

# Winter 2007-2008 Compiler Construction T10 – IR part 3

Mooly Sagiv and Roman Manevich  
School of Computer Science  
Tel-Aviv University

## Today

|             |                  |                         |     |                   |                  |                 |                 |
|-------------|------------------|-------------------------|-----|-------------------|------------------|-----------------|-----------------|
| IC          | Lexical Analysis | Syntax Analysis Parsing | AST | Symbol Table etc. | Inter. Rep. (IR) | Code Generation | exe             |
| IC Language |                  |                         |     |                   |                  |                 | Executable code |

- Today:
  - LIR
    - Spec. + microLIR
    - Strings and arrays
    - PA4
    - Optimizations

2

## LIR language

- Supports
  - Literal strings
  - Dispatch tables
  - Instruction set
  - Unbounded number of registers
  - Object/array allocation via library functions
    - Updating DVPtr with LIR instructions
    - Notice special syntax for parameter passing for static/virtual function calls
- No representation of activation records (frames)
- No calling sequence protocols
  - Just Call/Return instructions
  - **this** variable “magically” updated on call to virtual functions

3

## Translating call/return

TR[C.foo(e1, ..., en)]    R1 := TR[e1]    formal parameter name

...    Rn := TR[en]

StaticCall C.foo(x1=R1, ..., xn=Rn), R    actual argument register

TR[e1.foo(e2)]    R1 := TR[e1]

                  R2 := TR[e2]

VirtualCall R1.Cfoo(x=R2), R    Constant representing offset of method f in dispatch table of class type of e1

TR[return e]    R1 := TR[e]

Return R1

4

## LIR translation example

```

class A {
  int z;
  string s;
  int foo(int y) {
    int z=y+1;
    return z;
  }
  static void main(string[] args) {
    A p = new B();
    p.foo(5);
  }
}

class B extends A {
  int z;
  int foo(int y) {
    s = "y" + Library.itos(y);
    Library.println(s);
    int[] sarr = Library.stoa(s);
    int l = sarr.length;
    Library.println(l);
    return l;
  }
}

```

Annotations:

- Translating virtual functions (dispatch tables) - points to `foo` in class A
- Translating the main function - points to `main` in class A
- Translating virtual function calls - points to `p.foo(5)`
- Translation for literal strings - points to `"y"`
- Translating `.length` operator - points to `length` property access

5

## LIR program (manual trans.)

```

str1: "y="
_DV_A: [A.foo]
_DV_B: [B.foo]

.A.foo:
  Move y,R1
  Add i,R1
  Move R1,z
  Return z

.B.foo:
  Library __itos(y),R1
  Library __stringCat(str1,R1),R2
  Move this,R3
  MoveField R2,R3.3
  MoveField R3.3,R4
  Library __println(R4),Rdummy
  Library __stoa(R4),R5
  Move R5,sarr
  ArrayLength sarr,R6
  Move R6,l
  Library __println(l),Rdummy
  Return l

# main in A
.ic_main:
  Library __allocateObject(16),R1
  MoveField _DV_B.R1.0
  MoveField _DV_A.R1.0
  VirtualCall R1.0(y=5),Rdummy

```

Annotations:

- Literal string in program - points to `str1: "y="`
- dispatch table for class A - points to `_DV_A: [A.foo]`
- dispatch table for class B - points to `_DV_B: [B.foo]`
- int foo(int y) - points to `.A.foo:`
- int z=y+1; - points to `Add i,R1`
- return z; - points to `Return z`
- int foo(int y) - points to `.B.foo:`
- Library.itos(y); - points to `Library __itos(y),R1`
- "y" + Library.itos(y); - points to `Library __stringCat(str1,R1),R2`
- this.s = "y" + Library.itos(y); - points to `Move this,R3`
- int[] sarr = Library.stoa(s); - points to `Library __stoa(R4),R5`
- int l = sarr.length; - points to `ArrayLength sarr,R6`
- Library.println(s); - points to `Library __println(l),Rdummy`
- return l; - points to `Return l`
- A [static void main(string[] args){...}] - points to `# main in A`
- A p = new B(); - points to `Library __allocateObject(16),R1`
- Update DVPtr of new object - points to `MoveField _DV_B.R1.0`
- p.foo(5) - points to `VirtualCall R1.0(y=5),Rdummy`

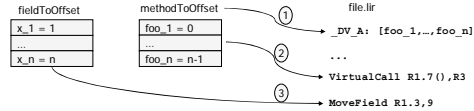
6

## Class layout implementation

```
class A {
  int x_1;
  ...
  boolean x_n;
  void foo_1(...) {-}
  ...
  int foo_n(...) {-}
}
```

```
class ClassLayout {
  Map<Method,Integer> methodToOffset;
  // DVPtr = 0
  Map<Field,Integer> fieldToOffset;
}
```

- 1: generate dispatch tables
- 2: determine method offsets in virtual calls
- 3: determine field offsets in field access statements



7

## microLIR simulator

- Java application
  - Accepts file.lir (your translation)
  - Executes program
- Use it to test your translation
  - Checks correct syntax
  - Performs lightweight semantic checks
  - Runtime semantic checks
  - Debug modes (-verbose:1|2)
  - Prints program statistics (#registers, #labels, etc.)
- Comes with sample inputs
- Comes with sources (you can use in PA4)
- Read manual

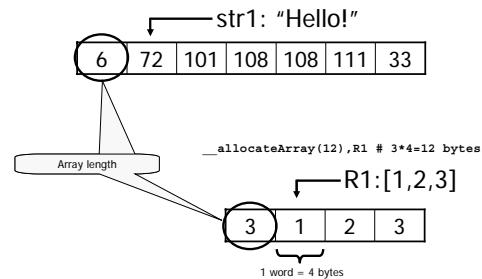
8

## PA4

- Translate AST to LIR (file.ic -> file.lir)
  - Dispatch table for each class
  - Literal strings (all literal strings in file.ic)
  - Instruction list for every function
    - Leading label for each function `_CLASS_FUNC`
    - Label of main function should be `_ic_main`
- Maintain internally for each function
  - List of LIR instructions
  - Reference to method AST node
    - Needed to generate frame information in PA5
- Maintain for each call instruction
  - Reference to method AST
    - Needed to generate call sequence in PA5
- Optimizations (WARNING: only after assignment works)
  - Keep optimized and non-optimized translations separately

9

## Representing arrays/strings



10

## LIR optimizations

- Aim to reduce number of LIR registers
  - Reduce size of activation records
  - Allow better register allocation in PA5
  - Also reduces number of instructions
- Avoid storing variables and constants in registers
- Use accumulator registers
- Reuse "dead" registers
- Weighted register allocation
- Global register allocation
  - Extra bonus
  - Let me know if you plan to implement
- Merge consecutive labels
  - Left to PA5 (generates additional labels)

11

## Avoid storing constants and variables in registers

- Naive translation of AST leaves
  - Each instruction has target register
  - For a constant  $TR[5] = \text{Move } 5, Rj$
  - For a variable  $TR[x] = \text{Move } x, Rk$
- Better translation
  - For a constant  $TR[5] = 5$
  - For a variable  $TR[x] = x$
  - What about  $TR[x+5] = ?$ 
    - WRONG:**  $TR[x+5] = \text{Add } TR[x], TR[5] = \text{Add } x, 5$
    - Right:**  $TR[x+5] = \text{Move } 5, R1$   
 $\text{Add } x, R1$
  - Assign to register if **both operands** non-registers

12

## Accumulator registers

- Use same register for sub-expression and result
- Very natural for 2-address code and previous optimization

TR[e1 OP e2]

|                   |                       |
|-------------------|-----------------------|
| Naive translation | Better translation    |
| R1 := TR[e1]      | R1 := TR[e1]          |
| R2 := TR[e2]      | R2 := TR[e2]          |
| R3 := R1 OP R2    | <b>R1 := R1 OP R2</b> |

13

## Accumulator registers

TR[e1 OP e2]

a+(b\*c)

|                   |                       |
|-------------------|-----------------------|
| Naive translation | Better translation    |
| R1 := TR[e1]      | R1 := TR[e1]          |
| R2 := TR[e2]      | R2 := TR[e2]          |
| R3 := R1 OP R2    | <b>R1 := R1 OP R2</b> |
| Move a,R1         | Move b,R1             |
| Move b,R2         | Mul c,R1              |
| Mul R1,R2         | Add a,R1              |
| Move R2,R3        |                       |
| Move c,R4         |                       |
| Add R3,R4         |                       |
| Move R4,R5        |                       |

14

## Accumulator registers cont.

- For instruction with registers R1,...,Rn dedicate one register for accumulation
- Accumulating instructions, use:
  - `MoveArray R1[R2],R1`
  - `MoveField R1.7,R1`
  - `StaticCall _foo(R1,...),R1`
  - ...

15

## Reuse registers

- Registers have very-limited lifetime
  - Currently stored values used exactly once
  - (All LIR registers become dead after statement)
- Suppose TR[e1 OP e2] translated as R1:=TR[e1], R2:=TR[e2], R1:=R1 OP R2  
Registers from TR[e1] can be reused in TR[e2]
- Algorithm:
  - Use a stack of temporaries (LIR registers)
  - Stack corresponds to recursive invocations of t := TR[e]
  - All the temporaries on the stack are alive

16

## Live register stack

- Implementation: use counter c to implement live register stack
  - Registers R(0)...R(c) are alive
  - Registers R(c+1),R(c+2)... can be reused
  - Push means increment c, pop means decrement c
- In the translation of R(c)=TR[e1 OP e2]

```
R(c) := TR[e1]
----- c = c + 1
R(c) := TR[e2]
----- c = c - 1
R(c) := R(c) OP R(c+1)
```

17

## Example

R0 := TR[((c\*d)-(e\*f))+(a\*b)]

```
----- c = 0
R0 := TR[((c*d)-(e*f))+(a*b)]
----- c = c + 1
R0 := c*d
----- c = c - 1
R1 := e*f
----- c = c + 1
R0 := R0-R1
----- c = c - 1
R1 := a*b
----- c = c - 1
R0 := R0 + R1
```

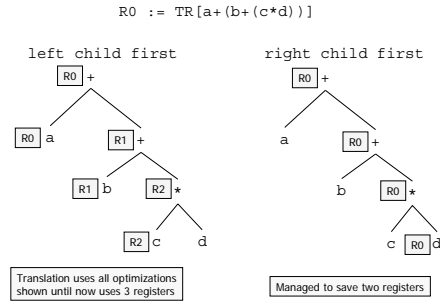
18

## Weighted register allocation

- Suppose we have expression  $e1 \text{ OP } e2$ 
  - $e1, e2$  without side-effects (function calls, assignments)
  - $\text{TR}[e1 \text{ OP } e2] = \text{TR}[e2 \text{ OP } e1]$
  - Does order of translation matter? ... Yes
- Use the **Sethi & Ullman algorithm**
  - Weighted register allocation – translate heavier sub-tree first
  - Optimal local (per-statement) allocation for side-effect-free statements

19

## Example



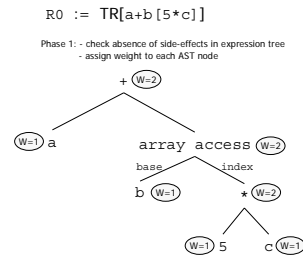
20

## Weighted register allocation

- Can save registers by re-ordering subtree computations
- Label each node with its weight
  - Weight = number of registers needed
  - Leaf weight known
  - Internal node weight
    - $w(\text{left}) > w(\text{right})$  then  $w = \text{left}$
    - $w(\text{right}) > w(\text{left})$  then  $w = \text{right}$
    - $w(\text{right}) = w(\text{left})$  then  $w = \text{left} + 1$
- Choose heavier child as first to be translated
- WARNING:** have to check that no side-effects exist before attempting to apply this optimization (pre-pass on the tree)

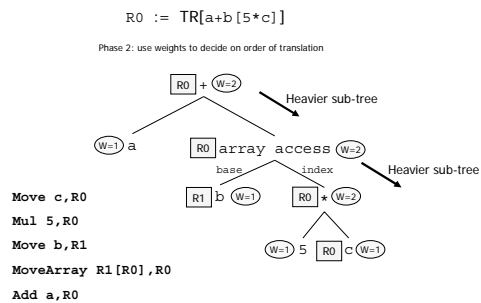
21

## Weighted reg. alloc. example



22

## Weighted reg. alloc. example



23

## Notes on weighted register alloc.

- Weighted register allocation in LIR
  - Get to know LIR optimization algorithms for exam
- Must reset LIR register counter after every IC statement:
  - $x=y; y=z;$  should not be translated to
 

```

Move y,R0
Move R0,x
Move z,R1
Move R1,y
                    
```
  - But rather to
 

```

Move y,R0
Move R0,x # Finished translating statement. Set c=0
Move z,R0
Move R0,y
                    
```

24

## Tips for PA4

- Keep **list** of LIR instructions for each translated method
- Keep **ClassLayout** information for each class
  - Field offsets
  - Method offsets
  - Don't forget to take superclass fields and methods into account
- May be useful to keep reference in each LIR instruction to AST node from which it was generated
- Two AST passes:
  - Pass 1:
    - Collect and name strings literals (**Map<ASTStringLiteral, String>**)
    - Create **ClassLayout** for each class
  - Pass 2: use literals and field/method offsets to translate method bodies
- Finally: print string literals, dispatch tables, print translation list of each method body

25

בוא ניקח הפסקה  
ונחזור לתרגול הכנה למבחן

26