Type Checking

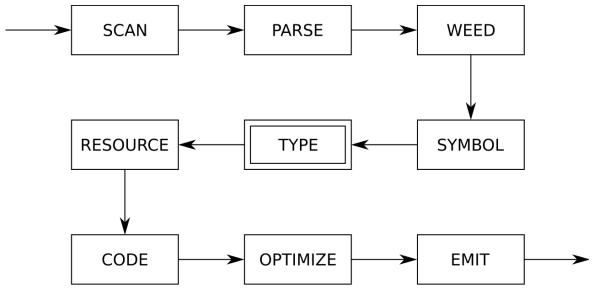
COMP 520: Compiler Design (4 credits)

Alexander Krolik

alexander.krolik@mail.mcgill.ca

MWF 8:30-9:30, TR 1080

http://www.cs.mcgill.ca/~cs520/2019/



Compiley “Mompiler” McCompilerface

Announcements (Friday, February 1st)

Milestones

- Group signup form `https://goo.gl/forms/zq6sYn8YLUsA6QEy1`, fill this out before next week
- We will contact you today if you have not yet filled out the form

Assignments

- Assignment 1 nearly completed grading! Solution programs available on myCourses
- Assignment 2
  - Any questions?
  - Two things: scoping rules, and initialization
  - **Due:** Friday, February 8th 11:59 PM

Dragon Names

1. Justin time;
2. Donkey;
3. T-Flex;
4. Yacc the Magic Dragon;
5. Mompiler;
6. Compiley McCompilerface.

Type Checking

In the symbol table phase we start processing semantic information

- Collect declarations of identifiers; and
- Associate identifier uses with their corresponding declarations.

The *type checker* will use this information for 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.

Types

A *type* describes possible values for an identifier

The JOOS types are similar to those found in Java

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

There is also an artificial type

- `polynull`

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

Type Annotations

A *type annotation* specifies a type *invariant* about the **run-time** behaviour. Consider the following example

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

Given the type annotations, during runtime we expect that

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

You can have types without annotations through type inference (i.e. in ML), or dynamic typing (i.e. Python).

Types can be arbitrarily complex in theory – see COMP 523.

Type Correctness

A program is *type correct* if the type annotations are valid invariants. i.e. the annotations correctly describe the possible values for each variable

Type correctness is undecidable though

```
int x = 0;
int j;

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. But this is undecidable!

Type Correctness

Since type correctness is undecidable in general, we perform a conservative analysis instead.

Static type correctness

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

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

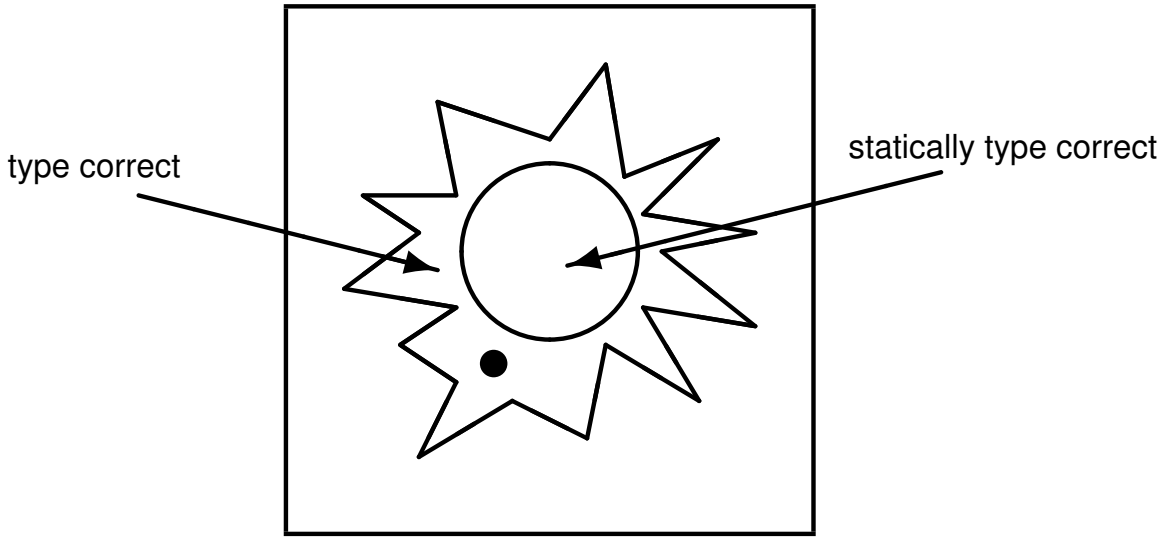
The type rules are chosen arbitrarily, but should be be

- Simple to understand;
- Efficient to decide;
- Conservative with respect to type correctness; and
- Provably lead to correct type behaviour.

Type rules are rarely the same between languages, and may not always be obvious.

Static Type Correctness

Due to their conservative nature, static type systems are necessarily flawed



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

```
int x = 87;
if (false) {
   x = true;
}
```

Type Rules

Type rules may be specified in different equivalent formats. Consider the function with signature

```
real sqrt(int x)
```

- Ordinary prose

  The argument to the sqrt function must be of type int; the result is of type real.

- Constraints on type variables

  $$\texttt{sqrt(x):} \quad [\![\texttt{sqrt(x)}]\!] = \texttt{real} \wedge [\![\texttt{x}]\!] = \texttt{int}$$

- Logical rules

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

There are always three kinds of rules

1. **Declarations**: introduction of variables;
2. **Propagations**: expression type determines enclosing expression type; and
3. **Restrictions**: expression type constrained by usage context

Logical Rules

In this class we focus on ordinary prose and logical rules for specifying type systems.

Logical rules

$\mathbf{\Gamma}$: Context/state/assumptions of the program, an abstraction of the symbol table

$$\frac{\mathbf{\Gamma} \vdash \texttt{P}}{\mathbf{\Gamma} \vdash \texttt{C}}$$

In plain English, this type rule specifies that

if `P` *is provable under context* $\mathbf{\Gamma}$*, then* `C` *is provable under context* $\mathbf{\Gamma}$

We use rules like this to construct declarations, propagations, and restrictions.

Logical Rules

There are 3 main operations that we cover in this class

Typing

The colon operation says that under context Γ it is provable that E is statically well typed and has type τ

$$\Gamma \vdash E : \tau$$

Modifying the context

In an abstract sense the context Γ represents the definition of identifiers (i.e. the symbol table). Given a declaration, we can therefore "push" elements onto the context for the statements that follow

$$\frac{\Gamma[\mathtt{x} \mapsto \tau] \vdash S}{\Gamma \vdash \tau\ \mathtt{x}; S}$$

Accessing the context

Lastly, we want to access elements that have been pushed onto the context

$$\frac{\Gamma(\mathtt{x}) = \tau}{\Gamma \vdash \mathtt{x} : \tau}$$

Judgements in JOOS

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

Statements

$\boldsymbol{S}$ is statically type correct with

- Class library $\boldsymbol{L}$;
- Current class $\boldsymbol{C}$;
- Current method $\boldsymbol{M}$; and
- Variables $\boldsymbol{V}$.

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

Expressions

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

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

Typechecker Implementation

A typechecker performs two key actions for verifying the semantic behaviour of a program

- Typechecks expressions are valid given the derived types; and
- Derives the type of parent expressions (propagation).

Implementation-wise this means

- A typechecker is typically implemented as a recursive traversal of the AST; and
- Assumes we have a symbol table giving declared types.

```
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);
}
```

Typechecking Statements

Statements do not have types, therefore they serve as boilerplate code to visit all expressions

```
void typeImplementationSTATEMENT(STATEMENT *s, CLASS *this, TYPE *returntype) {
   if (s == NULL) {
      return;
   }

   switch (s->kind) {
      case skipK:
         break;
      case localK:
         break;
      case expK:
         typeImplementationEXP(s->val.expS,this);
         break;
      [...]
      case sequenceK:
         typeImplementationSTATEMENT(s->val.sequenceS.first, this, returntype);
         typeImplementationSTATEMENT(s->val.sequenceS.second, this, returntype);
         break;
      [...]
   }
}
```

Typechecking Expressions

Expressions have a resulting type, therefore they also store the resulting type in the AST node

```
void typeImplementationEXP(EXP *e, CLASS *this) {
   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;
      [...]
   }
}
```

JOOS - Statement Sequences

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

Statement sequences are well typed if each statement in the sequence is well typed

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

JOOS - Declarations

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

$\boldsymbol{V}[\mathtt{x} \mapsto \boldsymbol{\tau}]$ says $\mathtt{x}$ maps to $\boldsymbol{\tau}$ within $\boldsymbol{V}$. This rule equivalently says that statement $\boldsymbol{S}$ must typecheck with $\mathtt{x}$ added to the symbol table.

Both declaration and use of identifiers is handled in the symbol table, therefore no action is needed

```
case localK:
   break;
```

JOOS - If Statements

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

An `if` statement in JOOS requires

- The condition to be well typed;
- The condition to have type boolean;
- The enclosed statement to be well typed; and
- If there is an `else` branch, the corresponding statement must also be well typed

Corresponding JOOS code

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

JOOS - Expression Statements

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

In JOOS, expression (which have a value) may be used as statements – think function calls, or assignments. For an expression statement to be well typed, the expression must be well typed

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

JOOS - Return Statements

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

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

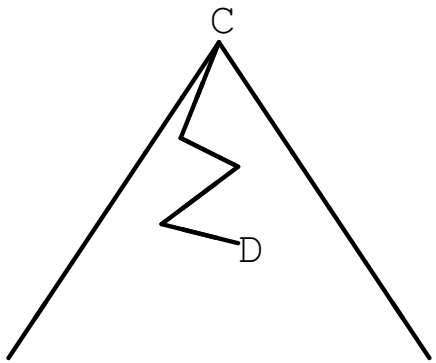
$\sigma := \tau$ says something of type σ can be assigned something of type τ (assignment compatibility)

```
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

In JOOS, assignment compatibility is defined as follows

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




```
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;
}
```

Announcements (Monday, February 4th)

Milestones

- Next Monday we will introduce the GoLite project!
- You will receive an invite to the “comp520” organization on GitHub this week; and
- If you do not yet have a group, please refer to your email.

Assignments

- Assignment 1 grading complete!
- Assignment 2
  - Initialization for declarations (`var a : int;`) added to spec
  - Any questions?
  - **Due:** Friday, February 8th 11:59 PM

JOOS - Variables (identifier expressions)

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

When using a variable, we look up the corresponding type in the *variables* portion of the context/symbol table. If it is defined and has type τ, then the expression also has type τ

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

JOOS - Assignment Expressions

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

In JOOS, assignments are expressions, allowing multiple assignments to occur in a single statement.

The resulting value is that which is stored in the variable, and thus we propagate the variable type. Assignments also require that the variable be defined, and the expression assignable to the variable type

```
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;
```

JOOS - Subtraction Expression

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

For integer subtraction (JOOS has no floating point type), both operands must be well typed as integers, and the resulting type is an integer

```
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;
```

JOOS - Plus Expression

The operator + is *overloaded* for handling string concatenation. In the case that a single operand is a string, the result is a string.

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

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

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

Coercions

A *coercion* is a conversion function that is inserted automatically by the compiler.

For plus expressions involving strings, the code

```
"abc" + 17 + x
```

is automatically transformed into string concatenation

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

What effect would a rule like

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

have on the type system if it were included?

JOOS - Plus Expression

```
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;
}
```

JOOS - Equality Expression

$$\frac{\boldsymbol{L},\boldsymbol{C},\boldsymbol{M},\boldsymbol{V} \vdash \boldsymbol{E_1} : \boldsymbol{\tau_1} \quad \boldsymbol{L},\boldsymbol{C},\boldsymbol{M},\boldsymbol{V} \vdash \boldsymbol{E_2} : \boldsymbol{\tau_2} \quad \boldsymbol{\tau_1} := \boldsymbol{\tau_2} \vee \boldsymbol{\tau_2} := \boldsymbol{\tau_1}}{\boldsymbol{L},\boldsymbol{C},\boldsymbol{M},\boldsymbol{V} \vdash \boldsymbol{E_1} == \boldsymbol{E_2} : \texttt{boolean}}$$

Equality in JOOS requires that both expressions are well typed, and that they are *comparable* – one is assignable (convertible) to the other. The result is of type boolean

```
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;
```

JOOS - `this`

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

In JOOS, the `this` keyword corresponds to the current object, and it's type is trivially the current class.

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

JOOS - Implicit Integer Cast

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

Characters are internally stored as integers, and can therefore be used at any point where integers are valid. The function `checkINT` therefore returns `true` for both character and integer types.

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

JOOS - Casting

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

A cast expression requires that the expression is well typed to some type τ, but also that τ is somewhere in the hierarchy of the destination type. Why?

```
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;
```

JOOS - `instanceof`

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

The `instanceof` operation resembles that of the cast, again requiring the expression type to be somewhere in the inheritance hierarchy of `C`

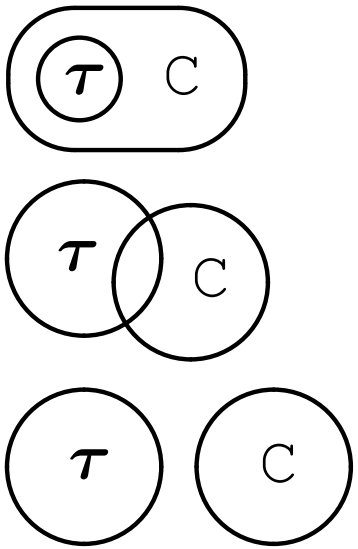
```
case instanceofK:
   typeImplementationEXP(e->val.instanceofE.left,class);
   if (e->val.instanceofE.left->type->kind != refK) {
      reportError("class reference expected",e->lineno);
   }
   if (!subClass(e->val.instanceofE.left->type->class,
                 e->val.instanceofE.class) &&
    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

Casting and `instanceof` Predicate

Why the predicate

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

for "`(C)` $\boldsymbol{E}$" and "$\boldsymbol{E}$ `instanceof C`"? Consider the following relationships between types τ and `C`



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

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

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

The last example corresponds to the following code, where `List` and `String` bear no relation to each other. JOOS disallows operations which are guaranteed to fail.

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

JOOS - Method Invocation

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

Try to explain the above!

1. Check that $\boldsymbol{E}$ corresponds to a class in the library
2. Check that the method exists in class $\boldsymbol{\sigma}$ or one of its superclasses
3. Check the arguments can be assigned
4. Fetch the return type

JOOS - Method Invocation

```
case invokeK:
   TYPE *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);
   } else {
      SYMBOL *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);
      } 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;
```

JOOS - Constructor Invocation

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

What does this do?! Think about the behaviour of overloading!

Overloading in Java

When the same method name has several implementations with different parameters/return types, the method is said to be *overloaded*.

Java picks the method with the narrowest *static* types – no runtime information is used.

```
public class A {
   public A(String a) { System.out.println("String"); }
   public A(Object o) { System.out.println("Object"); }

   public static void main(String[] args) {
      String p1 = "string";
      Object p2 = new Object();
      Object p3 = "string";
      new A(p1); // String
      new A(p2); // Object
      new A(p3); // Object
   }
}
```

Overloading in Java

In some cases there is no clear winner, they are both equally nice! In this case, we have 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");
   }
}
```

Compiling the class using `javac` yields an error

```
$ 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
```

JOOS - Constructor Invocation

The corresponding JOOS code for constructor invocation typechecking thus finds the best constructor and sets the appropriate expression type

```
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;
```

Kinds of Type Rules

Different kinds of type rules are

- *Axioms*

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

- *Predicates*

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

- *Inferences*

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

Type Proofs

A *type proof* is a tree in which

- Internal 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.

Example Type Proof

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

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

Testing Strategy

The testing strategy for the type checker involves

- Extending the pretty printer to print the type of every expression;
- Manually verifying the output types on a sufficient set of test programs; and
- Testing every branch/error message.