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.

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:

```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.

---

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:

```
type correct

<table>
<thead>
<tr>
<th>type correct</th>
<th>statically type correct</th>
</tr>
</thead>
</table>
```

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:
  
  `sqrt(x): [sqrt(x)] = real ∧ [x] = int`

- as logical rules:
  
  $$\frac{S \vdash x : \text{int}}{S \vdash \sqrt{x} : \text{real}}$$

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:

\[ L, C, M, V \vdash S \]

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.

Type rules for statement sequence:

\[
\begin{align*}
L, C, M, V \vdash S_1 & \quad L, C, M, V \vdash S_2 \\
L, C, M, V[x \mapsto \tau] \vdash S & \\
L, C, M, V \vdash \tau x; S
\end{align*}
\]

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

Corresponding JOOS source:

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

Type rules for return statements:

\[
\begin{align*}
type(L, C, M) &= void \\
L, C, M, V \vdash return & \quad \sigma := \tau \\
L, C, M, V \vdash return \ E
\end{align*}
\]

\( \sigma := \tau \) just says something of type \( \sigma \) can be assigned something of type \( \tau \).

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:

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

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==intK && t->kind==charK) return 0;
    if (s->kind==refK) return subClass(t->class,s->class);
    return 1;
}
```

```
C
-----------
| |
D
```

```
Type rule for expression statements:

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

\[ L, C, M, V \vdash E \]

Corresponding JOOS source:

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

Type rule for if-statement:

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

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

Corresponding JOOS source:

```java
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:

\[ V(x) = \tau \]

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

Corresponding JOOS source:

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

Type rule for assignment:

\[ L, C, M, V \vdash x : \tau \quad L, C, M, V \vdash E : \sigma \quad \tau := \sigma \]

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

Corresponding JOOS source:

```java
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:

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

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

Corresponding JOOS source:

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

Implicit integer cast:

\[ L, C, M, V \vdash E : \text{char} \]

\[ L, C, M, V \vdash E : \text{int} \]

Corresponding JOOS source:

```java
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:

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

\[ \tau_1 := \tau_2 \lor \tau_2 := \tau_1 \]

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

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 *this*:
\[ L, C, M, V \vdash \textit{this} : C \]

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

Type rule for *cast*:
\[ L, C, M, V \vdash E : \tau \quad \tau \leq \varnothing \quad \varnothing \leq \tau \\
L, C, M, V \vdash (C)E : C \]

Corresponding JOOS source:
```
case cast:
  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 &
    !subClass(e->val.castE.right->type->class, e->val.castE.class)) {
    reportError("class reference expected", e->lineno);
  } else {
    if (e->val.castE.right->type->kind==refK &
        !subClass(e->val.castE.right->type->class, e->val.castE.right->type->kind="poly Kl") {
      reportError("class reference expected", e->lineno);
    }
  }
  e->type = makeTYPEextref(e->val.castE.left, e->val.castE.class);
  break;
```

Type rule for *instanceof*:
\[ L, C, M, V \vdash E : \tau \quad \tau \leq C \quad C \leq \tau \\
L, C, M, V \vdash E \text{ instanceof } C : \text{boolean} \]

Corresponding JOOS source:
```
case instanceof:
  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.left->type->class, e->val.instanceofE.left->type->class)) {
    reportError("instanceof will always fail", e->lineno);
  }
  e->type = boolTYPE;
  break;
```

Why the predicate:
\[ \tau \leq C \quad C \leq \tau \]

for "\((C)E\)" and "\(E \text{ instanceof } C\)"?

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:

\[
L, C, M, V \vdash E : \sigma \land \sigma \in L
\]
\[
\exists \rho : \sigma \leq \rho \land m \in \text{methods}(\rho)
\]
\[
\neg \text{static}(m)
\]
\[
L, C, M, V \vdash E_i : \sigma_i
\]
\[
\text{argtype}(L, \rho, m, i) := \gamma_i \land \gamma_i := \sigma_i
\]
\[
\text{type}(L, \rho, m) = \tau
\]
\[
L, C, M, V \vdash E.m(E_1, \ldots, E_n) : \tau
\]

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:

\[
L, C, M, V \vdash E_i : \sigma_i
\]
\[
\exists \vec{\tau} : \text{constructor}(L, C, \vec{\tau}) \land 
\vec{\tau} := \sigma \land 
(\forall \vec{\gamma} : \text{constructor}(L, C, \vec{\gamma}) \land \vec{\gamma} := \sigma 
\downarrow 
\vec{\gamma} := \vec{\tau})
\]
\[
L, C, M, V \vdash \text{new } C(E_1, \ldots, E_n) : C
\]

Corresponding JOOS source:

```java
case newK:
    if (e->val.newE.class->modifier==modABSTRACT) {  
        reportStrError("illegal abstract constructor %s",
            e->val.newE.class->name,
            e->lineno);
    } else {  
        s = lookupHierarchy(e->val.newE.class->name,t->class);
        if (s->val.methodS->modifier==modSTATIC) {  
            reportStrError("static method %s may not be invoked",
                e->val.newE.name,e->lineno);
        } else {  
            s = lookupHierarchy(e->val.newE.name,t->class);
            e->val.newE.constructor = selectCONSTRUCTOR(s->val.constructor->name,
                e->val.newE.class->name);
            e->type = s->val.constructor->returntype;
        }
    }
    break;
```

Different kinds of type rules are:

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

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

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

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

A program is statically type correct iff it is the root of some type proof.

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

Type rules for plus:

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

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

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

The operator + is overloaded.

An example type proof:

\[
\begin{array}{c}
V^x = A \\
V^y = B \\
S \vdash x : A \\
S \vdash B x : B \\
L, C, M, V^x \vdash y(B)x : B \\
L, C, M, V^x \vdash B y : y(B)x; \\
L, C, M, V^x \vdash B y; y=Bx;
\end{array}
\]

where \( S = L, C, M, V^x \) and we assume that \( B \leq A \).

Type rules for plus:

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

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

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

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;
      } else if (!equalTYPE(left->type,stringTYPE) &&
               !equalTYPE(right->type,stringTYPE))
          reportError("arguments for + have wrong types",
                     lineno);
      } else
      { 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:

\[
L,C,M,V \vdash E_1: \tau \quad L,C,M,V \vdash E_2: \sigma
\]

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

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.