Type Checking

Matthew Dwyer
324E Nichols Hall

dwyer@cis.ksu.edu
http://www.cis.ksu.edu/~dwyer
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 JOOS types are:
  – `void` the empty type;
  – `int` the integers;
  – `boolean` \{ *true*, *false* \}; and
  – objects of a class \( C \) or any subclass.
• Plus an artificial type:
  – `polynull`
  – which is the type of the polymorphic `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` or one of its sub-classes.

Pretty weak language for defining invariants
Type Correctness

- A program is *type correct* if the type annotations are valid invariants.
- Type correctness is undecidable:
  ```c
  int x;
  int j;
  x = 0;
  scanf("%i", &j);
  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?
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
  \( \text{sqrt}(x) : \llbracket \text{sqrt}(x) \rrbracket = \text{real} \land \llbracket x \rrbracket = \text{int} \)

• logical rules
  \[
  \frac{S \vdash x : \text{int}}{S \vdash \text{sqrt}(x) : \text{real}}
  \]
Kinds of Rules

• Declarations
  – When a variable is introduced

• Propogations
  – 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 \( \tau \)

• The tuple

\[ L, C, M, V \]

• is an abstraction of the symbol table
Statement Sequences

\[
L, C, M, V \vdash S_1 \quad \text{and} \quad L, C, M, V \vdash S_2 \\
\frac{}{L, C, M, V \vdash S_1 \, S_2}
\]

\[
L, C, M, V[x \mapsto \tau] \vdash S \\
\frac{}{L, C, M, V \vdash \tau \, x; S}
\]
case sequenceK:
  typeImplementationSTATEMENT(
    s->val.sequenceS.first,
    class,returntype);
  typeImplementationSTATEMENT(
    s->val.sequenceS.second,
    class,returntype);
  break;
.
.
.
.
.
.
.
.
.

case localK:
  break;
Return Statements

\[
\text{type}(L,C,M) = \text{void} \\
\quad \quad L,C,M,V \vdash \text{return}
\]

\[
L,C,M,V \vdash E : \tau \quad \text{type}(L,C,M) = \sigma \quad \sigma := \tau \\
\quad \quad L,C,M,V \vdash \text{return } E
\]
Return Statements

case returnK:
    if (s->val.returnS!=NULL) {
        typeImplementationEXP(s->val.returnS,class);
    }
    if (returntype->kind==voidK && s->val.returnS!=NULL) {
        reportError("return value not allowed",s->lineno);
    }
    if (returntype->kind!=voidK && s->val.returnS==NULL) {
        reportError("return value expected",s->lineno);
    }
    if (returntype->kind!=voidK && s->val.returnS!=NULL) {
        if (!assignTYPE(returntype,s->val.returnS->type)) {
            reportError("illegal type of expression",
                s->lineno);
        }
    }
    break;
Assignment Compatibility

- `int := int;`
- `boolean := boolean;`
- `C := polynull;` and
- `C := D, if D ≤ C.`

```c
int assignTYPE(TYPE *s, TYPE *t)
{
    if (s->kind==refK && t->kind==polynullK) return 1;
    if (s->kind!=t->kind) return 0;
    if (s->kind==refK) return subClass(t->class,s->class);
    return 1;
}
```
Expression Statements

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

case expK:
    typeImplementationEXP(s->val.expS,class);
break;
If Statements

\[
L, C, M, V \vdash E : \text{boolean} \quad L, C, M, V \vdash S \\
\]

\[
L, C, M, V \vdash \text{if } (E) \ S
\]

case ifK:
    typeImplementationEXP(s->val.ifS.condition, class);
    checkBOOL(s->val.ifS.condition->type, s->lineno);
    typeImplementationSTATEMENT(s->val.ifS.body, 
                               class, returntype);
    break;
Variables

\[
\begin{equation}
V(x) = \tau
\end{equation}
\]

\[
L, C, M, V \vdash x : \tau
\]

case idK:
e->type = typeVar(e->val.idE.idsym);
break;
Assignment

\[
\begin{align*}
L, C, M, V & \vdash x : \tau & L, C, M, V & \vdash E : \sigma & \tau & ::= \sigma \\
L, C, M, V & \vdash x := E : \tau
\end{align*}
\]

case assignK:
    e->type = typeVar(e->val.assignE.leftsym);
    typeImplementationEXP(e->val.assignE.right,class);
    if (!assignTYPE(e->type,e->val.assignE.right->type)) {
        reportError("illegal assignment",e->lineno);
    }
    break;
Minus

\[ L, C, M, V \vdash E_1 : \text{int} \quad L, C, M, V \vdash E_2 : \text{int} \]

\[ L, C, M, V \vdash E_1 - E_2 : \text{int} \]

case minusK:
    typeImplementationEXP(e->val.minusE.left,class);
    typeImplementationEXP(e->val.minusE.right,class);
    checkINT(e->val.minusE.left->type,e->lineno);
    checkINT(e->val.minusE.right->type,e->lineno);
    e->type = intTYPE;
    break;
Equality

\[ L, C, M, V \vdash E_1 : \tau_1 \]
\[ L, C, M, V \vdash E_2 : \tau_2 \]
\[ \tau_1 ::= \tau_2 \lor \tau_2 ::= \tau_1 \]

\[ L, C, M, V \vdash E_1 == E_2 : \text{boolean} \]
case eqK:
    typeImplementationEXP(e->val.eqE.left,class);
    typeImplementationEXP(e->val.eqE.right,class);
    if (!assignTYPE(e->val.eqE.left->type,
                   e->val.eqE.right->type) &&
        !assignTYPE(e->val.eqE.right->type,
                   e->val.eqE.left->type)) {
        reportError("arguments for = have wrong types",
                    e->lineno);
    }
    e->type = boolTYPE;
    break;
case thisK:
    if (class==NULL) {
        reportError("'this' not allowed here",e->lineno);
    }
    e->type = classTYPE(class);
    break;
cast expression

\[
L, C, M, V \vdash E : \tau \quad \tau \leq C \lor C \leq \tau
\]

\[
L, C, M, V \vdash (C)E : C
\]
case castK:
    typeImplementationEXP(e->val.castE.right,class);
    e->type = makeTYPEextref(e->val.castE.left,
                              e->val.castE.leftsym);
    if (e->val.castE.right->type->kind!=refK && ...!=polynullK) {
        reportError("class reference expected",e->lineno);
    } else {
        if (e->val.castE.right->type->kind==refK &&
            !subClass(e->val.castE.leftsym,
                       e->val.castE.right->type->class) && \textit{vice versa}) {
            reportError("cast will always fail",e->lineno);
        }
    }
break;
instanceof expression

\[
L, C, M, V \vdash E : τ \quad τ \leq C \lor C \leq τ
\]

\[
L, C, M, V \vdash E \text{ instanceof } C : \text{boolean}
\]
case instanceofK:
    typeImplementationEXP(e->val.instanceofE.left,class);
    if (e->val.instanceofE.left->type->kind!=refK) {
        reportError("class reference expected",e->lineno);
    }
    if (!subClass(e->val.instanceofE.left->type->class,
                  e->val.instanceofE.rightsym) && vice versa) {
        reportError("instanceof will always fail",e->lineno);
    }
    e->type = boolTYPE;
    break;
Think about why the predicate
\[ \tau \leq C \lor C \leq \tau \]
is used for \((C)E\) and \(E\) instanceof \(C\)?
Sub-type Testing

succeeds if $\tau \leq C$

don’t know if $C \leq \tau$

fails if $\tau \not\leq C \lor C \not\leq \tau$
Method Invocation

\[
L, C, M, V \vdash E : \sigma \\
L, C, M, V \vdash E_i : \sigma_i \\
type(L, \sigma, m) = \tau \\
\text{argtype}(L, \sigma, m, i) := \sigma_i \\
\hline \\
L, C, M, V \vdash E.m(E_1, \ldots, E_n) : \tau
\]
case invokeK:
    t = typeImplementationRECEIVER(
        e->val.invokeE.receiver,class);
    typeImplementationARGUMENT(e->val.invokeE.args,class);
    if (t->kind!=refK) {
        reportError("receiver must be an object",e->lineno);
        e->type = polynullTYPE;
    } else {
        s = lookupHierarchy(e->val.invokeE.name,t->class);
        e->val.invokeE.method = s;
        ...
    }
if (s==NULL || s->kind!=methodSym) {
    reportStrError("no such method called %s", ...
    e->type = polynullTYPE;
} else {
    if (s->val.methodS.modifier==modSTATIC) {
        reportStrError("static method %s may not be invoked", ...
    }
    typeImplementationFORMALARGUMENT(
        s->val.methodS.formals, ...
    e->type = s->val.methodS.returntype;
    }
}
break;
Kinds of Type Rules

- Axioms (i.e., given facts)
  \[ L, C, M, V \vdash \text{this : } C \]
- Predicates (i.e., boolean tests on type vars)
  \[ \tau \leq C \lor C \leq \tau \]
- Inferences (i.e., given \( x \) we can conclude \( y \))
  \[
  L, C, M, V \vdash E_1 : \text{int} \quad L, C, M, V \vdash E_2 : \text{int}
  \]
  \[ L, C, M, V \vdash E_1 - E_2 : \text{int} \]
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

where \( S = L, C, M, V[x \mapsto A][y \mapsto B] \) and \( B \leq A \)
\[
\begin{align*}
L,C,M,V &\vdash E_1 : \text{int} \quad L,C,M,V \vdash E_2 : \text{int} \\
&\quad \quad \quad L,C,M,V \vdash E_1 + E_2 : \text{int}
\end{align*}
\]

\[
\begin{align*}
L,C,M,V &\vdash E_1 : \text{String} \quad L,C,M,V \vdash E_2 : \tau \\
&\quad \quad \quad L,C,M,V \vdash E_1 + E_2 : \text{String}
\end{align*}
\]

\[
\begin{align*}
L,C,M,V &\vdash E_1 : \tau \quad L,C,M,V \vdash E_2 : \text{String} \\
&\quad \quad \quad L,C,M,V \vdash E_1 + E_2 : \text{String}
\end{align*}
\]

The + operator is overloaded
case plusK:
    typeImplementationEXP(e->val.plusE.left, class);
    typeImplementationEXP(e->val.plusE.right, class);
    e->type = typePlus(e->val.plusE.left,
                        e->val.plusE.right, e->lineno);
    break;
TYPE *typePlus(EXP *left, EXP *right, int lineno)
{
    if (equalTYPE(left->type,intTYPE) &&
        equalTYPE(right->type,intTYPE)) {
        return intTYPE;
    }
    if (!equalTYPE(left->type,stringTYPE) &&
        !equalTYPE(right->type,stringTYPE)) {
        reportError("arguments for + have wrong types", 
                    lineno);
    }
    left->tostring = 1;
    right->tostring = 1;
    return stringTYPE;
}
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

\[
\frac{L,C,M,V \vdash E_1 : \tau \quad L,C,M,V \vdash E_2 : \sigma}{L,C,M,V \vdash E_1 + E_2 : \text{String}}
\]

be included to handle coercions?
Testing a Type Checker

• The testing strategy for the type checker involves a further extension of the pretty printer, where the type of every expression is printed explicitly.

• These types are then compared to a corresponding manual construction for a sufficient collection of programs.

• Furthermore, every error message should be provoked by some test program.