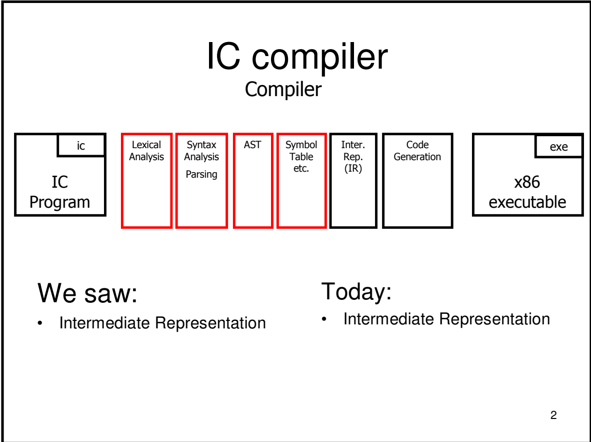
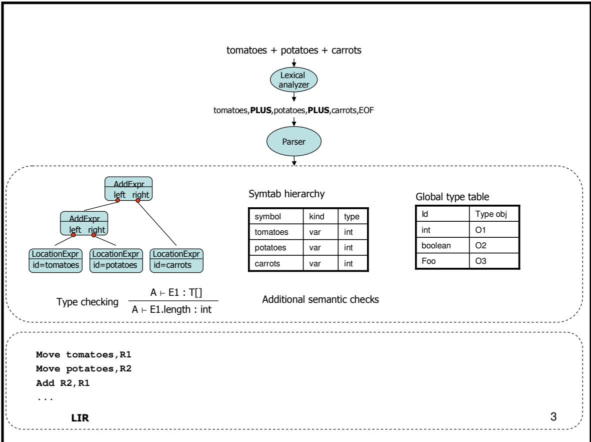
Compiler Construction

Intermediate Representation II

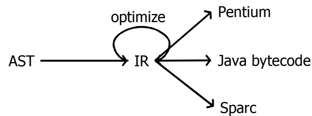
Ran Shaham and Ohad Shacham
School of Computer Science
Tel-Aviv University





Intermediate representation

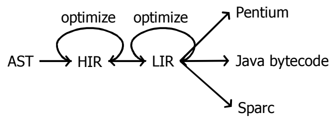
- Allows language-independent, machine independent optimizations and transformations
- Easy to translate from AST
- Easy to translate to assembly



4

Multiple IRs

- Some optimizations require high-level structure
- Others more appropriate on low-level code
- Solution: use multiple IR stages



5

LIR instructions

Instruction	Meaning
Move c,Rn	Rn = c (Immediate (constant))
Move x,Rn	Rn = x (Memory (variable))
Move Rn,x	x = Rn
Add Rm,Rn	Rn = Rn + Rm
Sub Rm,Rn	Rn = Rn – Rm
Mul Rm,Rn	Rn = Rn * Rm
…	

Note 1: rightmost operand = operation destination
Note 2: two register instr - second operand doubles as source and destination

6

Runtime organization

- Representation of basic types
- Representation of allocated objects
  - Class instances
    - Dispatch vectors
  - Strings
  - Arrays
- Procedures

7

Representing data at runtime

- Source language types
  - int, boolean, string, object types, arrays etc.
- Target language types
  - bytes, address representation, …
- Compiler maps source types to some combination of target types
  - Implement source types using target types

8

Representing basic types in IC

- int, boolean, string
  - Simplified representation: 32 bit for all types
  - **`boolean`** type could be implemented with single byte
- Arithmetic operations
  - Addition, subtraction, multiplication, division, remainder
- Mapped directly to target language types and operations
  - Exception: string concatenation implemented using library function **`__stringCat`**

9

Pointer types

- Represent addresses of source language data structures
- Usually implemented as an unsigned integer (4 bytes)
- Pointer dereferencing – retrieves pointed value
- May produce an error
  - Null pointer dereference

10

Object types

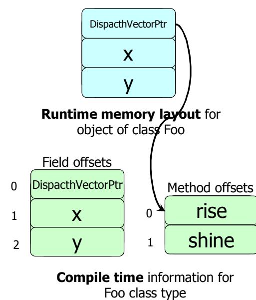
- Basic operations
  - Field selection
    - computing address of field, dereferencing address
  - Method invocation
    - Identifying method to be called, calling it

11

Object types

```
class Foo {
  int x;
  int y;

  void rise() {…}
  void shine() {…}
}
```

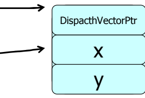


12

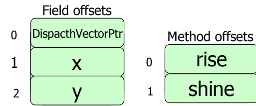
Field selection

```
Foo f;
int q;

q = f.x;
```



Runtime memory layout for object of class Foo



Compile time information for Foo class type generated during LIR translation

```
Move f,R1
MoveField R1.1,R2
Move R2,q
```

13

Implementation

- Store map for each `ClassType`
  - From field to offset
    - Note that 0 reserved for DispatchVectorPtr
  - From method to offset

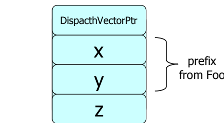
14

Object types and inheritance

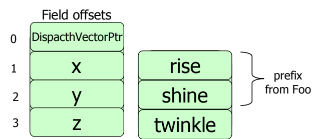
```
class Foo {
  int x;
  int y;

  void rise() {…}
  void shine() {…}
}

class Bar extends Foo {
  int z;
  void twinkle() {…}
  void rise() {…}
}
```



Runtime memory layout for object of class Bar



Compile time information for Bar class type

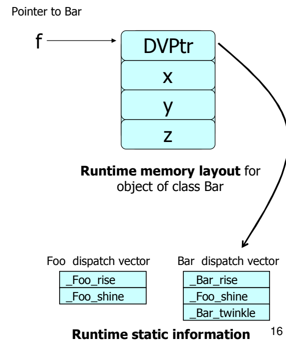
15

Object types and polymorphism

```
class Foo {
  …
  void rise() {…}
  void shine() {…}
}

class Bar extends Foo {
  void rise() {…}
}

class Main {
  void main() {
    Foo f = new Bar();
    f.rise();
}
```



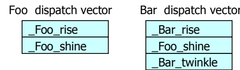
16

Dynamic binding

```
class Foo {
  …
  void rise() {…}
  void shine() {…}
}

class Bar extends Foo {
  void rise() {…}
}
```

```
class Main {
  void main() {
    Foo f = new Bar();
    f.rise();
}
```



- Finding the right method implementation
- Done at runtime according to object type
- Using the *Dispatch Vector* (a.k.a. *Dispatch Table*)

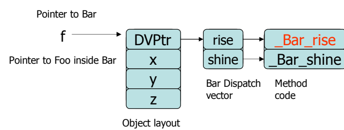
17

Dispatch vectors in depth

```
class Foo {
  …
  void rise() {…}   0
  void shine() {…}  1
}
```

```
class Bar extends Foo {
  void rise() {…}  0
}
```

```
class Main {
  void main() {
    Foo f = new Bar();
    f.rise();
}
```



- Vector contains addresses of methods
- Indexed by method-id number
- A method signature has the same id number for all subclasses

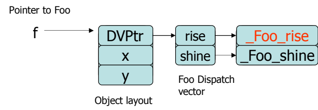
18

Dispatch vectors in depth

```
class Foo {
  …
  void rise() {…}    0
  void shine() {…}   1
}
```

```
class Bar extends Foo {
  void rise() {…}  0
}
```

```
class Main {
  void main() {
    Foo f = new Foo();
    f.rise();
}
```



19

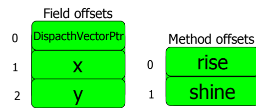
Object creation

```
Foo f = new Bar();
```

|Bar| = |x|+|y|+|z|+|DVPtr| =1+1+1 +1= 4 (16 bytes)

```
Library __allocateObject(16),R1
MoveField _Bar_DV,R1.0
Move R1,f
```

Label generated for class type `Bar` during LIR translation



Compile time information for Foo class type generated during LIR translation

20

LIR translation example

```
class A {
  int x;
  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.printi(l);
    return l;
  }
}
```

Translating virtual functions (dispatch tables)

Translating the main function

Translating virtual function calls

Translation for literal strings

Translating `.length` operator

21

LIR program (manual trans.)

```
str1: "y="                                # Literal string in program
_DV_A: [_A_foo]                           # dispatch table for class A
_DV_B: [_B_foo]                           # dispatch table for class B

_A_foo:                                   # int foo(int y)
Move y,R1                                 # int z=y+1;
Add 1,R1
Move R1,z
Return z                                  # return z;

_B_foo:                                   # int foo(int y)
Library __itos(y),R1                      # Library.itos(y);
Library __stringCat(str1,R1),R2           # "y=" + Library.itos(y);
Move this,R3                              # this.s = "y=" + Library.itos(y);
MoveField R2,R3.2
MoveField R3.2,R4
Library __println(R4),Rdummy              # Library.println(s);
Library __stoa(R4),R5                     # int[] sarr = Library.stoa(s);
Move R5,sarr
ArrayLength sarr,R6                       # int l = sarr.length;
Move R6,l
Library __printi(l),Rdummy                # Library.printi(l)
Return l                                  # return l;
# main in A
_ic_main:                                 # A {static void main(string[] args)…}
Library __allocateObject(16),R1           # A p = new B();
MoveField _DV_B,R1.0                      # Update DVPtr of new object
VirtualCall R1.0(y=5),Rdummy              # p.foo(5)
```


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;
}
```

fieldToOffset
x_1 = 1
…
x_n = n

methodToOffset
foo_1 = 0
…
foo_n = n-1

file.lir

```
(1) _DV_A: [foo_1,…,foo_n]
...
(2) VirtualCall R1.7(),R3
(3) MoveField R1.3,R9
```


LIR optimizations

- Aim to reduce number of LIR registers and number of instructions
- Avoid storing variables and constants in registers
- Use accumulator registers
- Reuse "dead" registers
- Weighted register allocation

Avoid storing constants and variables in registers

- Don't allocate target register for each instruction
  - TR[5] = Move 5,Rj
  - TR[x] = Move x,Rk
- For a constant TR[5] = 5
- For a variable TR[x] = x
- TR[x+5] = Move 5,R1
  Add x,R1
  - Assign to register if **both operands** non-registers

25

Accumulator registers

- Use same register for sub-expression and result

```
TR[e1 OP e2]

R1 := TR[e1]
R2 := TR[e2]
R1 := R1 OP R2
```

26

Accumulator registers

```
TR[e1 OP e2]

a+(b*c)
```

```
R1 := TR[e1]
R2 := TR[e2]
R3 := R1 OP R2

Move a,R1
Move b,R2
Mul R1,R2
Move R2,R3
Move c,R4
Add R3,R4
Move R4,R5
```

```
R1 := TR[e1]
R2 := TR[e2]
R1 := R1 OP R2

Move b,R1
Mul c,R1
Add a,R1
```

27

Accumulator registers cont.

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

28

Reuse registers

- Registers have very-limited lifetime
- TR[e1 OP e2] = R1:=TR[e1]
R2:=TR[e2]
R1:=R1 OP R2

Registers from TR[e1] can be reused in TR[e2]

- Solution:
  - Use a stack of LIR registers
  - Stack corresponds to recursive invocations of t := TR[e]
  - All the temporaries on the stack are alive

29

Weighted register allocation

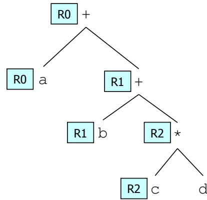
- Sethi & Ullman algorithm
  - Two expression e1, e2 and an operation OP
  - e1,e2 without side-effects
    - function calls
  - TR[e1 OP e2] = TR[e2 OP e1]
  - Weighted register allocation
    - translate heavier sub-tree first

30

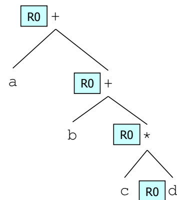
Example

R0 := TR[a+(b+(c*d))]

left child first



right child first



Translation uses all optimizations shown until now uses 3 registers

Managed to save two registers

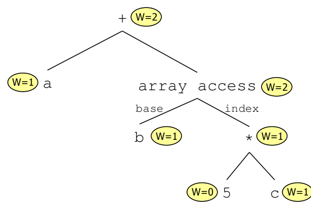
Weighted register allocation

- 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(left) > w(right) then w = left
    - w(right) > w(left) then w = right
    - w(right) = w(left) then w = left + 1
- Choose heavier child as first to be translated
- Have to check that no side-effects exist

Weighted reg. alloc. example

R0 := TR[a+b[5*c]]

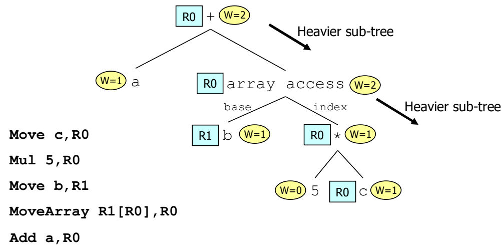
Phase 1: - check absence of side-effects in expression tree
- assign weight to each AST node



Weighted reg. alloc. example

R0 := TR[a+b[5*c]]

Phase 2: use weights to decide on order of translation




```
Move c,R0
Mul 5,R0
Move b,R1
MoveArray R1[R0],R0
Add a,R0
```

34

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

35

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

36

microLIR simulator

- Written by Roman Manevich
- 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

37