CS 565 October 27, 1997
Jens Palsberg Readings: OOTS, Chapter 1–4

Lecture Note 9:

The Java Type System

Contents

1 Motivation

- Work in the tradition of SIMULA, BETA, C++, EIFFEL, and Java.
- Initial observation: "types = sets of classes" gives simple description of existing type systems.
- Goal: to explain and improve these—not to overthrow them.

2 The BOPL Language

- Basic Object Programming Language.
- An untyped language model underlying our discussions.
- Supports the essential features:
  - classes and objects;
  - assignments; and
  - late binding.
- Inheritance and genericity are not considered at this stage.

The Essence of Objects

- A program consists of a *class library* and a *main expression*.
- A class contains *methods* and *instance variables*.
- An *object* is an instance of a particular class.
- The evaluation of an expression yields a value and may modify the state.
- A *value* is either an integer, a boolean, a pointer to an object, or the constant nil.

Expressions

- Object creation.
- Assignment to instance variables.
- Conditionals.
- Iteration.
- Sequencing.
- The self meta-variable.
- Message sends to objects.
- Dynamic verification of the class of an object.

Syntax

- Like SMALLTALK, but with a SIMULAsque syntax.

$$
\begin{array}{lcl}
\textsc{program} & ::= & \textsc{classlist?}\ \textsc{exp} \\
\\
\textsc{classlist?} & ::= & \varepsilon \mid \textsc{classlist} \\
\textsc{classlist} & ::= & \textsc{class} \mid \textsc{classlist}\ \textsc{class} \\
\\
\textsc{class} & ::= & \textbf{class}\ \textsc{id}\ \textsc{varlist?}\ \textsc{methodlist?}\ \textbf{end} \mid \\
 & & \textbf{class}\ \textsc{id}\ \textbf{is}\ \textsc{id} \\
\\
\textsc{methodlist?} & ::= & \varepsilon \mid \textsc{methodlist} \\
\textsc{methodlist} & ::= & \textsc{method} \mid \textsc{methodlist}\ \textsc{method} \\
\textsc{method} & ::= & \textbf{method}\ \textsc{id}\ \textsc{formals}\ \textsc{exp}\ \textbf{end} \\
\\
\textsc{varlist?} & ::= & \varepsilon \mid \textsc{varlist} \\
\textsc{varlist} & ::= & \textsc{var} \mid \textsc{varlist}\ \textsc{var} \\
\textsc{var} & ::= & \textbf{var}\ \textsc{dec} \\
\\
\textsc{formals} & ::= & (\ \textsc{declist?}\ ) \\
\textsc{declist?} & ::= & \varepsilon \mid \textsc{declist} \\
\textsc{declist} & ::= & \textsc{dec} \mid \textsc{declist}\ ;\ \textsc{dec} \\
\textsc{dec} & ::= & \textsc{idlist} \\
\textsc{idlist} & ::= & \textsc{id} \mid \textsc{idlist}\ ,\ \textsc{id}
\end{array}
$$

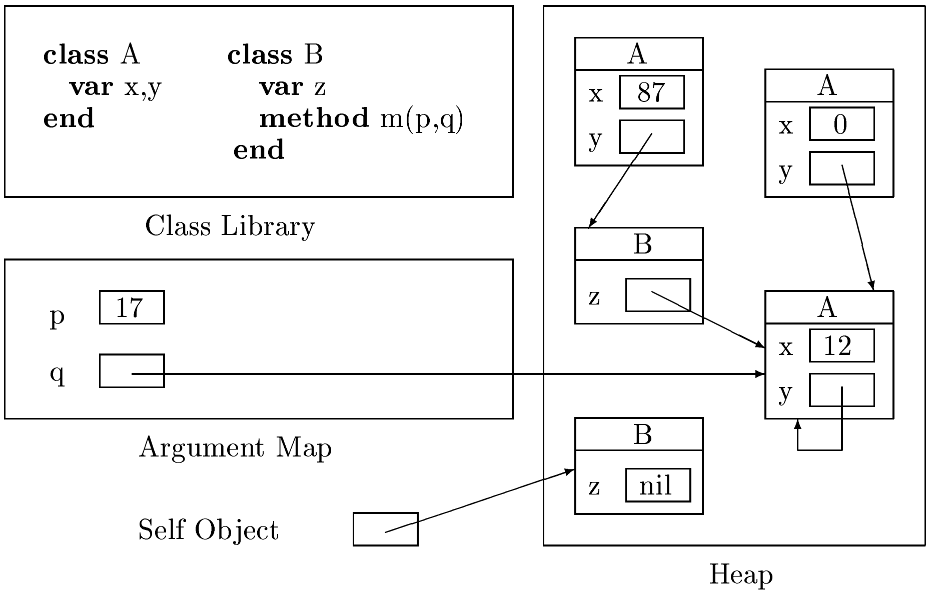
More Syntax

$$
\begin{array}{lcl}
\textsc{exp} & ::= & \textsc{int} \mid \\
 & & \textsc{exp binop exp} \mid \\
 & & \text{false} \mid \\
 & & \text{true} \mid \\
 & & \textsc{exp}\ \text{not} \mid \\
 & & \textsc{id} := \textsc{exp} \mid \\
 & & \textsc{exp}\ ;\ \textsc{exp} \mid \\
 & & \textbf{if}\ \textsc{exp}\ \textbf{then}\ \textsc{exp}\ \textbf{else}\ \textsc{exp}\ \textbf{end} \mid \\
 & & \textbf{while}\ \textsc{exp}\ \textbf{do}\ \textsc{exp}\ \textbf{end} \mid \\
 & & \text{nil} \mid \\
 & & \text{self} \mid \\
 & & \textsc{id}\ \textbf{new} \mid \\
 & & \textsc{exp}\ \textbf{class new} \mid \\
 & & \textsc{exp}\ \textbf{instance-of}\ \{\ \textsc{idlist}\ \} \mid \\
 & & \textsc{id} \mid \\
 & & \textsc{exp}\ .\ \textsc{id}\ (\ \textsc{explist}?\ ) \mid \\
 & & (\ \textsc{exp}\ ) \\
\textsc{binop} & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{=} \mid \text{and} \mid \text{or} \mid \texttt{<} \\
\textsc{explist}? & ::= & \varepsilon \mid \textsc{explist} \\
\textsc{explist} & ::= & \textsc{exp} \mid \textsc{explist}\ ,\ \textsc{exp} \\
 & & \\
\textsc{int} & ::= & \textsc{digit} \mid \textsc{digit int} \\
\textsc{digit} & ::= & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\
\textsc{id} & ::= & \textsc{letter} \mid \textsc{id letter} \mid \textsc{id digit} \\
\textsc{letter} & ::= & a \mid b \mid \ldots \mid z \mid A \mid B \mid \ldots \mid Z
\end{array}
$$

The State

- A BOPL expression is evaluated in a *state* consisting of:
  - the class library;
  - the heap;
  - the argument map; and
  - the self object.
- The *value* of an expression is either:
  - the constant nil;
  - an integer;
  - a boolean; or
  - a pointer to an object.
- An *object* is a record that contains:
  - the name of its class; and
  - a value for each instance variable.

A Possible State



- Plus run-time stack.
- Naïve implementation.

Run-Time Errors

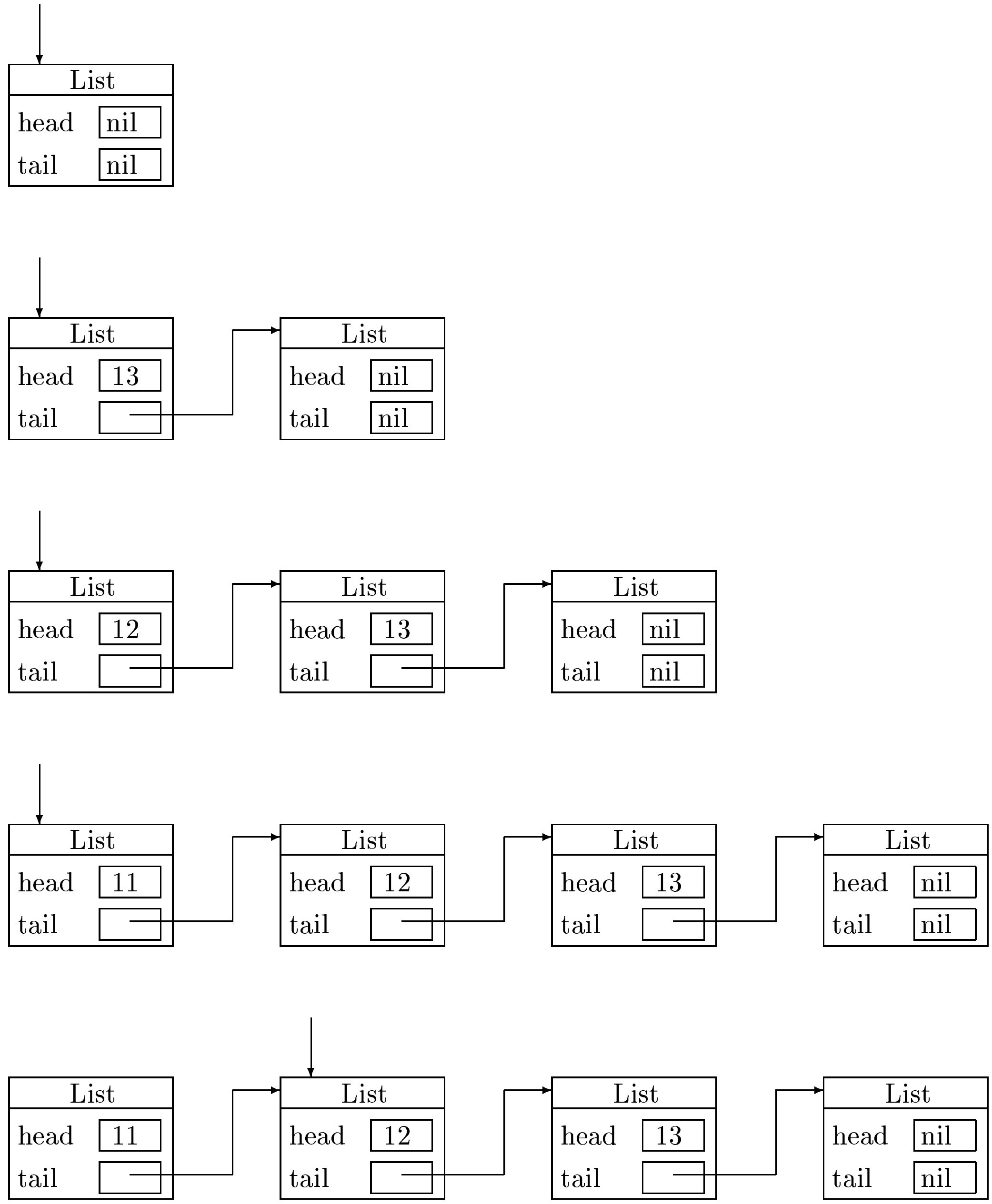
- The possible run-time errors are:
  - type mismatch in unary and binary operators;
  - conditions in **if** and **while** not evaluating to booleans;
  - the class of a receiver not implementing a method for the message it is being sent; or
  - the receiver evaluating to nil.
- The last two are combined known as *message-not-understood*.

Lists

```
class List
    var head, tail
    method Set(h, t)
        head := h;
        tail := t;
        self
    end
    method Cons(h)
        (self class new).Set(h, self)
    end
    method Car()
        head
    end
    method Cdr()
        tail
    end
end
```

```
(List new).Cons(13).Cons(12).Cons(11).Cdr().Car()
```

Evaluation of Lists



3 Type Checking

What is a Type?

- A type is an *invariant*.
- If a variable v contain only integer values, then we may say that v has type Int.
- Thus, the declaration

  var v: Int

  specifies an invariant about v.
- A program in which all type declarations are valid invariants is said to be *type correct*.
- This is formally undecidable, but can be approximated.
- If a type checker can verify that a program is type correct, then the program is *statically* type correct.

Why Types?

An untyped program may be:

- Unreadable.

  Types provide documentation;

  "Well-typed programs are more readable"

- Unreliable.

  Types provide a safety guarantee;

  "Well-typed programs cannot go wrong"

- Inefficient.

  Types enable optimizations;

  "Well-typed programs are faster"

World-Assumptions

- The *closed-world* assumption:

  All parts of the program are known at compile-time. Type checking is global for the entire program.

- The *open-world* assumption:

  Some parts of the program are unknown at compile-time. Type checking is local for each module.

- Both make sense, but at different stages of the development process.

- This has significant influence on typing issues.

Types are Sets of Classes

- The type of E is denoted $[\![E]\!]$.
- $[\![E]\!] = \{A, B, C, \ldots\}$.
- Invariant: E can only evaluate to nil or an instance of a class mentioned in $[\![E]\!]$.
- Generalizes the "class types" of C++, SIMULA, BETA, and Eiffel (they correspond to special sets).
- Explains subtyping, dynamic typing, and the type of nil.
- Note: only values have types, and methods are not values.
- Limitation: no type of *even* integers, unless we create a subclass.

What is Subtyping?

- When can the assignment x := E be allowed?
- The invariant on x must be respected.
- Clearly, $[\![x]\!] \supseteq [\![E]\!]$ is sufficient.
- So, subtyping is set inclusion.
- This is required of all statically type correct programs.

What is the Type of nil?

- It is $\{\}$ (the empty set).
- This satisfies the invariant:

  If nil evaluates to an instance (which it does not), then the class of that instance is in $\{\}$.
- The assignment x := nil is always allowed, since $[\![\mathrm{x}]\!] \supseteq \{\}$.
- This becomes awkward if types are just classes: nil must then be an instance of a paradoxical common subclass of all classes.

Type Annotations

- They are placed at e.g. instance variables:

  var x: {A,B,C}

  Only instances of the classes A, B, C can get stored in the variable x.

- They have consequences throughout the program:

  $$\underset{\{A,B,C\}}{\overset{E}{\uparrow}}\ .\ m(\ \ldots)$$

  Control will be passed to the implementation of m in either A, B, or C.

- Uses only notions already present in the programming language.

- No elaborate language of type expressions must be mastered.

Safety Guarantee

- Avoid the error: *message-not-understood*.
- When the type of the receiver is known:

  $$\begin{array}{c} E \;\; . \; m(\,\ldots) \\ \uparrow \\ \{A,B,C\} \end{array}$$

  check that the classes A, B, and C each implement a method named m.
- The safety of message sends at run-time is then guaranteed at compile-time.
- But the receiver may still be nil (requires data flow analysis).

Optimization

- With safety guarantee, leave out checks for message-not-understood.
- Inline message sends if type of receiver is a singleton.
- Reject message sends if type of receiver is empty.
- Use hash functions to dispatch messages.
- Discard "dead code":

  $$\underset{\{A,B,C\}}{\underset{\uparrow}{E}}\ .\ m(\,\ldots)$$

  e.g. implementations of m in classes other than A, B, and C.

Constraint Technique

- Types must be *declared*, *propagated*, and *verified*.
- Constraints give a uniform presentation.
- Each expression E has a type variable $[\![E]\!]$.
- From the syntax of a program we generate constraints that express:
  - the declared type information;
  - the propagation of types; and
  - the requirements for static type correctness.
- The program is statically type correct if these constraints are solvable.

Constraints for Pascal

- Program:

var x,y: Integer;
var z: **array of** Boolean;

writeln(x+y);
z[87] := true;

- Type constraints:

decl	$[\![\mathrm{x}]\!] = \mathrm{Integer}$
decl	$[\![\mathrm{y}]\!] = \mathrm{Integer}$
decl	$[\![\mathrm{z}]\!] = \textbf{array of } \mathrm{Boolean}$
vfy	$[\![\mathrm{x}]\!] = [\![\mathrm{y}]\!]$
prop	$[\![\mathrm{x}]\!] = \mathrm{Integer} \Rightarrow [\![\mathrm{x+y}]\!] = \mathrm{Integer}$
prop	$[\![\mathrm{x}]\!] = \mathrm{Real} \Rightarrow [\![\mathrm{x+y}]\!] = \mathrm{Real}$
prop	$[\![87]\!] = \mathrm{Integer}$
prop	$[\![\mathrm{true}]\!] = \mathrm{Boolean}$
vfy	$[\![\mathrm{z[87]}]\!] = [\![\mathrm{true}]\!]$
prop	$[\![\mathrm{z}]\!] = \textbf{array of } [\![\mathrm{z[87]}]\!]$

Annotations in BOPL

- For now, types are finite sets of classes.
- We will attach types three places in programs.
- At instance variables; the associated invariant is that the instance variable can only contain nil or the values described by the type.
- At formal arguments; the associated invariant is the same as for instance variables.
- At method results; the associated invariant is that the body of the method can only evaluate to nil or the vales described by the type.

An Annotated Program

```
class A
    var x: A
    var y: {A,B,Int}
    method m(f: B) returns B
        f.m(x)
    end
end

class B
    method m(g: A) returns B
        self
    end
end

(A new).m(B new).m(nil)
```

Constraints for BOPL

- Type variables as before.
- Assume that all variable and argument names are distinct (rename if necessary).
- Use the notation:

  $$[\![\mathrm{x}]\!],\ [\![\mathrm{f}]\!],\ \text{and}\ [\![\mathrm{E}]\!]$$

  for the type variables associated with the instance variable x, the formal argument f, and the expression E.
- Use the notation $[\![\text{self-C}]\!]$ for the self meta-variable in the class C.
- Generate constraints from syntax of expressions.

Type Constraints

1) "INT". For any integer constant k we have:

$$[\![\mathrm{k}]\!] = \{\mathrm{Int}\}$$

2) "true" and "false". For the boolean constants we have:

$$[\![\mathrm{true}]\!] = \{\mathrm{Bool}\} \;\text{ and }\; [\![\mathrm{false}]\!] = \{\mathrm{Bool}\}$$

3) "nil". For the nil constant we have the constraint:

$$[\![\mathrm{nil}]\!] = \{\}$$

4) "EXP_1 + EXP_2". For the addition we have:

$$\begin{aligned}[\![\mathrm{EXP}_1]\!] &= \{\mathrm{Int}\}\\ [\![\mathrm{EXP}_2]\!] &= \{\mathrm{Int}\}\\ [\![\mathrm{EXP}_1 + \mathrm{EXP}_2]\!] &= \{\mathrm{Int}\}\end{aligned}$$

11) "EXP not". For the negation we have:

$$\begin{aligned}[\![\mathrm{EXP}]\!] &= \{\mathrm{Bool}\}\\ [\![\mathrm{EXP\ not}]\!] &= \{\mathrm{Bool}\}\end{aligned}$$

Type Constraints

12) "$\text{ID} := \text{EXP}$". For the assignment we have:

$$\llbracket \text{EXP} \rrbracket \subseteq \llbracket \text{ID} \rrbracket$$
$$\llbracket \text{ID} := \text{EXP} \rrbracket = \llbracket \text{EXP} \rrbracket$$

13) "$\text{EXP}_1 \,;\, \text{EXP}_2$". For the sequence we have:

$$\llbracket \text{EXP}_1 \,;\, \text{EXP}_2 \rrbracket = \llbracket \text{EXP}_2 \rrbracket$$

14) "**if** EXP_1 **then** EXP_2 **else** EXP_3 **end**". For the conditional we have:

$$\llbracket \text{EXP}_1 \rrbracket = \{\text{Bool}\}$$
$$\llbracket \textbf{if } \text{EXP}_1 \textbf{ then } \text{EXP}_2 \textbf{ else } \text{EXP}_3 \textbf{ end} \rrbracket$$
$$= \llbracket \text{EXP}_2 \rrbracket \cup \llbracket \text{EXP}_3 \rrbracket$$

15) "**while** EXP_1 **do** EXP_2 **end**". For the iteration we have:

$$\llbracket \text{EXP}_1 \rrbracket = \{\text{Bool}\}$$
$$\llbracket \textbf{while } \text{EXP}_1 \textbf{ do } \text{EXP}_2 \textbf{ end} \rrbracket = \{\}$$

Type Constraints

16) "self". For the self meta-variable we have:

$$[\![\text{self-C}]\!] = \{\text{C}\}$$

17) "ID **new**". For object creation we have

$$[\![\text{ID } \mathbf{new}]\!] = \{\text{ID}\}$$

18) "EXP **class new**". For indirect object creation we have:

$$[\![\text{EXP}]\!] \subseteq \{\textit{all classes}\}$$
$$[\![\text{EXP } \mathbf{class\ new}]\!] = [\![\text{EXP}]\!]$$

19) "EXP **instance-of** {IDLIST}". For dynamic check of class membership we have:

$$[\![\text{EXP } \mathbf{instance\text{-}of} \ \{ \text{ IDLIST } \}]\!] = \{ \text{ IDLIST } \}$$

20) "ID". For a variable or a formal argument we have:

$$[\![\text{ID}]\!] = \textit{the declared type}$$

Type Constraints

21) "$\textsc{exp}$. $\textsc{id}$ ($\textsc{exp}_1$, ..., $\textsc{exp}_n$)". For the message send, the constraints are:

$$[\![\textsc{exp}]\!] \subseteq \{\textit{all legal receiver classes}\}$$

$$\mathrm{C} \in [\![\textsc{exp}]\!] \Rightarrow \begin{cases} [\![\textsc{exp}_1]\!] \subseteq [\![\textsc{id}_1]\!] \\ \vdots \\ [\![\textsc{exp}_n]\!] \subseteq [\![\textsc{id}_n]\!] \\ [\![\textsc{exp}.\textsc{id}(\textsc{exp}_1,\ldots,\textsc{exp}_n)]\!] \supseteq \textsc{type} \supseteq [\![\textsc{exp}_0]\!] \end{cases}$$

where C implements a method for the message:

method $\textsc{id}(\textsc{id}_1{:}\textsc{type}_1, \ldots, \textsc{id}_n{:}\textsc{type}_n)$ **returns** $\textsc{type}$
$\textsc{exp}_0$
end

An Example

```
class A
    var x: A
    var y: {A,B,Int}
    method m(f: B) returns B
        f.m(x)
    end
end

class B
    method m(g: A) returns B
        self
    end
end

(A new).m(B new).m(nil)
```

Types for all subexpressions:

$\llbracket \mathrm{x} \rrbracket = \llbracket \mathrm{g} \rrbracket = \{\mathrm{A}\}$
$\llbracket \mathrm{y} \rrbracket = \{\mathrm{A,B,Int}\}$
$\llbracket \mathrm{f} \rrbracket = \{\mathrm{B}\}$
$\llbracket \mathrm{f.m(x)} \rrbracket = \{\mathrm{B}\}$
$\llbracket \text{self-B} \rrbracket = \{\mathrm{B}\}$
$\llbracket \mathrm{A}\ \mathbf{new} \rrbracket = \{\mathrm{A}\}$
$\llbracket \mathrm{B}\ \mathbf{new} \rrbracket = \{\mathrm{B}\}$
$\llbracket (\mathrm{A}\ \mathbf{new}).\mathrm{m}(\mathrm{B}\ \mathbf{new}) \rrbracket = \{\mathrm{B}\}$
$\llbracket \mathrm{nil} \rrbracket = \{\}$
$\llbracket (\mathrm{A}\ \mathbf{new}).\mathrm{m}(\mathrm{B}\ \mathbf{new}).\mathrm{m}(\mathrm{nil}) \rrbracket = \{\mathrm{B}\}$

More Example

2)	20)	$[\![\mathrm{x}]\!] = \{\mathrm{A}\}$
3)	20)	$[\![\mathrm{y}]\!] = \{\mathrm{A,B,Int}\}$
4)	20)	$[\![\mathrm{f}]\!] = \{\mathrm{B}\}$
10)	20)	$[\![\mathrm{g}]\!] = \{\mathrm{A}\}$
5)	21)	$[\![\mathrm{f}]\!] \subseteq \{\mathrm{A,B}\}$
5)	22)	$\mathrm{A} \in [\![\mathrm{f}]\!] \Rightarrow \begin{cases} [\![\mathrm{x}]\!] \subseteq [\![\mathrm{f}]\!] \\ [\![\mathrm{f.m(x)}]\!] \supseteq \{\mathrm{B}\} \supseteq [\![\mathrm{f.m(x)}]\!] \end{cases}$
5)	22)	$\mathrm{B} \in [\![\mathrm{f}]\!] \Rightarrow \begin{cases} [\![\mathrm{x}]\!] \subseteq [\![\mathrm{g}]\!] \\ [\![\mathrm{f.m(x)}]\!] \supseteq \{\mathrm{B}\} \supseteq [\![\text{self-B}]\!] \end{cases}$
11)	16)	$[\![\text{self-B}]\!] = \{\mathrm{B}\}$
15)	17)	$[\![\mathrm{A}\ \mathbf{new}]\!] = \{\mathrm{A}\}$
15)	17)	$[\![\mathrm{B}\ \mathbf{new}]\!] = \{\mathrm{B}\}$
15)	21)	$[\![\mathrm{A}\ \mathbf{new}]\!] \subseteq \{\mathrm{A,B}\}$
15)	22)	$\mathrm{A} \in [\![\mathrm{A}\ \mathbf{new}]\!] \Rightarrow$ $\begin{cases} [\![\mathrm{B}\ \mathbf{new}]\!] \subseteq [\![\mathrm{f}]\!] \\ [\![(\mathrm{A}\ \mathbf{new}).\mathrm{m}(\mathrm{B}\ \mathbf{new})]\!] \supseteq \{\mathrm{B}\} \supseteq [\![\mathrm{f.m(x)}]\!] \end{cases}$
15)	22)	$\mathrm{B} \in [\![\mathrm{A}\ \mathbf{new}]\!] \Rightarrow$ $\begin{cases} [\![\mathrm{B}\ \mathbf{new}]\!] \subseteq [\![\mathrm{g}]\!] \\ [\![(\mathrm{A}\ \mathbf{new}).\mathrm{m}(\mathrm{B}\ \mathbf{new})]\!] \supseteq \{\mathrm{B}\} \supseteq [\![\text{self-B}]\!] \end{cases}$
15)	3)	$[\![\mathrm{nil}]\!] = \{\}$
15)	21)	$[\![(\mathrm{A}\ \mathbf{new}).\mathrm{m}(\mathrm{B}\ \mathbf{new})]\!] \subseteq \{\mathrm{A,B}\}$
15)	22)	$\mathrm{A} \in [\![(\mathrm{A}\ \mathbf{new}).\mathrm{m}(\mathrm{B}\ \mathbf{new})]\!] \Rightarrow$ $\begin{cases} [\![\mathrm{nil}]\!] \subseteq [\![\mathrm{f}]\!] \\ [\![(\mathrm{A}\ \mathbf{new}).\mathrm{m}(\mathrm{B}\ \mathbf{new}).\mathrm{m}(\mathrm{nil})]\!] \supseteq \{\mathrm{B}\} \supseteq [\![\mathrm{f.m(x)}]\!] \end{cases}$
15)	22)	$\mathrm{B} \in [\![(\mathrm{A}\ \mathbf{new}).\mathrm{m}(\mathrm{B}\ \mathbf{new})]\!] \Rightarrow$ $\begin{cases} [\![\mathrm{nil}]\!] \subseteq [\![\mathrm{g}]\!] \\ [\![(\mathrm{A}\ \mathbf{new}).\mathrm{m}(\mathrm{B}\ \mathbf{new}).\mathrm{m}(\mathrm{nil})]\!] \supseteq \{\mathrm{B}\} \supseteq [\![\text{self-B}]\!] \end{cases}$

Dynamic Type Checks

- **method** TransPrint(g: {Circle,Rectangle,Triangle})
  returns {Circle,Rectangle,Triangle}

  ```
  g.Translate(100,200);
  g.Print;
  g
  end
  ```

  c:=(App **new**).TransPrint(c)

- Not statically correct, but yields no run-time error.

- The compiler responds:

  – *I know what I am doing. Go away!*
  The program has been rejected for a good reason.

  – *This is not my problem, its yours!*
  The programmer may insert a dynamic check.

  – *I will take care of this!*
  The compiler will insert a dynamic check.

- c:=((App **new**).TransPrint(c) **instance-of** Circle)

When to Use Them

- If dynamic checks are inserted blindly, then any program will be accepted by the compiler.
- Want to make use of the declared type information as far as possible.

1) When a program fragment is statically correct, then it is accepted and no dynamic checks are inserted.
2) When a program fragment is statically incorrect but might succeed at run-time, then it is accepted and dynamic checks are inserted.
3) When a program fragment is statically incorrect and cannot succeed at run-time, then it is rejected.

- Minimize the number of dynamic checks and filter out most meaningless programs.

How to Use Them

- At an assignment $\text{ID} := \text{EXP}$.
- Constraint is $[\![\text{ID}]\!] \supseteq [\![\text{EXP}]\!]$.

1) $[\![\text{ID}]\!] \supseteq [\![\text{EXP}]\!]$. The expression will always evaluate to an object of a legal class.

2) $[\![\text{ID}]\!] \cap [\![\text{EXP}]\!] \neq \{\}$. The expression might evaluate to an object of a legal class. The program is accepted after the assignment has been transformed into $\text{ID} := (\text{EXP}$ **instance-of** $[\![\text{ID}]\!])$.

3) $[\![\text{ID}]\!] \cap [\![\text{EXP}]\!] = \{\}$. The expression cannot possibly evaluate to an object of a legal class. The program is rejected.

- Similarly for arguments and return values.

An Example

- Before:

```
class A
    var x: {Int, Bool}
    var y: Int
    method m(f: A) returns Int
        y := x
    end
end

class B
    method m(g: {A, B}) returns Int
        g.m(g)
    end
end
```

- After:

```
class A
    var x: {Int, Bool}
    var y: Int
    method m(f: A) returns Int
        y := (x instance-of Int)
    end
end

class B
    method m(g: {A, B}) returns Int
        g.m(g instance-of A)
    end
end
```

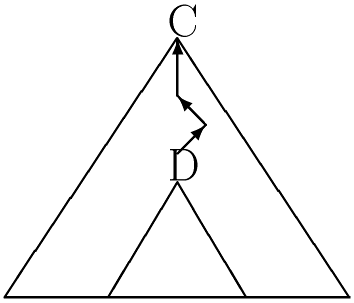
Inheritance: Class Types

- TYPE ::= Void | Int | Bool | $\uparrow$C
- Special sets:
  - the expression Void specifies the *empty set*;
  - the expressions Int and Bool specify the sets {Int} and {Bool}; and
  - the expression $\uparrow$C specifies the set of C and all its subclasses.
- Under the closed-world assumption, they denote finite sets.
- Under the open-world assumption, they denote infinite sets.
- Type constraints can be systematically derived from the general ones.

Properties

- Subset inclusion can be decided:

$\subseteq$	Void	Int	Bool	$\uparrow$C
Void	*true*	*true*	*true*	*true*
Int	*false*	*true*	*false*	*false*
Bool	*false*	*false*	*true*	*false*
$\uparrow$D	*false*	*false*	*false*	D $\leq$ C

- D $\leq$ C is transitive, reflexive subclassing.
- For class types, *subtyping* is *subclassing*.
- 

- Class types are closed under intersection, but *not* under union.

Derived Type Constraints

1) "$\textsc{int}$". For any integer constant k we have:

$$[\![\mathrm{k}]\!] = \mathrm{Int}$$

2) "true" and "false". We have:

$$[\![\mathrm{true}]\!] = \mathrm{Bool} \quad \text{and} \quad [\![\mathrm{false}]\!] = \mathrm{Bool}$$

3) "nil". For the nil constant we have:

$$[\![\mathrm{nil}]\!] = \mathrm{Void}$$

4) "$\textsc{exp}_1$ **+** $\textsc{exp}_2$". For the addition we have:

$$\begin{array}{l} [\![\textsc{exp}_1]\!] = \mathrm{Int} \;\wedge\; [\![\textsc{exp}_2]\!] = \mathrm{Int} \\ [\![\textsc{exp}_1 \;\textbf{+}\; \textsc{exp}_2]\!] = \mathrm{Int} \end{array}$$

12) "$\textsc{id} := \textsc{exp}$". For the assignment we have:

$$[\![\textsc{exp}]\!] \subseteq [\![\textsc{id}]\!] \;\wedge\; [\![\textsc{id} := \textsc{exp}]\!] = [\![\textsc{exp}]\!]$$

Derived Type Constraints

14) "**if** EXP_1 **then** EXP_2 **else** EXP_3 **end**". We have:

$$\llbracket \text{EXP}_1 \rrbracket = \text{Bool}$$
$$\llbracket \textbf{if } \text{EXP}_1 \textbf{ then } \text{EXP}_2 \textbf{ else } \text{EXP}_3 \textbf{ end} \rrbracket$$
$$= \llbracket \text{EXP}_2 \rrbracket = \llbracket \text{EXP}_3 \rrbracket$$

19) "EXP **instance-of** $\Uparrow$ID". For the dynamic check of class membership we have:

$$\llbracket \text{EXP } \textbf{instance-of} \Uparrow \text{ID} \rrbracket = \Uparrow \text{ID}$$

21) Message sends:

$$\llbracket \text{EXP} \rrbracket \subseteq \{\textit{all legal receiver classes}\}$$

This constraint is false if $\llbracket \text{EXP} \rrbracket$ is Int or Bool. If $\llbracket \text{EXP} \rrbracket = \Uparrow \text{C}$ then C must implement a method for the message.

$$\text{C} \in \llbracket \text{EXP} \rrbracket \Rightarrow \ldots$$

Suppose $\llbracket \text{EXP} \rrbracket$ is a cone set of the form $\Uparrow$D for some class D. The condition $\text{C} \in \llbracket \text{EXP} \rrbracket$ can then be verified by checking that C is a subclass of D.

No Surprises

- The derived constraints correspond closely to the type rules for C++, SIMULA, BETA, and EIFFEL.
- Supports the open-world assumption and, hence, separate compilation.
- Gives quite a few more dynamic checks, since the types are much coarser.

Other Type Systems

- A general recipe for new type systems.
- Define new type expressions.
- Show how to expand into sets.
- Under the closed-world assumption, that's it!
- Under the open-world assumption, specialize the type constraints.
- This may involve development of new algorithms for $\subseteq$, $\cap$, and $\cup$ on type expressions.
- Suggestions: union types, interfaces.

Summary

- Types improve the readability, reliability, and efficiency of programs.
- A type is a set of classes, and subtyping is set inclusion.
- A type expresses an invariant of run-time behavior.
- The programs that satisfy this invariant are called type correct.
- Static type correctness is a decidable property that implies type correctness.
- Dynamic type correctness holds when the type invariants are checked at run-time.
- Dynamic typing is necessary for realistic applications.