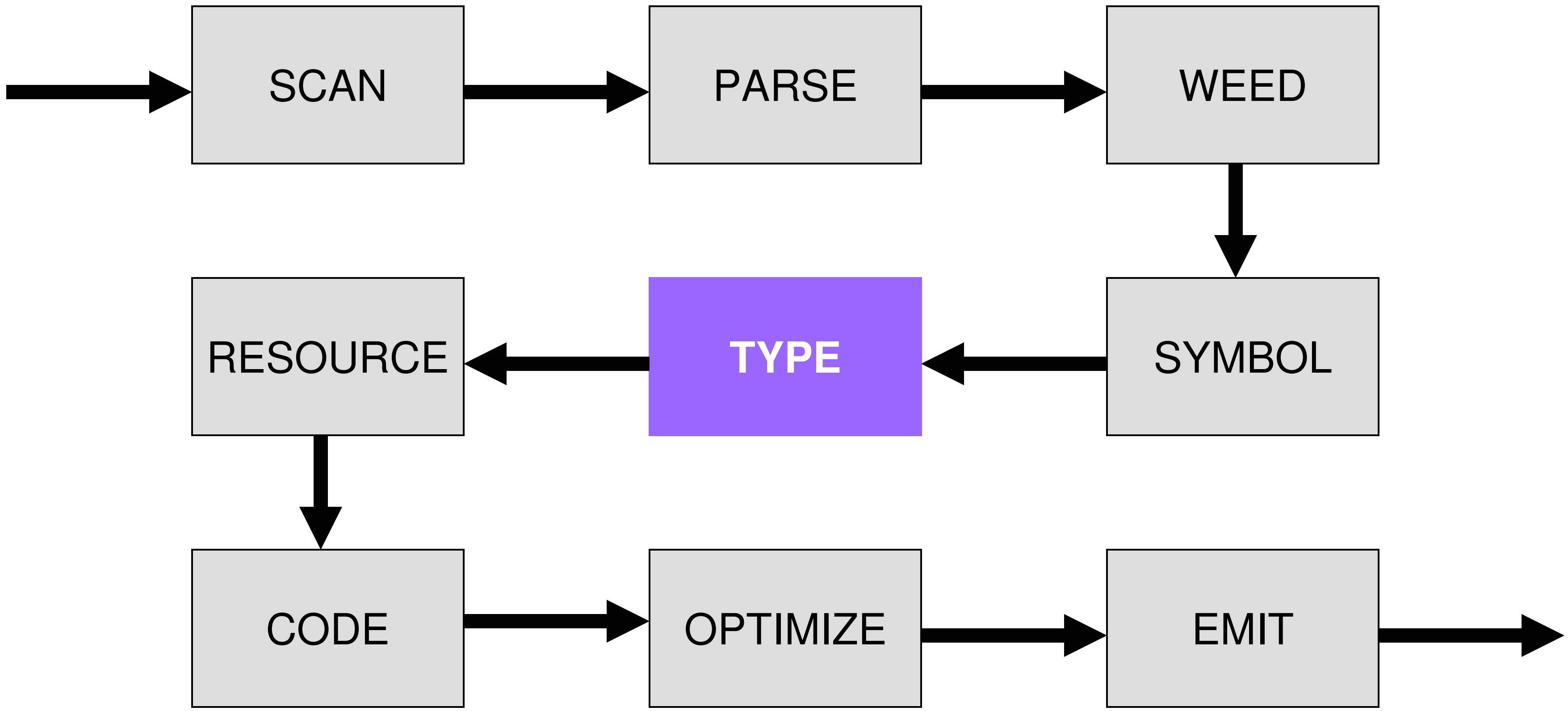
Compiler

Type Checking

© Copyright 2005, Matthew B. Dwyer and Robby. The syllabus and lectures for this course are copyrighted materials and may not be used in other course settings outside University of Nebraska-Lincoln and Kansas State University in their current form or modified form without express written permission of one of the copyright holders. During this course, students are prohibited from selling notes to or being paid for taking notes by any person or commercial firm without the express written permission of one of the copyright holders.

Compiler Architecture



Role of Type Checker

- determine the types of all expressions;
- check that values and variables are used correctly; and
- resolve certain ambiguities by transforming the program.

Some languages have no type checker.

What is a type?

- A *type* defines a set of possible values
- The SJ/ESJC types are:
  - `void` the empty type;
  - `int` the integers;
  - `boolean` { true, false}; and
  - objects of a class `C`.
- Plus an artificial type:
  - `null` constant.

Types as Invariants

Type annotations

- `int x;`
- `Cons y;`

specify an invariant on run-time behavior

- `x` will always contain an integer value
- `y` will always contain `null` or a reference to an object of type `Cons`

Pretty weak language for defining invariants

Type Correctness

- A program is *type correct* if the type annotations are valid invariants.
- Type correctness is undecidable:

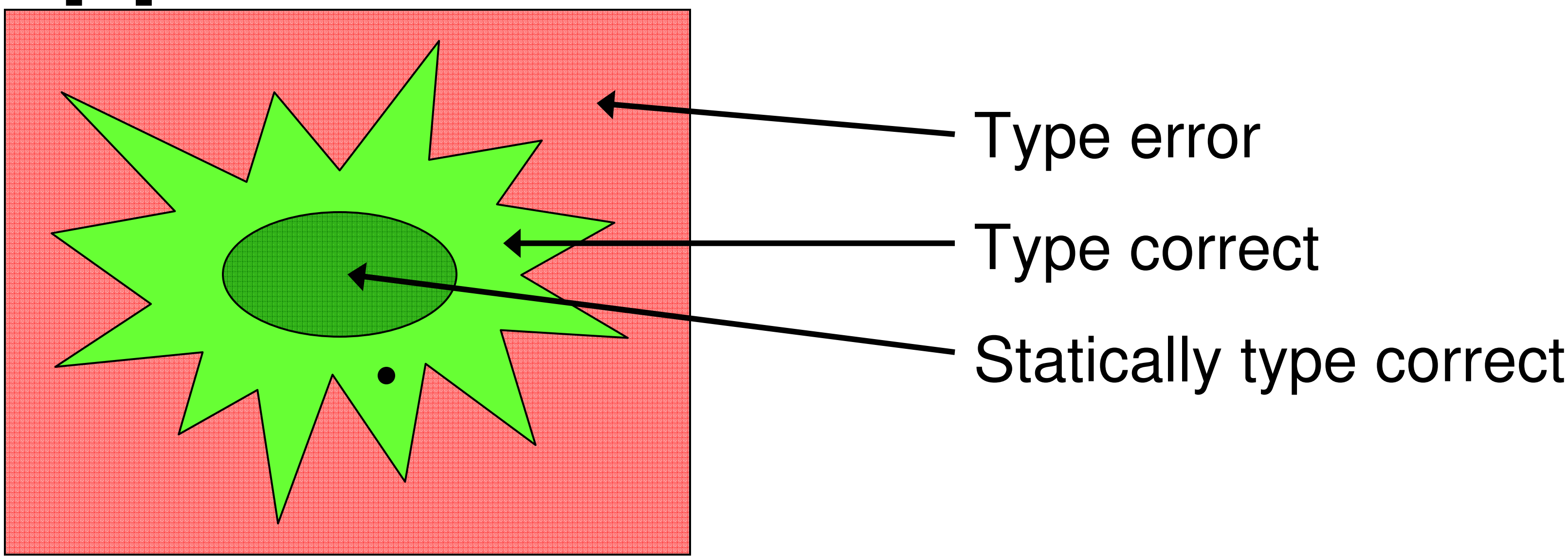
```
int x;
int j;
x = 0;
// get j from input
TM(j);
x = true;
```

- where `TM(j)` simulates the *j*'th Turing machine on empty input.
- The program is type correct if and only if `TM(j)` does not halt on empty input.

Static Typing

- A program is *statically* type correct if it satisfies some type rules.
- The type rules are chosen to be:
  - simple to understand;
  - efficient to decide; and
  - conservative with respect to type correctness.
- Type rules are rarely canonical.

Type Systems are Approximate



There will always be programs that are type correct, but are unfairly rejected by the static type checker.

For You To Do

- Can you think of a program that is type correct, but will be rejected by a type checker?

Rejected Type Correct Program

```
int x;
x = 87;
if (false) x = true;
```

Type Rules

Three ways to specify rules

- prose

  `The argument to the sqrt function must be of type int;  the result is of type real.`

- constraints on type variables

  $$\texttt{sqrt(x):}\quad [\![\texttt{sqrt(x)}]\!] = \texttt{real} \wedge [\![\texttt{x}]\!] = \texttt{int}$$

- logical rules

  $$\frac{\mathcal{S} \vdash \texttt{x} : \texttt{int}}{\mathcal{S} \vdash \texttt{sqrt(x)} : \texttt{real}}$$

Kinds of Rules

- Declarations
  - When a variable is introduced
- Propagations
  - When an expression's type is used to determine the type of an enclosing expression
- Restrictions
  - When the type of an expression is constrained by its usage context

Judgements

- Type judgement for statements

$$L, C, M, V \vdash S$$

- Means that S is statically type correct with:
  - class library L;
  - current class C;
  - current method M; and
  - variables V

Judgements

- Type judgement for expressions

$$L, C, M, V \vdash E : \tau$$

- Means that E is statically type correct and has type τ
- The tuple

$$L, C, M, V$$

- is an abstraction of the symbol table

Statement Sequences

$$\frac{L, C, M, V \vdash S_1 \qquad L, C, M, V \vdash S_2}{L, C, M, V \vdash S_1\ S_2}$$

$$\frac{L, C, M, V[\mathtt{x} \mapsto \tau] \vdash S}{L, C, M, V \vdash \tau\ \ \mathtt{x}\,\mathtt{;}\, S}$$

…given by the order of statement visitations in a block

Return Statements

$$\frac{\mathit{type}(L, C, M) = \texttt{void}}{L, C, M, V \vdash \texttt{return}}$$

$$\frac{L, C, M, V \vdash E : \tau \quad \mathit{type}(L, C, M) = \sigma \quad \sigma := \tau}{L, C, M, V \vdash \texttt{return}\ E}$$

Return Statements

```
@Override public boolean visit(ReturnStatement node) {
  Expression e = node.getExpression();
  if (methodReturnType == tf.Void && e != null) {
    throw new Error(node, "Unexpected return's expression in \""
                          + node + "\"");
  } else if (methodReturnType != tf.Void && e == null) {
    throw new Error(node, "Expecting a return's expression in \""
                          + node + "\"");
  } else if (methodReturnType != tf.Void && e != null) {
    e.accept(this);
    Type t = getResult();
    if (t != methodReturnType) {
      throw new Error(node, "Expecting " + methodReturnType.name
                            + " return expression in \"" + node
                            + "\"");
    }
  }
  return super.visit(node);
}
```

…assignment compatibility in SJ is simple!

Expression Statements

$$\frac{L, C, M, V \vdash E : \tau}{L, C, M, V \vdash E}$$

```java
@Override public boolean visit(ExpressionStatement node) {
  Expression e = node.getExpression();
  e.accept(this);
  if (e instanceof Assignment) {
    // assignment should not have a resulting type.
    assert getResult() == null;
  } else if (node.getExpression() instanceof MethodInvocation) {
    // method invocation's result can be any type (including void)
    // so we can ignore it.
    getResult();
  } else { // throw error }
  return false; }
```

If Statements

$$\frac{L, C, M, V \vdash E : \texttt{boolean} \qquad L, C, M, V \vdash S}{L, C, M, V \vdash \texttt{if (}E\texttt{)}\ S}$$

```
@Override public boolean visit(IfStatement node) {
  node.getExpression().accept(this);
  if (getResult() != tf.Boolean) { // throw error }
  node.getThenStatement().accept(this);
  node.getElseStatement().accept(this);
  return false;
}
```

Variables

$$\frac{V(\mathbf{x}) = \tau}{L, C, M, V \vdash \mathbf{x} : \tau}$$

```java
@Override public boolean visit(SimpleName node) {
  ASTNode parent = node.getParent();
  if (parent instanceof Expression || parent instanceof Statement) {
    Object o = symbolMap.get(node);
    if (o instanceof FieldDeclaration) {
      FieldDeclaration fd = (FieldDeclaration) o;
      setResult(node, convertType(node, fd.getType()));
    } else if (o instanceof SingleVariableDeclaration) {
      SingleVariableDeclaration svd = (SingleVariableDeclaration) o;
      setResult(node, convertType(node, svd.getType()));
    } else if (o instanceof VariableDeclarationStatement) {
      VariableDeclarationStatement vds = (VariableDeclarationStatement) o;
      setResult(node, convertType(node, vds.getType()));
    } else { // throw error } return false; }
```

Assignment

$$\frac{L,C,M,V \vdash \mathtt{x} : \tau \quad L,C,M,V \vdash E : \sigma \quad \tau :\!= \sigma}{L,C,M,V \vdash \mathtt{x} \mathtt{:=} E}$$

```java
@Override public boolean visit(Assignment node) {
  node.getLeftHandSide().accept(this);
  Type lhsType = getResult();
  node.getRightHandSide().accept(this);
  Type rhsType = getResult();
  if (lhsType != rhsType) {
    throw new Error(node, "Type mismatch in \"" + node + "\": "
                       + lhsType + " = " + rhsType);
  }
  // no need to set the type result for assignments since
  // assignments in StaticJava are statements,
  // i.e., they are evaluated for their side-effects.
  return false;
}
```

Minus (Arithmetic Expresion)

$$\frac{L,C,M,V \vdash E_1 : \mathtt{int} \qquad L,C,M,V \vdash E_2 : \mathtt{int}}{L,C,M,V \vdash E_1\text{-}E_2 : \mathtt{int}}$$

```
@Override public boolean visit(InfixExpression node) {
  node.getLeftOperand().accept(this);
  Type lhsType = getResult();
  node.getRightOperand().accept(this);
  Type rhsType = getResult();
  InfixExpression.Operator op = node.getOperator();
  if (… || op == InfixExpression.Operator.MINUS) {
    if (lhsType != tf.Int) { // throw error }
    if (rhsType != tf.Int) { // throw error }
    setResult(node, tf.Int);
  } … }
```

Equality

$$\frac{\begin{array}{c} L,C,M,V \vdash E_1 : \tau_1 \\ L,C,M,V \vdash E_2 : \tau_2 \\ \tau_1 := \tau_2 \vee \tau_2 := \tau_1 \end{array}}{L,C,M,V \vdash E_1 \texttt{==} E_2 : \texttt{boolean}}$$

Method Invocation

$$\frac{\begin{array}{l} L, C, M, V \vdash E : \sigma \\ L, C, M, V \vdash E_i : \sigma_i \\ \mathit{type}(L, \sigma, \mathtt{m}) = \tau \\ \mathit{argtype}(L, \sigma, \mathtt{m}, i) := \sigma_i \end{array}}{L, C, M, V \vdash E\,\mathtt{.m(}E_1, \ldots, E_n\mathtt{)} : \tau}$$

Method Invocation (1)

```
@Override public boolean visit(MethodInvocation node) {
  String className = node.getExpression() == null ? this.className
                : ((SimpleName) node.getExpression()).getIdentifier();
  String methodName = node.getName().getIdentifier();
  int numOfArgs = node.arguments().size();
  Type[] argTypes = new Type[numOfArgs];
  for (int i = 0; i < numOfArgs; i++) {
    ((Expression) node.arguments().get(i)).accept(this);
    argTypes[i] = getResult();
  }
  MethodDeclaration md = (MethodDeclaration) symbolMap.get(node);
  if (md == null) {
    Method m = resolveMethod(node, className, methodName, argTypes);
    typeCheckMethodInvocation(node, className, methodName, argTypes, m);
  } else {
    typeCheckMethodInvocation(node, className, methodName, argTypes, md);
  }
  return false; }
```

Method Invocation (2)

```
protected void typeCheckMethodInvocation(MethodInvocation node,
    String className, String methodName, Type[] argTypes,
    MethodDeclaration md) {
  int numOfParams = md.parameters().size();
  if (argTypes.length != numOfParams) { // throw error }
  for (int i = 0; i < numOfParams; i++) {
    Type t = convertType(node, ((SingleVariableDeclaration) md
                              .parameters().get(i)).getType());
    if (t != argTypes[i]) { // throw error }
  }
  Type returnType = convertType(node, md.getReturnType2());
  setResult(node, returnType);
}
```

Kinds of Type Rules

- Axioms (i.e., given facts)

$$L, C, M, V \vdash \texttt{this} : C$$

- Predicates (i.e., boolean tests on type vars)

$$\tau := \tau'$$

- Inferences (i.e., given x we can conclude y)

$$\frac{L,C,M,V \vdash E_1 : \texttt{int} \qquad L,C,M,V \vdash E_2 : \texttt{int}}{L,C,M,V \vdash E_1\texttt{-}E_2 : \texttt{int}}$$

Type Proofs

- A type checker constructs a proof of the type correctness of a given program
- A *type proof* is a tree in which
  - nodes are inferences; and
  - leaves are axioms or true predicates.
- A program is statically type correct if and only if it is the root of a type proof tree
  - A type proof is a *trace* of a successful run of the type checker

A Type Proof

- $L,C,M,V \vdash$ `int x; int y; y = x;`

$$\dfrac{\dfrac{\dfrac{\dfrac{V[\mathtt{x}\mapsto\mathtt{int}][\mathtt{y}\mapsto\mathtt{int}](\mathtt{y})=\mathtt{int}}{\mathcal{S}\vdash\mathtt{y}:\mathtt{int}} \quad \dfrac{V[\mathtt{x}\mapsto\mathtt{int}][\mathtt{y}\mapsto\mathtt{B}](\mathtt{x})=\mathtt{int}}{\mathcal{S}\vdash\mathtt{x}:\mathtt{int}}}{L,C,M,V[\mathtt{x}\mapsto\mathtt{int}][\mathtt{y}\mapsto\mathtt{int}]\vdash\mathtt{y\text{=}x;}}\ \mathtt{int}:=\mathtt{int}}{L,C,M,V[\mathtt{x}\mapsto\mathtt{int}]\vdash\mathtt{int\ y;\ y\text{=}x;}}}{L,C,M,V\vdash\mathtt{int\ x;\ int\ y;\ y\text{=}x;}}$$

Java Type Checking — this

$$L, C, M, V \vdash \texttt{this} : C$$

Java Type Checking – Cast Expression

$$\frac{L,C,M,V \vdash E : \tau \quad \tau \leq \texttt{C} \vee \texttt{C} \leq \tau}{L,C,M,V \vdash \texttt{(C)}E : \texttt{C}}$$

Java Type Checking — instanceof Expression

$$\frac{L, C, M, V \vdash E : \tau \qquad \tau \leq \texttt{C} \vee \texttt{C} \leq \tau}{L, C, M, V \vdash E \ \texttt{instanceof C} : \texttt{boolean}}$$

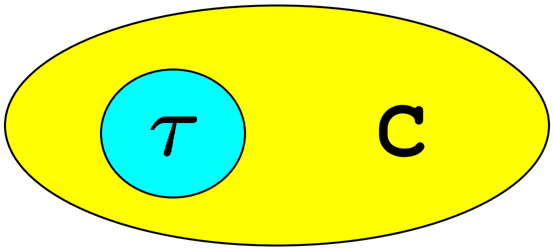
For You To Do

Think about why the predicate

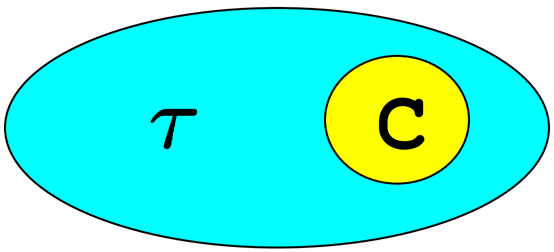
$$\tau \leq \mathtt{C} \vee \mathtt{C} \leq \tau$$

is used for `(C)E` and `E instanceof C`?

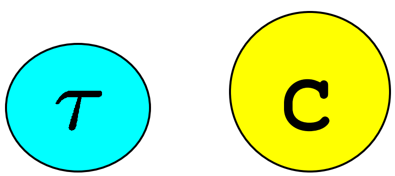
Java Type Checking – Sub-type Testing



succeeds if $\tau \leq C$



don't know if $C \leq \tau$



fails if $\tau \not\leq C \land C \not\leq \tau$

Java Type Checking –
A Type Proof

$$
\dfrac{\dfrac{\dfrac{\dfrac{V[\mathtt{x}\mapsto\mathtt{A}][\mathtt{y}\mapsto\mathtt{B}](\mathtt{y})=\mathtt{B}}{\mathcal{S}\vdash\mathtt{y}:\mathtt{B}} \qquad \dfrac{\dfrac{V[\mathtt{x}\mapsto\mathtt{A}][\mathtt{y}\mapsto\mathtt{B}](\mathtt{x})=\mathtt{A}}{\mathcal{S}\vdash\mathtt{x}:\mathtt{A}} \quad \mathtt{A}\leq\mathtt{B}\vee\mathtt{B}\leq\mathtt{A}}{\mathcal{S}\vdash(\mathtt{B})\mathtt{x}:\mathtt{B}} \quad \mathtt{B}:=\mathtt{B}}{L,C,M,V[\mathtt{x}\mapsto\mathtt{A}][\mathtt{y}\mapsto\mathtt{B}]\vdash\mathtt{y=(B)x;}}}{L,C,M,V[\mathtt{x}\mapsto\mathtt{A}]\vdash\mathtt{B\ y;\ \ y=(B)x;}}}{L,C,M,V\vdash\mathtt{A\ x;\ \ B\ y;\ \ y=(B)x;}}
$$

where $\mathcal{S}=L,C,M,V[\mathtt{x}\mapsto\mathtt{A}][\mathtt{y}\mapsto\mathtt{B}]$ and $\mathtt{B}\leq\mathtt{A}$

Java Type Checking — Plus

$$\frac{L,C,M,V \vdash E_1 : \texttt{int} \quad L,C,M,V \vdash E_2 : \texttt{int}}{L,C,M,V \vdash E_1\texttt{+}E_2 : \texttt{int}}$$

$$\frac{L,C,M,V \vdash E_1 : \texttt{String} \quad L,C,M,V \vdash E_2 : \tau}{L,C,M,V \vdash E_1\texttt{+}E_2 : \texttt{String}}$$

$$\frac{L,C,M,V \vdash E_1 : \tau \quad L,C,M,V \vdash E_2 : \texttt{String}}{L,C,M,V \vdash E_1\texttt{+}E_2 : \texttt{String}}$$

The + operator is *overloaded*

Java Type Checking – Coercion

- A coercion is a conversion function that is inserted automatically by the compiler
- For example

```
"abc" + 17 + x
```

is transformed into

```
"abc" + (new Integer(17).toString())
      + x.toString()
```

For You To Do

Could a rule like

$$\frac{L,C,M,V \vdash E_1 : \tau \quad L,C,M,V \vdash E_2 : \sigma}{L,C,M,V \vdash E_1 \texttt{+} E_2 : \texttt{String}}$$

be included to handle coercions?