx() { ... ? ... } // next slide
    Stm unNx() { ... ? ... } // homework ;)
    abstract Stm unCx(Label t, Label f);
}
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);
}
For comparisons (e.g. $x < 5$):

```java
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);
    }
}
```
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**

\( x < 5 \land y > 0 \) is treated as

\[
(x < 5) \ ? \ (y > 0) \ : \ 0
\]

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

```java
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)
        { ... ? ... }
}
```
When using a `IfThenElseCtx` as an expression:

```java
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( MOV ( TEMP(r), e2.unEx()),
                           JUMP (j)) ),
                 SEQ ( LABEL(f), SEQ( MOV (TEMP(r), e3.unEx()),
                                 JUMP (j)) ),
                 LABEL(j)),
                 TEMP (r));
    }
}
```
When using a IfThenElseCtx as a conditional:

```java
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))));
}
```
1 Contexts

2 Canonical Trees
Mismatches between IR and machine code

- Evaluation order of \texttt{ESEQ}'s within expressions must be made explicit, same for \texttt{CALL} nodes.
- \texttt{CALL} nodes at argument expression of other \texttt{CALL}s cause problems with registers.
- \texttt{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 $\text{SEQ}$ and $\text{ESEQ}$ nodes $\Rightarrow$ simple list of statements!
- $\text{CALL}$ can only be subtree of $\text{EXP}(\ldots)$ or $\text{MOVE}(\text{TEMP} t,\ldots)$.
- Group sequences into basic blocks without internal jumps or labels.
- Arrange basic blocks where every $\text{CJUMP}$ is followed by false branch.
Re-writing of $\text{ESEQ}(1)$

$\text{ESEQ}(s_1, \text{ESEQ}(s_2, e)) \Rightarrow \text{ESEQ}(\text{SEQ}(s_1, s_2), e)$
Re-writing of $\text{ESEQ}(2)$

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

$\text{MEM}(\text{ESEQ}(s, e_1)) \Rightarrow \text{ESEQ}(s, \text{MEM}(e_1))$

$\text{JUMP}(\text{ESEQ}(s, e_1)) \Rightarrow \text{ESEQ}(s, \text{JUMP}(e_1))$

$\text{CJUMP}(\text{op}, \text{ESEQ}(s, e_1), e_2, l_1, l_2) \Rightarrow \text{SEQ}(s, \text{CJUMP}(\text{op}, e_1, e_2, l_1, l_2))$
Re-writing of $\text{ESEQ}(3)$

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

CJUMP($op$, $e_1$, ESEQ($s$, $e_2$), $l_1$, $l_2$) \quad \Rightarrow \quad \text{SEQ}(\text{MOVE}(\text{TEMP} \ t, e_1), \text{SEQ}(s, \text{CJUMP}(op, \text{TEMP} \ t, e_2, l_1, l_2)))$
Re-writing of $\text{ESEQ}(4)$

If $s$ and $e_1$ commute, we can optimize:

\[
\begin{align*}
\text{BINOP} & \quad \text{ESEQ} \\
\text{op} & \quad e_1 \\
\text{ESEQ} & \quad s \\
& \quad e_2
\end{align*}
\]

\[
\text{BINOP(op, } e_1, \text{ESEQ(s, } e_2)) \quad \Rightarrow \quad \text{ESEQ(s, } \text{BINOP(op, } e_1, e_2))
\]

\[
\text{CJUMP(op, } e_1, \text{ESEQ(s, } e_2), l_1, l_2) \quad \Rightarrow \quad \text{SEQ(s, } \text{CJUMP(op, } e_1, e_2, l_1, l_2))
\]

**Example**

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

\[
\text{CALL}(f, \text{args}) \Rightarrow \text{ESEQ} (\text{MOVE} (\text{TEMP} \ t, \ \text{CALL}(f, \text{args})), \text{TEMP} \ t)
\]
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.
    
    $\text{CJUMP}(\text{cond}, a, b, t, f')$
    
    $\text{LABEL } f'$
    
    $\text{JUMP (NAME } f)$

- Remove jumps that are immediately followed by their target label.
### Building Traces of Basic Blocks

<table>
<thead>
<tr>
<th>prologue statements</th>
<th>prologue statements</th>
<th>prologue statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>JUMP(NAME(test))</td>
<td>JUMP(NAME(test))</td>
<td>JUMP(NAME test)</td>
</tr>
<tr>
<td>LABEL(test)</td>
<td>LABEL(test)</td>
<td>LABEL(body)</td>
</tr>
<tr>
<td>CJUMP(&gt;,i, N,done,body)</td>
<td>CJUMP(≤,i, N,done,done)</td>
<td>LABEL(body)</td>
</tr>
<tr>
<td>LABEL(body)</td>
<td>LABEL(done)</td>
<td>loop body statements</td>
</tr>
<tr>
<td>loop body statements</td>
<td>epilogue statements</td>
<td>epilogue statements</td>
</tr>
<tr>
<td>LABEL(done)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>epilogue statements</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>loop body statements</td>
<td></td>
</tr>
<tr>
<td></td>
<td>JUMP(NAME test)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>LABEL(done)</td>
</tr>
<tr>
<td></td>
<td>LABEL(done)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>epilogue statements</td>
<td></td>
</tr>
</tbody>
</table>
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