CS421/413

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

\[ n \leq m \]

\[ A \vdash \{ a_1 : T_1, \ldots, a_n : T_n \} \leq \{ a_1 : T'_1, \ldots, a_n : T'_n \} \]

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

\[ A \vdash \{ a_1 : T_1, \ldots, a_n : T_n \} \leq \{ a_1 : T'_1, \ldots, a_n : T'_n \} \]

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: \( T_1 \rightarrow T_2 \leq T'_1 \rightarrow T'_2 \)
- Consider function \( f \) of type \( T_1 \rightarrow T_2 \):

\[
\begin{array}{c}
T_1' \\
\downarrow \quad f \\
T_1 \\
\downarrow \quad T_2 \\
T_2'
\end{array}
\]

Contravariance/Covariance

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

\[
\begin{align*}
T_1' &\leq T_2' \\
T_2 &\leq T_2'
\end{align*}
\]

\[
T_1 \rightarrow T_2 \leq T_1' \rightarrow T_2'
\]

Java Array Subtyping

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

\[
T_3 \leq T_2 \\
T_3[\ ] \leq T_2[\ ]
\]

- 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:  
  
  \[
  \begin{align*}
  A &\vdash E : \text{bool} \\
  A &\vdash S_1 : T \\
  A &\vdash S_2 : T
  \end{align*}
  \]

  
  \[
  A \vdash \text{if} ( E ) S_1 \text{ else } S_2 : T
  \]
- Problem: if `S_1` has type `T_1`, `S_2` has type `T_2`. Old check: `T_1`  
- New check: need type `T`. How to unify `T_1`, `T_2`?  
- Occurs in Java: `?:` operator

**General Typing Derivation**

\[
\begin{align*}
A &\vdash S_1 : T_1\prec T \\
A &\vdash S_2 : T_2\prec T
\end{align*}
\]

\[
A \vdash ( E ) S_1 \text{ else } S_2 : T
\]

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`

\[
\text{LUB(C3, C5)} = I2
\]

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

- Source code (character stream)
- Token stream
- Abstract syntax tree

**Lexical Analysis**
- Regular expressions

**Syntactic Analysis**
- Grammars
- Abstract syntax tree + symbol tables, types

**Semantic Analysis**
- Static semantics

**Intermediate Code Generation**

Intermediate Code

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

```
AST → IR
```

- **Optimize**
- Pentium
- Java bytecode
- *Alpha*

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 re-target

```
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

```
AST → IR
```

- **Optimize**
- Pentium
- Java bytecode
- *Alpha*

Multiple IRs

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

```
AST → HIR → LIR
```

- **Optimize**
- Pentium
- Java bytecode
- *Alpha*

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

AST → HIR → LIR → Java bytecode

- Pentium
- Alpha

Multiple IRs

- Usually two IRs:
  - High-level IR
    - Language-independent
    - (but closer to language)
  - Low-level IR
    - Machine independent
    - (but closer to machine)

C

Pentium

Fortran

HIR

LIR

Java bytecode

Pascal

Alpha

Multiple IRs

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

C

Pentium

Fortran

HIR

LIR

Java bytecode

Pascal

Alpha

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
  \[ a = b \text{ OP } c \]
- Has at most three addresses (may have fewer)
- Also named quadruples because can be represented as: \((a, b, c, \text{ OP})\)
- Example:
  \[ a = (b+c)^*(-e); \quad t1 = b + c \]
  \[ t2 = -e \]
  \[ a = t1 \ast t2 \]

Low IR Instructions

- Assignment instructions:
  - Binary operations: \(a = b \text{ OP } c\)
    - Arithmetic, logic, comparisons
  - Unary operation \(a = \text{ OP } b\)
    - Arithmetic, logic
  - Copy instruction: \(a = b\)
  - Load/store: \(a = \ast b, \ast a = b\)
  - 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 \(f(a_1, \ldots, a_n)\)
  - \(a = \text{ call } f(a_1, \ldots, a_n)\)
  - 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
  \[ a = b \text{ OP } c \quad a = \text{ OP } b \]
- 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:
  \[ a = \ast b, \ast a = b\]
  - Models a load/store machine
- Address-of instruction: \(a = \& b\)
- Array accesses:
  \[ a = b[i] \quad a[i] = b\]
- Field accesses:
  \[ a = b.f \quad a.f = b\]
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(a1, ..., an)
- ... and function call assignments: a = call f(a1, ..., an)
- No explicit representation of argument passing, stack frame setup, etc.

Example

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

Another Example

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

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