CS412/413

Introduction to Compilers
Radu Rugina

Lecture 16: Intermediate Representation
24 Feb 03

Record Subtyping

- **Width Subtyping:** types of inherited fields must match in the subtype

$$\frac{n \le m}{A \vdash \{a_1: T_1, \dots, a_m: T_m\} <: \{a_1: T_1, \dots, a_n: T_n\}}$$

- **Depth subtyping:** corresponding immutable fields may be subtypes; exact match not required

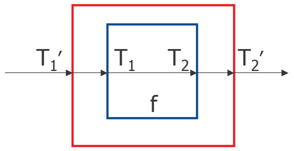
$$\frac{A \;\vdash\; T_i <: T_i' \;{}^{(i \in 1..n)}}{A \vdash \{a_1: T_1 \dots a_n{:}T_n\} <: \{a_1: T_1' \dots a_n: T_n'\}}$$

Depth Subtyping

- Depth subtyping for objects:
  - Mutable components must be type invariant
  - Immutable components may be type covariant
- Immutable components:
  - Methods (but Java is conservative)
  - Constant fields: final in Java

Function Subtyping

- Function subtyping: $T_1 \rightarrow T_2 \;<:\; T_1' \rightarrow T_2'$
- Consider function f of type $T_1 \rightarrow T_2$:



Contravariance/Covariance

- Function argument types may be contravariant
- Function result types may be covariant

$$\frac{T_1' <: T_1 \qquad T_2 <: T_2'}{T_1 \rightarrow T_2 \;<:\; T_1' \rightarrow T_2'}$$

Java Array Subtyping

- Java has array type constructor: for any type T, T [ ] is an array of T's
- Java also has subtype rule:

$$\frac{T_1 <: T_2}{T_1[\,] <: T_2[\,]}$$

- Is this rule safe?

Java Array Subtyping

- Example:

  Elephant <: Animal

  Animal [ ] x;
  Elephant [ ] y;
  x = y;
  x[0] = new Rhinoceros(); // oops!
- Covariant modification: unsound
- Java does run-time check!

Unification

- Some rules more problematic: if
- Rule:

$$\frac{A \mid\text{--}\ E : \text{bool} \qquad A \mid\text{--}\ S_1 : T \qquad A \mid\text{--}\ S_2 : T}{A \mid\text{--}\ \text{if (E) } S_1 \text{ else } S_2 : T}$$

- Problem: if S_1 has type T_1, S_2 has type T_2. Old check: $T_1 = T_2$. New check: need type T. How to unify T_1 , T_2 ?
- Occurs in Java: ?: operator

General Typing Derivation

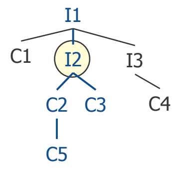
$$\frac{A \vdash E : \text{bool} \qquad \dfrac{A\vdash S_1 : T_1 \quad T_1 <: T}{A \vdash S_1 : T} \qquad \dfrac{A\vdash S_2 : T_2 \quad T_2 <: T}{A \vdash S_2 : T}}{A \vdash \text{if (E) } S_1 \text{ else } S_2 : T}$$

How to pick T ?

Unification

- Idea: unified type is least common ancestor in type hierarchy (least upper bound)
- Partial order of types must be a lattice

if (b) new C5() else new C3() : I2



LUB(C3, C5) = I2

Logic: I2 must be same as or a subtype of any type (e.g. I1) that could be the type of both a value of type C3 and a value of type C5

What if no LUB?

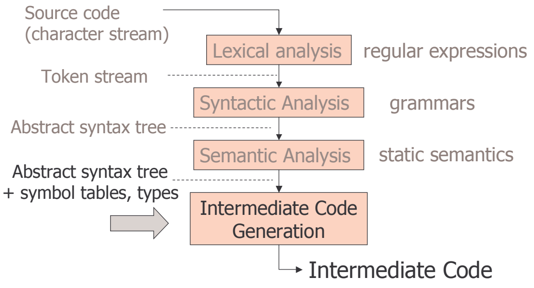
Summary: Semantic Analysis

- Check errors not detected by lexical or syntax analysis
- Scope errors:
  - Variables not defined
  - Multiple declarations
- Type errors:
  - Assignment of values of different types
  - Invocation of functions with different number of parameters or parameters of incorrect type
  - Incorrect use of return statements

Semantic Analysis

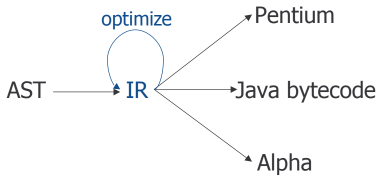
- Type checking
  - Use type checking rules
  - Static semantics = formal framework to specify type-checking rules
- There are also control flow errors:
  - Must verify that a break or continue statement is always enclosed by a while (or for) statement
  - Java: must verify that a break X statement is enclosed by a for loop with label X
  - Can easily check control-flow errors by recursively traversing the AST

Where We Are



Intermediate Code

- IR = Intermediate Representation
- Allows language-independent, machine-independent optimizations and transformations



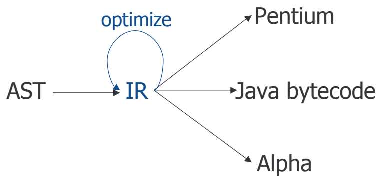
What Makes a Good IR?

- Easy to translate from AST
- Easy to translate to assembly
- Narrow interface: small number of node types (instructions)
  - Easy to optimize
  - Easy to retarget

AST (>40 node types)
↓
IR (13 node types)
↓
Pentium (>200 opcodes)

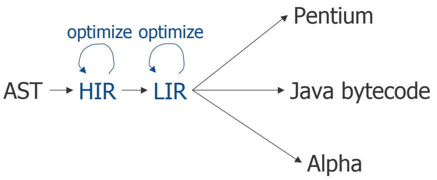
Multiple IRs

- Some optimizations require high-level structure
- Others more appropriate on low-level code



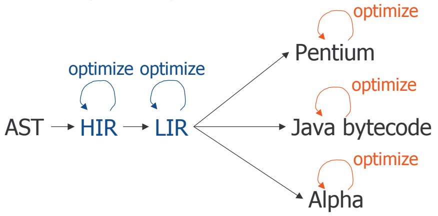
Multiple IRs

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



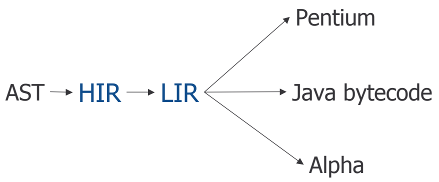
Machine Optimizations

- … some other optimizations take advantage of the features of the target machine
- Machine-specific optimizations



Next Lectures

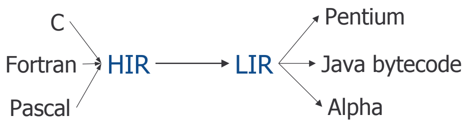
- Next few lectures: intermediate representation
- Optimizations covered later



Multiple IRs

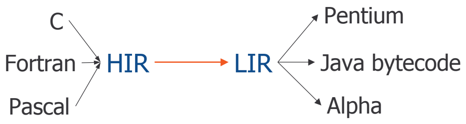
- Usually two IRs:

High-level IR	Low-level IR
Language-independent (but closer to language)	Machine independent (but closer to machine)



Multiple IRs

- Another benefit: a significant part of the translation from high-level to low-level is
  - Language-independent
  - Machine-independent



High-Level IR

- High-level intermediate representation is essentially the AST
  - Must be expressive for all input languages
- Preserves high-level language constructs
  - Structured control flow: if, while, for, switch, etc.
  - variables, methods
- Allows high-level optimizations based on properties of source language (e.g. inlining)

Low-Level IR

- Low-level representation is essentially an abstract machine
- Has low-level constructs
  - Unstructured jumps, instructions
- Allows optimizations specific to these constructs (e.g. register allocation, branch prediction)

Low-Level IR

- Alternatives for low-level IR:
  - Three-address code or quadruples (Dragon Book):
    a = b OP c
  - Tree representation (Tiger Book)
  - Stack machine (like Java bytecode)
- Advantages:
  - Three-address code: easier dataflow analysis
  - Tree IR: easier instruction selection
  - Stack machine: easier to generate

Three-Address Code

- In this class: three-address code

  a = b OP c

- Has at most three addresses (may have fewer)
- Also named quadruples because can be represented as: (a, b, c, OP)
- Example:

```
a = (b+c)*(-e);          t1 = b + c
                         t2 = – e
                         a = t1 * t2
```

Low IR Instructions

- Assignment instructions:
  - – Binary operations: a = b OP c
    - Arithmetic, logic, comparisons
  - – Unary operation a = OP b
    - Arithmetic, logic
  - – Copy instruction: a = b
  - – Load /store: a = *b, *a = b
  - – Other data movement instructions

Low IR Instructions (Ctd)

- Flow of control instructions:
  - – label L : label instruction
  - – jump L : Unconditional jump
  - – cjump a L : conditional jump
- Function call
  - – call f(a_1, …., a_n)
  - – a = call f(a_1, …., a_n)
  - – Is an extension to quads
- … IR describes the Instruction Set of an abstract machine

Temporary Variables

- The operands in the quadruples can be:
  - – Program variables
  - – Integer constants
  - – Temporary variables
- Temporary variables = new locations
  - – Use temporary variables to store intermediate values

Arithmetic / Logic Instructions

- Abstract machine supports a variety of different operations

  a = b OP c   a = OP b

- Arithmetic operations: ADD, SUB, DIV, MUL
- Logic operations: AND, OR, XOR
- Comparisons: EQ, NEQ, LE, LEQ, GE, GEQ
- Unary operations: MINUS, NEG

Data Movement

- Copy instruction: a = b
- Load/store instructions:

  a = *b   *a = b

  - – Models a load/store machine
- Address-of instruction: a = &b
- Array accesses:

  a = b[i]   a[i] = b

- Field accesses:

  a = b.f   a.f = b

Branch Instructions

- Label instruction:

  label L
- Unconditional jump: go to statement after label L

  jump L
- Conditional jump: test condition variable a; if true, jump to label L

  cjump a L
- Alternative: two conditional jumps:

  tjump a L   fjump a L

Call Instruction

- Supports function call statements

  call f(a_1, …, a_n)
- … and function call assignments:

  a = call f(a_1, …, a_n)
- No explicit representation of argument passing, stack frame setup, etc.

Example

```
n = 0;
while (n < 10)  {
   n = n + 1
}
```

⇨

```
n = 0
label test
t2 = n < 10
t3 = not t2
cjump t3 end
label body
n = n + 1
jump test
label end
```

Another Example

```
m = 0;
if (c  == 0) {
   m = m+ n*n;
} else {
   m = m + n;
}
```

⇨

```
m = 0
t1 = c == 0
cjump t1 trueb
m = m+n
jump end
label trueb
t2 = n * n
m = m + t2
label end
```

How To Translate?

- May have nested language constructs
  - Nested if and while statements
- Need an algorithmic way to translate
- Solution:
  - Start from the AST representation
  - Define translation for each node in the AST
  - Recursively translate nodes in the AST