

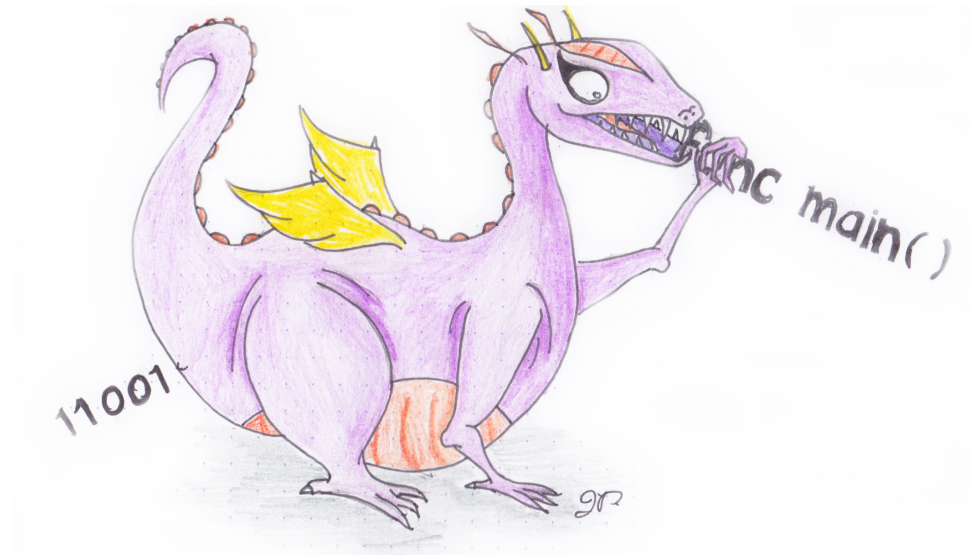
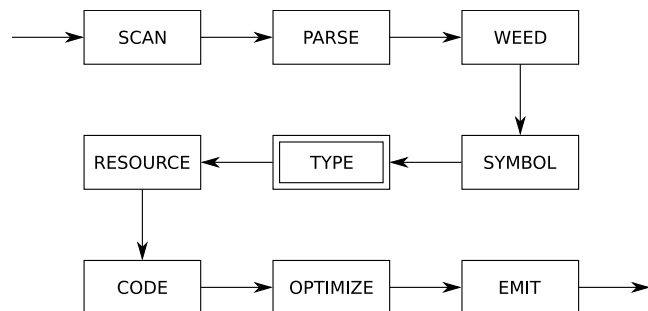
# Type Checking

COMP 520: Compiler Design (4 credits)

Alexander Krolik

`alexander.krolik@mail.mcgill.ca`

MWF 13:30-14:30, MD 279



McCompiley

## **Announcements (Wednesday, February 1st)**

### **Milestones:**

- Assignment 2 due **Friday, February 10th 11:59PM** on myCourses


MCGILL SCIENCE UNDERGRADUATE SOCIETY PRESENTS

# SUS ACADEMIA WEEK 2017

**FEBRUARY 6 – 11**

<b>MON 6</b>	<b>TUES 7</b>	<b>WED 8</b>	<b>THURS 9</b>	<b>FRI 10</b>	<b>SAT 11</b>
NSERC information session Prep 101: Talking to a Med Student	Major matchmaker Dr. Mark Wainberg: How Today's Young Scientists Can Transform Our World Tomorrow	Science abroad – research opportunities Assassin's Creed: Videogame development at Ubisoft	McGill Prof Reddit AMA – Professor Pavelka Scientists for Love: Finding Love with Your Senses	Internships, Jobs, and Networking in Science – CaPS Jay Olsen: Psychology of Magic	<b>KEYNOTE SPEAKER</b> Veritasium: YouTube star Derek Muller

For times, locations, and more information visit the Facebook page – *Stranger Things: SUS Academia Week 2017!*



**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.**

Static type correctness is undecidable:

```
int x;
```

```
int j;
```

```
x = 0;
```

```
scanf("%i", &j);
```

```
TM(j);
```

```
x = true; // does this invalid type assignment happen?
```

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.**

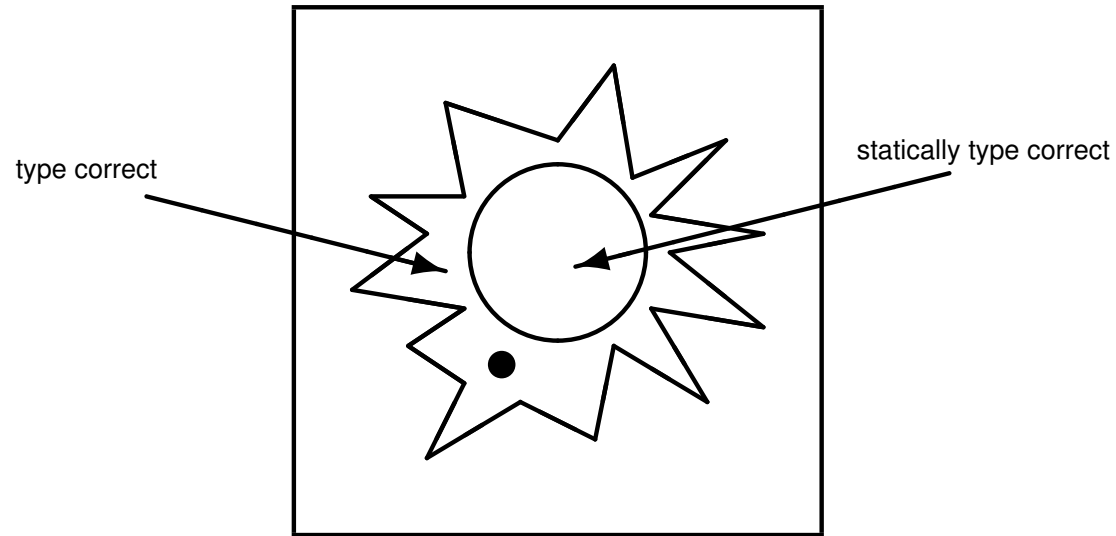
The type rules are chosen to be:

- 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.

```
int x;
```

```
x = 87;
```

```
if (false) x = true;
```

**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:

$$\text{sqrt}(x) : \llbracket \text{sqrt}(x) \rrbracket = \text{real} \wedge \llbracket x \rrbracket = \text{int}$$

- as logical rules:

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

**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.

## 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);
}
```

## The key parts are type checking statements and expressions

```
void typeImplementationSTATEMENT (STATEMENT *s, CLASS *this, TYPE *returntype) {
    if (s!=NULL) {
        switch (s->kind) {
            case skipK:
                break;
            case localK:
                break;
            case expK:
                typeImplementationEXP (s->val.expS, this);
                break;
            case returnK:
                [...]
            case sequenceK:
                typeImplementationSTATEMENT (s->val.sequences.first,
                    this, returntype);
                typeImplementationSTATEMENT (s->val.sequences.second,
                    this, returntype);
                break;
            [...]
        }
    }
}
```

## Type checking expressions also stores the resulting type in the expression node

```
void typeImplementationEXP (EXP *e, CLASS *this) {
    SYMBOL *s;
    TYPE *t;
    switch (e->kind) {
        case idK:
            e->type = typeVar(e->val.idE.idsym);
            break;
        case assignK:
            e->type = typeVar(e->val.assignE.leftsym);
            typeImplementationEXP (e->val.assignE.right, this);
            if (!assignTYPE (e->type, e->val.assignE.right->type)) {
                reportError("illegal assignment", e->lineno);
            }
            break;
        case orK:
            typeImplementationEXP (e->val.orE.left, this);
            typeImplementationEXP (e->val.orE.right, this);
            checkBOOL (e->val.orE.left->type, e->lineno);
            checkBOOL (e->val.orE.right->type, e->lineno);
            e->type = boolTYPE;
            break;
        [...]
    }
}
```

**Type rules for statement sequence:**

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

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

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

Corresponding JOOS source:

**case** sequenceK:

```
typeImplementationSTATEMENT(s->val.sequencesS.first, class, returntype);
typeImplementationSTATEMENT(s->val.sequencesS.second, class, returntype);
break;
```

[...]

**case** localK:

```
break;
```

**Type rules for return statements:**

$$\frac{\text{return\_type}(L, C, M) = \text{void}}{L, C, M, V \vdash \text{return}}$$

$$\frac{L, C, M, V \vdash E : \tau \quad \text{return\_type}(L, C, M) = \sigma \quad \sigma := \tau}{L, C, M, V \vdash \text{return } E}$$

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

Corresponding JOOS source:

```

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

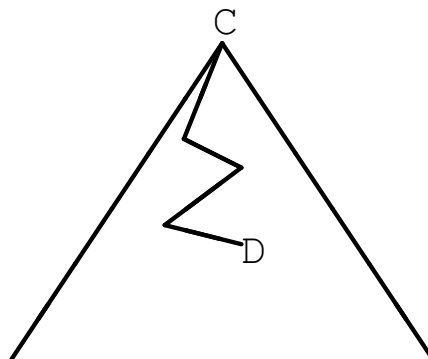
```



```
}  
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 *lhs, TYPE *rhs) {  
    if (lhs->kind==refK && rhs->kind==polynullK) return 1;  
    if (lhs->kind==intK && rhs->kind==charK) return 1;  
    if (lhs->kind!=rhs->kind) return 0;  
    if (lhs->kind==refK) return subClass(rhs->class, lhs->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:**

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

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:**

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

Corresponding JOOS source:

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

**Type rule for assignment:**

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

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:**

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

Corresponding JOOS source:

```
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:**

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

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:**

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

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`:**

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

Corresponding JOOS source:

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

**Type rule for cast:**

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

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`:

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

Corresponding JOOS source:

```

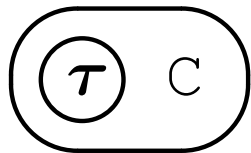
case instanceofK:
  typeImplementationEXP (e->val.instanceofE.left, class);
  if (e->val.instanceofE.left->type->kind!=refK) {
    reportError("class reference expected", e->lineno);
  }
  if (!subClass (e->val.instanceofE.left->type->class,
    e->val.instanceofE.class) &&
    !subClass (e->val.instanceofE.class,
    e->val.instanceofE.left->type->class)) {
    reportError("instanceof will always fail", e->lineno);
  }
  e->type = boolTYPE;
  break;

```

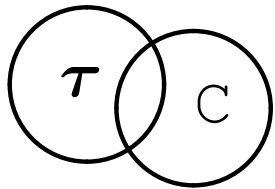
**Why the predicate?:**

$$\tau \leq C \vee C \leq \tau$$

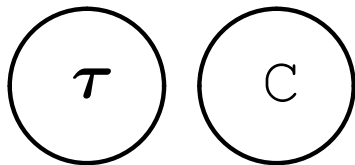
for “(C) *E*” and “*E* instanceof C”?



succeeds  $\tau \leq C$



really useful  $C \leq \tau$



fails  $\tau \not\leq C \wedge C \not\leq \tau$

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:**

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

**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:**

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

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*:

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

- *predicates*:

$$\tau \leq C \vee C \leq \tau$$

- *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  
*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:**

$$\begin{array}{c}
\frac{V[x \mapsto A][y \mapsto B](y) = B}{\mathcal{S} \vdash y : B} \quad \frac{\frac{V[x \mapsto A][y \mapsto B](x) = A}{\mathcal{S} \vdash x : A} \quad A \leq B \vee B \leq A}{\mathcal{S} \vdash (B) x : B} \quad B := B \\
\hline
L, C, M, V[x \mapsto A][y \mapsto B] \vdash y = (B) x : B \\
\hline
L, C, M, V[x \mapsto A][y \mapsto B] \vdash y = (B) x; \\
\hline
L, C, M, V[x \mapsto A] \vdash B \ y; \ y = (B) x; \\
\hline
L, C, M, V \vdash A \ x; \ B \ y; \ y = (B) x;
\end{array}$$

where  $\mathcal{S} = L, C, M, V[x \mapsto A][y \mapsto B]$  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:

```
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:

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

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.