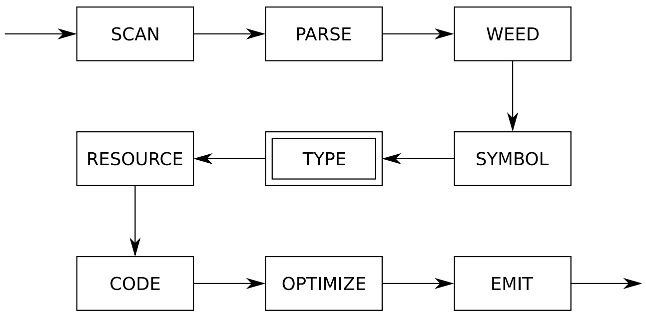
Type Checking

Recap and Final Part

COMP 520: Compiler Design (4 credits)

Professor Laurie Hendren

`hendren@cs.mcgill.ca`



WendyTheWhitespace-IntolerantDragon

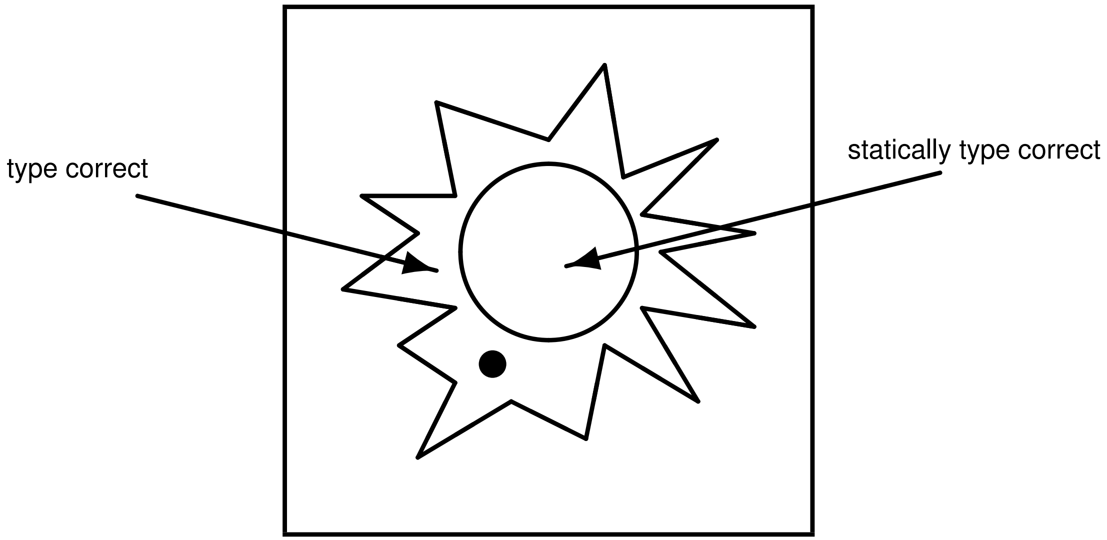
WendyTheWhitespacenogarDtnareIotnI

The *type checker* has severals 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 τ.

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.

```
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:

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

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

$V[\mathrm{x} \mapsto \tau]$ just says x maps to τ within V.

Corresponding JOOS source:

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

case localK:
    break;
```

Assignment compatibility:

- - `int:=int`;
  - `int:=char`;
  - `char:=char`;
  - `boolean:=boolean`;
  - `C:=polynull`; and
  - `C:=D`, if $D \leq C$.
- Where are the assignment compatibility rules used?
- What are other reasonable assignment compatibility rules?

Type rule for equality:

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

Corresponding JOOS source:

```
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:

$$\frac{\begin{array}{l} L, C, M, V \vdash E : \sigma \wedge \sigma \in L \\ \exists\, \rho\colon \sigma \leq \rho \wedge \mathtt{m} \in \textit{methods}(\rho) \\ \neg\textit{static}(\mathtt{m}) \\ L, C, M, V \vdash E_i : \sigma_i \\ \textit{argtype}(L, \rho, \mathtt{m}, i) := \sigma_i \\ \textit{return\_type}(L, \rho, \mathtt{m}) = \tau \end{array}}{L, C, M, V \vdash E\,.\,\mathtt{m}\,(E_1, \ldots, E_n)\ : \tau}$$

Corresponding JOOS source:

```
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:

$$\frac{\begin{array}{l} L, C, M, V \vdash E_i : \sigma_i \\ \exists\, \vec{\tau}:\ \mathit{constructor}(L, \texttt{C}, \vec{\tau}) \wedge \\ \quad \vec{\tau} := \vec{\sigma} \wedge \\ \quad (\forall \vec{\gamma}:\ \mathit{constructor}(L, \texttt{C}, \vec{\gamma}) \wedge \vec{\gamma} := \vec{\sigma} \\ \qquad\qquad \Downarrow \\ \qquad\qquad \vec{\gamma} := \vec{\tau} \\ \quad) \end{array}}{L, C, M, V \vdash \texttt{new C(}E_1, \ldots, E_n\texttt{)} : \texttt{C}}$$

Corresponding JOOS source:

```
case newK:
    if (e->val.newE.class->modifier==modABSTRACT) {
       reportStrError("illegal abstract constructor %s",
                      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

```
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*:

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

- *predicates*:

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

- *inferences*:

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

A *type proof* is a tree in which:

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

A program is statically type correct *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:

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

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

Type rules for plus:

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

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

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

The operator + is *overloaded*.

Corresponding JOOS source:

```
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:

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

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.