Type Checking
Recap and Final Part
COMP 520: Compiler Design (4 credits)
Professor Laurie Hendren
hendren@cs.mcgill.ca

WendyTheWhitespace-IntolerantDragon
WendyTheWhitespaceenogarDtnarelotnI
The *type checker* has several tasks:

- 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.
Static type systems are necessarily flawed:
What are the advantages/disadvantages of static type checking?
What are the advantages/disadvantages of dynamic type checking?
What are the advantages/disadvantages of type inference?
The 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 \( V \).

The 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.
From an implementation point of view ....

- A recursive traversal through the AST;
- Assuming we have a symbol table giving declared types;
- First type-checking the components; and
- then checking structure.

```c
void typeImplementationCLASSFILE(CLASSFILE * c)
{
    if (c!=NULL) {
        typeImplementationCLASSFILE(c->next);
        typeImplementationCLASS(c->class);
    }
}

void typeImplementationCLASS(CLASS * c)
{
    typeImplementationCONSTRUCTOR(c->constructors,c);
    uniqueCONSTRUCTOR(c->constructors);
    typeImplementationMETHOD(c->methods,c);
}
```
Type rules for statement sequence:

\[
L, C, M, V \vdash S_1 \quad L, C, M, V \vdash S_2
\]

\[
L, C, M, V \vdash S_1 \quad S_2
\]

\[
L, C, M, V \vdash x \mapsto \tau \quad \vdash S
\]

\[
L, C, M, V \vdash \tau \quad x; S
\]

\(V[x \mapsto \tau]\) just says \(x\) maps to \(\tau\) within \(V\).

Corresponding JOOS source:

```java
  case sequenceK:
    typeImplementationSTATEMENT(s->val.sequenceS.first, class, returntype);
    typeImplementationSTATEMENT(s->val.sequenceS.second, class, returntype);
    break;

  ...

  case localK:
    break;
```
Assignment compatibility:

- \( \text{int} := \text{int} \);
- \( \text{int} := \text{char} \);
- \( \text{char} := \text{char} \);
- \( \text{boolean} := \text{boolean} \);
- \( \text{C} := \text{polynull}; \text{and} \)
- \( \text{C} := \text{D}, \text{if } D \leq C \).

- Where are the assignment compatibility rules used?
- What are other reasonable assignment compatibility rules?
Type rule for equality:

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

\[ L,C,M,V \vdash E_1 == E_2 : \text{boolean} \]

Corresponding JOOS source:

```java
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;
```
Type rule for method invocation:

\[
\begin{align*}
    L, C, M, V & \vdash E : \sigma \land \sigma \in L \\
    \exists \rho : \sigma \leq \rho \land m \in \text{methods}(\rho) \\
    \neg \text{static}(m) \\
    L, C, M, V & \vdash E_i : \sigma_i \\
    \text{argtype}(L, \rho, m, i) & := \sigma_i \\
    \text{return\_type}(L, \rho, m) & = \tau \\
    \hline
    L, C, M, V & \vdash E \cdot m(E_1, \ldots, E_n) : \tau
\end{align*}
\]
Corresponding JOOS source:

```java
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);
        if (s==NULL || s->kind!=methodSym) {
            reportStrError("no such method called %s",
                e->val.invokeE.name, e->lineno);
            e->type = polynullTYPE;
        } else {
            e->val.invokeE.method = s->val.methodS;
            if (s->val.methodS.modifier==modSTATIC) {
                reportStrError(
                    "static method %s may not be invoked",
                    e->val.invokeE.name, e->lineno);
            }
            typeImplementationFORMALARGUMENT(
                s->val.methodS->formals,
                e->val.invokeE.args, e->lineno);
            e->type = s->val.methodS->returntype;
        }
    }
break;
```
Type rule for constructor invocation:

\[ L, C, M, V \vdash E_i : \sigma_i \]

\[ \exists \vec{\tau} : \text{constructor}(L, C, \vec{\tau}) \land \vec{\tau} := \vec{\sigma} \land (\forall \vec{\gamma} : \text{constructor}(L, C, \vec{\gamma}) \land \vec{\gamma} := \vec{\sigma} \downarrow \vec{\gamma} := \vec{\tau}) \]

\[ \vdash L, C, M, V \vdash \text{new } C(E_1, \ldots, E_n) : C \]

Corresponding JOOS source:

```java
case newK:
    if (e->val.newE.class->modifier==modABSTRACT) {
        reportStrError("illegal abstract constructor \%s", e->val.newE.class->name, e->val.newE.class->name, e->lineno);
    }
    typeImplementationARGUMENT(e->val.newE.args, this);
    e->val.newE.constructor =
        selectCONSTRUCTOR(e->val.newE.class->constructors, e->val.newE.args, e->lineno);
    e->type = classTYPE(e->val.newE.class);
    break;
```
Simple example of an ambiguous constructor call

```java
public class AmbConst
{
    AmbConst(String s, Object o)
    {
    }

    AmbConst(Object o, String s)
    {
    }

    public static void main(String args[])
    {
        Object o = new AmbConst("abc","def");
    }
}
```

> javac AmbConst.java
AmbConst.java:9: error: reference to AmbConst is ambiguous
    Object o = new AmbConst("abc","def");
     ^
    both constructor AmbConst(String,Object) in AmbConst and
    constructor AmbConst(Object,String) in AmbConst match
1 error
Different kinds of type rules are:

- **axioms:**
  \[ L, C, M, V \vdash \text{this} : C \]

- **predicates:**
  \[ \tau \leq C \lor C \leq \tau \]

- **inferences:**
  \[
  \frac{L, C, M, V \vdash E_1 : \text{int} \quad L, C, M, V \vdash E_2 : \text{int}}{L, C, M, V \vdash E_1 - E_2 : \text{int}}
  \]
A type proof is a tree in which:

- nodes are inferences; and
- leaves are axioms or true predicates.

A program is statically type correct

\[ \text{iff} \]

it is the root of some type proof.

A type proof is just a trace of a successful run of the type checker.
An example type proof:

\[
\begin{align*}
&\quad V[x\mapsto A][y\mapsto B](y)=B \\
\therefore &\quad S \vdash y : B \\
\hline
&\quad S \vdash x : A \\
&\quad S \vdash (B)x : B
\end{align*}
\]

\[
\begin{align*}
&\quad L, C, M, V[x\mapsto A][y\mapsto B] \vdash y=(B)x : B \\
\hline
&\quad L, C, M, V[x\mapsto A][y\mapsto B] \vdash y=(B)x; \\
\hline
&\quad L, C, M, V[x\mapsto A] \vdash B \ y; \ y=(B)x; \\
\hline
&\quad L, C, M, V \vdash A \ x; \ B \ y; \ y=(B)x;
\end{align*}
\]

where \( S = L, C, M, V[x\mapsto A][y\mapsto B] \) and we assume that \( B \leq A \).
Type rules for plus:

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

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

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

The operator + is overloaded.
Corresponding JOOS source:

```java
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;
}
```
A coercion is a conversion function that is inserted automatically by the compiler.

The code:

"abc" + 17 + x

is transformed into:

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

What effect would a rule like:

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

have on the type system if it were included?
What are the advantages/disadvantages of static type checking?
What are the advantages/disadvantages of dynamic type checking?
What are the advantages/disadvantages of type inference?
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.