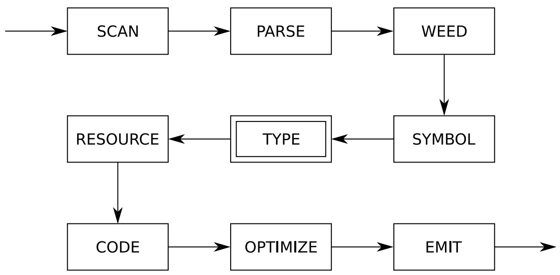
Type checking



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.

A *type* describes possible values.

The JOOS types are:

- `void`: the empty type;
- `int`: the integers;
- `char`: the characters;
- `boolean`: `true` and `false`; and
- `C`: objects of class `C` or any subclass.

Plus an artificial type:

- `polynull`

which is the type of the polymorphic `null` constant.

A *type annotation*:

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

specifies an *invariant* about the run-time behavior:

- `x` will always contain an integer value; and
- `y` will always contain `null` or an object of type `Cons` or any subclass.

Usual type annotations are not very expressive as invariants.

You can have types without annotations, through type inference (e.g. in ML).

Types can be arbitrarily complex in theory.

A program is *type correct* if the type annotations are valid invariants.

Type correctness is undecidable:

```
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.

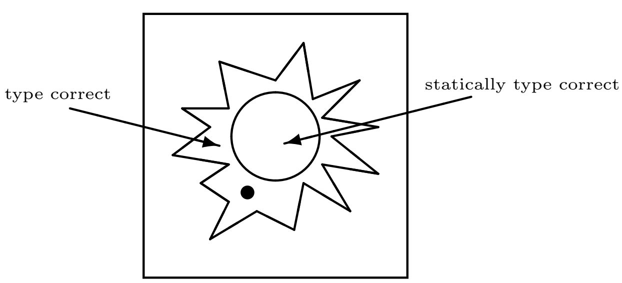
A program is *statically* type correct if it satisfies some type rules.

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.

Static type systems are necessarily flawed:



There is always *slack*, i.e. programs that are unfairly rejected by the type checker. Some are even quite useful.

Can you think of such a program?

Type rules may be specified:

- in ordinary prose:
  `The argument to the sqrt function must be of type int; the result is of type real.`
- as constraints on type variables:
  $\texttt{sqrt(x)}: \quad [\![\texttt{sqrt(x)}]\!]=\texttt{real} \wedge [\![\texttt{x}]\!]=\texttt{int}$
- as logical rules:

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

There are always three kinds:

1. declarations: introduction of variables;
2. propagations: expression type determines enclosing expression type; and
3. restrictions: expression type constrained by usage context

The judgement for statements:

$$\boldsymbol{L, C, M, V \vdash S}$$

means that $\boldsymbol{S}$ is statically type correct with:

- class library $\boldsymbol{L}$;
- current class $\boldsymbol{C}$;
- current method $\boldsymbol{M}$; and
- variables $\boldsymbol{V}$.

The judgement for expressions:

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

means that $\boldsymbol{E}$ is statically type correct and has type $\boldsymbol{\tau}$.

The tuple $\boldsymbol{L, C, M, V}$ is an abstraction of the symbol table.

Type rules for statement sequence:

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

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

$\boldsymbol{V}[\texttt{x} \mapsto \boldsymbol{\tau}]$ just says `x` maps to $\boldsymbol{\tau}$ within $\boldsymbol{V}$.

Corresponding JOOS source:

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

case localK:
     break;
```


Type rules for return statements:

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

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

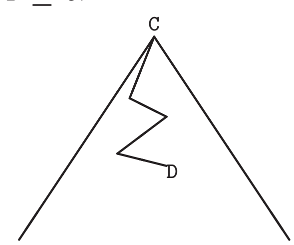
$\boldsymbol{\sigma := \tau}$ just says something of type $\boldsymbol{\sigma}$ can be assigned something of type $\boldsymbol{\tau}$.

Corresponding JOOS source:

```
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`;
- `int:=char`;
- `char:=char`;
- `boolean:=boolean`;
- `C:=polynull`; and
- `C:=D`, if `D` $\leq$ `C`.



Corresponding JOOS source:

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

Type rule for expression statements:

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

Corresponding JOOS source:

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

Type rule for `if`-statement:

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

Corresponding JOOS source:

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


Type rule for variables:

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

Corresponding JOOS source:

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

Type rule for assignment:

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

Corresponding JOOS source:

```
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;
```


Type rule for minus:

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

Corresponding JOOS source:

```
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;
```

Implicit integer cast:

$$\frac{L,C,M,V \vdash E : \texttt{char}}{L,C,M,V \vdash E : \texttt{int}}$$

Corresponding JOOS source:

```
int checkINT(TYPE *t, int lineno)
{ if (t->kind!=intK && t->kind!=charK) {
     reportError("int type expected",lineno);
     return 0;
  }
  return 1;
}
```


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 `this`:

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

Corresponding JOOS source:

```
case thisK:
     if (class==NULL) {
        reportError("'this' not allowed here",e->lineno);
     }
     e->type = classTYPE(class);
     break;
```


Type rule for cast:

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

Corresponding JOOS source:

```
case castK:
     typeImplementationEXP(e->val.castE.right,class);
     e->type = makeTYPEextref(e->val.castE.left,
                              e->val.castE.class);
     if (e->val.castE.right->type->kind!=refK &&
         e->val.castE.right->type->kind!=polynullK) {
        reportError("class reference expected",e->lineno);
     } else {
        if (e->val.castE.right->type->kind==refK &&
            !subClass(e->val.castE.class,
                      e->val.castE.right->type->class) &&
            !subClass(e->val.castE.right->type->class,
                      e->val.castE.class)) {
           reportError("cast will always fail",e->lineno);
        }
     }
     break;
```


Type rule for `instanceof`:

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

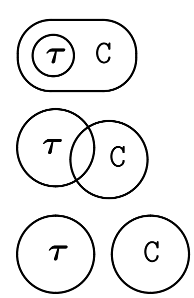
Corresponding JOOS source:

```
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.class) &&
       !subClass(e->val.instanceofE.class,
                 e->val.instanceofE.left->type->class)) {
      reportError("instanceof will always fail",e->lineno);
   }
   e->type = boolTYPE;
   break;
```


Why the predicate:

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

for "`(C)`E" and "E `instanceof C`"?



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

really useful $\texttt{C} \leq \tau$

fails $\tau \not\leq \texttt{C} \wedge \texttt{C} \not\leq \tau$

Circle denotes type and all its subtypes. For instance, the following would fail to type check, as no subtype of `List` can ever be a subtype of the final (!) class `String`:

```
List l;
if(l instanceof String) ...
```

Type rule for method invocation:

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

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:

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

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;
```


Different kinds of type rules are:

- *axioms*:

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

- *predicates*:

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

- *inferences*:

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

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:

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

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

Type rules for plus:

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

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

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

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:

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

have on the type system if it were included?

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.