Winter 2007-2008
Compiler Construction
T9 – IR part 2 + Runtime organization

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

Announcements

- אין תרגול שבוע הבא עקב נסיעה לחו"ל
- תרגול השלמה נוסף (ואחרון)
  - אפריל 2? **אפריל 9?**
- תרצו תרגול חזרה לפני מבחן?
  - תנו לי תאריך בין מרץ 30 לאפריל 22
- אנא השתתפו בסקר ההוראה – הסקר משפיע

Today

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

- Today:
  - IR
    - LIR language
    - Translating HIR nodes
  - Runtime organization
    - Objects
    - Polymorphism
- Next time:
  - Lowering techniques
  - PA4

Recap

tomatoes + potatoes + carrots → Lexical analyzer → tomatoes,**PLUS**,potatoes,**PLUS**,carrots,EOF → Parser

AddExpr (left right) → AddExpr (left right), LocationExpr id=carrots; AddExpr → LocationExpr id=tomatoes, LocationExpr id=potatoes

Symtab hierarchy

symbol	kind	type
tomatoes	var	int
potatoes	var	int
carrots	var	int

Global type table

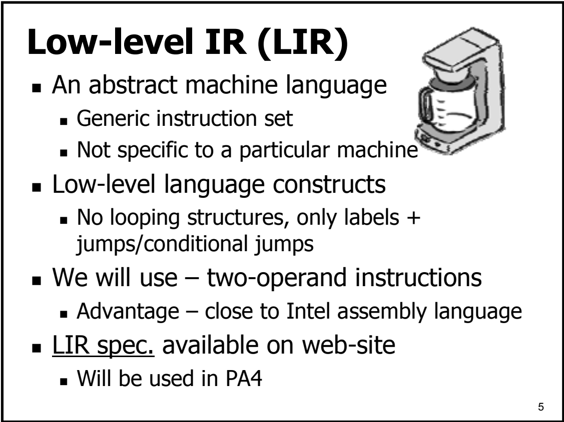
Id	Type obj
int	O1
boolean	O2
Foo	O3

Type checking $\frac{A \vdash E1 : T[]}{A \vdash E1.length : int}$

Additional semantic checks

```
Move tomatoes,R1
Move potatoes,R2
Add R2,R1
...
```

LIR



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

Example

```
x = 42;
while (x > 0) {
  x = x - 1;
}
```

```
Move 42,R1
Move R1,x
_test_label:
Move x,R1
Compare 0,R1
JumpLE _end_label
Move x,R1
Move 1,R2
Sub R1,R2
Move R2,x
Jump _test_label
_end_label:
```

Compare results stored in global compare register (implicit register)

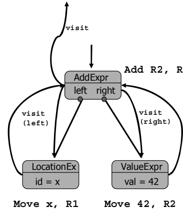
Condition depends on compare register

(warning: code shown is a naïve translation)

7

Translating expressions – example

TR[x + 42]




```
Move x, R1
Move 42, R2
Add R2, R1
```

8

Translation (IR lowering)

- How to translate HIR to LIR?
- Assuming HIR has AST form (ignore non-computation nodes)
  - Define how each HIR node is translated
  - Recursively translate HIR (HIR tree traversal)
- TR[e] = LIR translation of HIR construct e
  - A sequence of LIR instructions
  - Temporary variables = new locations
    - Use temporary variables (LIR registers) to store intermediate values during translation

9

Translating expressions

Binary operations (arithmetic and comparisons)

TR[e1 OP e2]

Fresh virtual (LIR) register generated by translation

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

Shortcut notation to indicate target register NOT LIR instruction

Unary operations

TR[OP e]

```
R1 := TR[e]
R1 := OP R1
```

10

Translating (short-circuit) OR

TR[e1 OR e2]

```
R1 := TR[e1]
Compare 1,R1
JumpTrue _end_label
R2 := TR[e2]
Or R2,R1
_end_label:
```

(OR can be replaced by Move operation since R1 is 0)

11

Translating (short-circuit) AND

TR[e1 AND e2]

```
R1 := TR[e1]
Compare 0,R1
JumpTrue _end_label
R2 := TR[e2]
And R2,R1
_end_label:
```

(AND can be replaced by Move operation since R1 is 1)

12

Translating array and field access

```
TR[e1[e2]]      R1 := TR[e1]
                R2 := TR[e2]
                MoveArray R1[R2], R3

TR[e1.f]        R1 := TR[e1]
                MoveField R1.cf,R3
```

Given class type of e1, need to compute offset of field f: c_f

Need to identify class type of e1 from semantic analysis phase

Constant representing offset of field f in objects of class type of e1 (we shall discuss object representation soon)

Translating statement block

```
TR[s1; s2; … ; sN]      TR[s1]
                        TR[s2]
                        TR[s3]
                        …
                        TR[sN]
```


Translating if-then-else

```
TR[if (e)          R1 := TR[e]
  then s1          Compare 0,R1
  else s2]         JumpTrue _false_label
                   TR[s1]
                   Jump _end_label
                   _false_label:
                   TR[s2]
                   _end_label:
```

Fresh labels generated during translation

Translating if-then

```
TR[if (e) then s]      R1 := TR[e]
                       Compare 0,R1
                       JumpTrue _end_label
                       TR[s1]
                       _end_label:
```

Can avoid generating labels unnecessarily

Translating while

```
TR[while (e) s]      _test_label:
                     R1 := TR[e]
                     Compare 0,R1
                     JumpTrue _end_label
                     TR[s]
                     Jump _test_label
                     _end_label
```


Translating call/return

```
TR[C.foo(e1,…,en)]   R1 := TR[e1]
                     …
                     Rn := TR[en]
                     StaticCall C_foo(x1=R1,…,xn=Rn),R

TR[e1.foo(e2)]       R1 := TR[e1]
                     R2 := TR[e2]
                     VirtualCall R1.cfoo(x=R2),R

TR[return e]         R1 := TR[e]
                     Return R1
```

formal parameter name

actual argument register

Use R_{dummy} for void

Constant representing offset of method f in dispatch table of class type of e1

Lowering implementation

- Define classes for LIR instructions
  - Recommend using the ones from microLIR
- Define `LoweringVisitor`
  - Apply visitor to translate each method
  - Mechanism to generate new LIR registers (keep track of registers for next phase)
  - Mechanism to generate new labels
  - For each node type translation returns
    - List of LIR instructions TR[e]
    - Target register
  - Splice lists: `instructions.addAll`(TR[e]) (input is a tree, output is a flat list)

19

Example revisited

```
TR[
   x = 42;
   while (x > 0) {
   x=x-1;
   }
]

Move 42,R1
Move R1,x
_test_label:
Move x,R1
Compare 0,R1
JumpLE _end_label
TR[x=x-1]
Jump _test_label
_end_label:
```

```
TR[x = 42]
TR[
   while (x > 0) {
   x=x-1;
   }
]

Move 42,R1
Move R1,x
_test_label:
Move x,R1
Compare 0,R1
JumpLE _end_label
Move x,R1
Move 1,R2
Sub R2,R1
Move R1,x
Jump _test_label
_end_label:
```

spliced list for TR[x=x-1]

```
Move 42,R1
Move R1,x
TR[
   while (x > 0) {
   x=x-1;
   }
]
```

(actually a DFS walk)

20

Naïve translation

- Pretend all LIR registers are local variables
  - Increases memory needed for each procedure during runtime
  - Expensive access vs. accessing real registers (spilling)
- Generate fresh LIR register per HIR node
  - Lifetime of registers very limited
- Allow consecutive labels
  - Code bloat

21

Better translation

- Pretend all LIR registers are local variables
  - Ideally: leave it to register allocation phase to store LIR registers in machine registers
  - In this project we ignore this inefficiency
    - Limit register allocation to single LIR instructions
- Optimize LIR register allocation
  - Reuse registers
  - Avoid generating registers for HIR leaves
- Avoid generating consecutive labels
- Optimizations techniques discussed next time

22

Runtime organization

- Representation of basic types
- Representation of allocated objects
  - Class instances
    - Dispatch vectors
  - Strings
  - Arrays
- Procedures
  - Activation records (frames)

Discussed next time (relevant mostly for PA5)

23

Representing data at runtime

- Source language types
  - int, boolean, string, object types, arrays etc.
- Target language types
  - Single bytes, integers, address representation
- Compiler maps source types to some combination of target types
  - Implement source types using target types
- Compiler maps basic operations on source types to combinations of operations on target types
- We will limit our discussion to IC

24

Representing basic types in IC

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

25

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
  - Insert code to check fragile operations (in PA5)
- Only implicit in our case

26

Object types

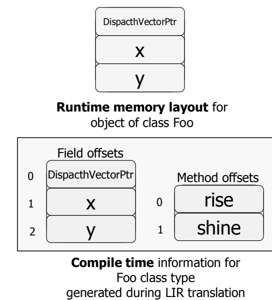
- Basic operations
  - Field selection
    - computing address of field, dereferencing address
  - Method invocation
    - Identifying method to be called, calling it
- How does it look at runtime?

27

Object types

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

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



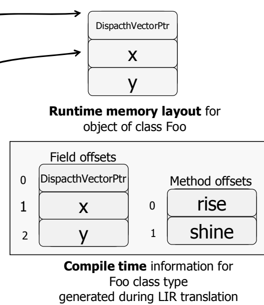
28

Field selection

```
Foo f;
int q;

q = f.x;
```

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



29

Class layout implementation

- Store maps for each `ClassType`
  - From field to offset (starting from 1, since 0 reserved for DispatchVectorPtr)
  - From method to offset

30

Object types and inheritance

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

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

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

DispacthVectorPtr | x | y | z — x, y: prefix from Foo

Runtime memory layout for object of class Bar

Field offsets: 0 DispatchVectorPtr, 1 x, 2 y, 3 z — rise, shine, twinkle (rise, shine: prefix from Foo)

Compile time information for Bar class type

31

Object creation and initialization

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

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

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

Initializes value of DVPtr for Bar

Pointer to Bar: f → DVPtr | x | y | z

Runtime memory layout for object of class Bar

Foo dispatch vector
_Foo_rise
_Foo_shine

Bar dispatch vector
_Bar_rise
_Foo_shine
_Bar_twinkle

Runtime static information

32

Dynamic binding

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

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

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

Foo dispatch vector
_Foo_rise
_Foo_shine

Bar dispatch vector
_Bar_rise
_Foo_shine
_Bar_twinkle

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

33

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();
}
```

Pointer to Bar: f → DVPtr | x | y | z (Object layout; Pointer to Foo inside Bar) → rise | shine (Bar Dispatch vector) → _Bar_rise | _Bar_shine (Method code)

- Vector contains addresses of methods
- Indexed by method-id number
- A method should have same id number in all subclasses

34

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();
}
```

Pointer to Foo: f → DVPtr | x | y (Object layout) → rise | shine (Foo Dispatch vector) → _Foo_rise | _Foo_shine

35

Object creation

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

LIR translation computes object for each class type size of object
|Foo| = |x|+|y|+|z|+|DVPtr| =
1 + 1 + 1 + 1 = 4 (16 bytes)

LIR translation computes object for each class type field offsets and method offsets for DispatchVector

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

Label generated for class type Bar during LIR translation

DispatchVectorPtr | x | y

Runtime memory layout for object of class Foo

Field offsets: 0 DispatchVectorPtr, 1 x, 2 y — Method offsets: 0 rise, 1 shine

Compile time information for Foo class type generated during LIR translation

36

Good luck in the exam!

37