

Type Checking

CS 208 - Bucknell University

Spring 2012

Outline

Type equivalence

Type checking in Java

Type system

A *type system* is designed such that it prevents programs from running bugs.

A *type checker* is the software implementing the type system (validating or rejecting programs).

- It should not be too strict. The programmer will be frustrated of having his program rejected too often.
- It should not be too permissive. The programmer is happy when the type checker catch bugs.

The type system defines the policy of valid operations. The *type checker* checks that all the variables, values, instructions, function calls have the expected type.

Type equivalence

There are many ways of constructing new types from basic types (\times , $+$, \rightarrow , recursive types). The first question that will arise is the **equivalence** of two types T_1 and T_2 .

Two possibilities:

- Name equivalence
- Structural equivalence

Name equivalence

Definition: Two types T_1 and T_2 are equivalent (name equivalent) if they are the same or if they are defined in the same place.

Example in C:

```
typedef int A, B;
```

```
int x = 1;
```

```
A y = 2;
```

```
B z = 3;
```

```
x = y;
```

```
y = z;
```

Structural equivalence

Two types T_1 and T_2 are structurally equivalent, written $T_1 \equiv T_2$ when the following conditions hold:

Structural equivalence

Two types T_1 and T_2 are structurally equivalent, written $T_1 \equiv T_2$ when the following conditions hold:

- If T_1 and T_2 are primitive types then $T_1 \equiv T_2$ iff T_1 and T_2 are name equivalent.

Structural equivalence

Two types T_1 and T_2 are structurally equivalent, written $T_1 \equiv T_2$ when the following conditions hold:

- If T_1 and T_2 are primitive types then $T_1 \equiv T_2$ iff T_1 and T_2 are name equivalent.
- If $T_1 = A_1 \times B_1$ and $T_2 = A_2 \times B_2$ then $T_1 \equiv T_2$ iff $A_1 \equiv A_2$ and $B_1 \equiv B_2$.

Structural equivalence

Two types T_1 and T_2 are structurally equivalent, written $T_1 \equiv T_2$ when the following conditions hold:

- If T_1 and T_2 are primitive types then $T_1 \equiv T_2$ iff T_1 and T_2 are name equivalent.
- If $T_1 = A_1 \times B_1$ and $T_2 = A_2 \times B_2$ then $T_1 \equiv T_2$ iff $A_1 \equiv A_2$ and $B_1 \equiv B_2$.
- If $T_1 = A_1 + B_1$ and $T_2 = A_2 + B_2$ then $T_1 \equiv T_2$ iff either $A_1 \equiv A_2$ and $B_1 \equiv B_2$, or $A_1 \equiv B_2$ and $B_1 \equiv A_2$.

Structural equivalence

Two types T_1 and T_2 are structurally equivalent, written $T_1 \equiv T_2$ when the following conditions hold:

- If T_1 and T_2 are primitive types then $T_1 \equiv T_2$ iff T_1 and T_2 are name equivalent.
- If $T_1 = A_1 \times B_1$ and $T_2 = A_2 \times B_2$ then $T_1 \equiv T_2$ iff $A_1 \equiv A_2$ and $B_1 \equiv B_2$.
- If $T_1 = A_1 + B_1$ and $T_2 = A_2 + B_2$ then $T_1 \equiv T_2$ iff either $A_1 \equiv A_2$ and $B_1 \equiv B_2$, or $A_1 \equiv B_2$ and $B_1 \equiv A_2$.
- If $T_1 = A_1 \rightarrow B_1$ and $T_2 = A_2 \rightarrow B_2$ then $T_1 \equiv T_2$ iff $A_1 \equiv A_2$ and $B_1 \equiv B_2$.

Structural equivalence

Two types T_1 and T_2 are structurally equivalent, written $T_1 \equiv T_2$ when the following conditions hold:

- If T_1 and T_2 are primitive types then $T_1 \equiv T_2$ iff T_1 and T_2 are name equivalent.
- If $T_1 = A_1 \times B_1$ and $T_2 = A_2 \times B_2$ then $T_1 \equiv T_2$ iff $A_1 \equiv A_2$ and $B_1 \equiv B_2$.
- If $T_1 = A_1 + B_1$ and $T_2 = A_2 + B_2$ then $T_1 \equiv T_2$ iff either $A_1 \equiv A_2$ and $B_1 \equiv B_2$, or $A_1 \equiv B_2$ and $B_1 \equiv A_2$.
- If $T_1 = A_1 \rightarrow B_1$ and $T_2 = A_2 \rightarrow B_2$ then $T_1 \equiv T_2$ iff $A_1 \equiv A_2$ and $B_1 \equiv B_2$.
- otherwise T_1 is not equivalent to T_2 .

Example

$$T_1 = \textit{int}$$

$$T_2 = \textit{float}$$

$$T_3 = A \textit{int} + B T_2 + C T_1$$

$$T_4 = C \textit{int} + B T_2 + A T_1$$

Equality for recursive types

Equality for recursive types is an even more **harder** problem. It generates even more computation.

Equality for recursive types

Equality for recursive types is an even more **harder** problem. It generates even more computation.

$$T_1 = Nil + (S \times T_1)$$

$$T_2 = Nil + (S \times T_2)$$

$$T_3 = Nil + (S \times T_2)$$

Equality for recursive types

Equality for recursive types is an even more **harder** problem. It generates even more computation.

$$T_1 = Nil + (S \times T_1)$$

$$T_2 = Nil + (S \times T_2)$$

$$T_3 = Nil + (S \times T_2)$$

Even worse with **mutually recursive types**:

$$T_1 = Nil + (S \times T_1)$$

$$T_2 = Nil + (S \times T_3)$$

$$T_3 = Nil + (S \times T_2)$$

Equality for recursive types

Equality for recursive types is an even more **harder** problem. It generates even more computation.

$$T_1 = Nil + (S \times T_1)$$

$$T_2 = Nil + (S \times T_2)$$

$$T_3 = Nil + (S \times T_2)$$

Even worse with **mutually recursive types**:

$$T_1 = Nil + (S \times T_1)$$

$$T_2 = Nil + (S \times T_3)$$

$$T_3 = Nil + (S \times T_2)$$

Structural equivalence for recursive types is a nightmare.

Structural equivalence vs Name equivalence

- Structural equivalence brings confusion. You might have types that are equivalent unexpectedly.
- Name equivalence is used by most programming languages, in particular Java.

Type checking

Case study on Java

Definition. A **type checker** insures that all types are used as they should be.

Let us study the type checking of Java looking at the interaction between **overloading** and **implicit coercion**.

Case study on Java

What is the output of this program ?

```
class TestTC {  
  
    public static void main(String args[]){  
        int a = 3;  
        int    five1 = 5;  
        double five2 = 5.0;  
        double r1;  
        double r2;  
  
        r1 = (five1 / (five2 / a));  
        r2 = (five2 / (five1 / a));  
  
        System.out.println("R" + 1 + " = " + r1);  
        System.out.println("R" + 2 + " = " + r2);  
    }  
}
```

The four operations of type checking

The type checking of an expression is a precise sequence of steps which can be of 4 kinds:

The four operations of type checking

The type checking of an expression is a precise sequence of steps which can be of 4 kinds:

1. Checking. *What function is that ?*

The four operations of type checking

The type checking of an expression is a precise sequence of steps which can be of 4 kinds:

1. Checking. *What function is that ?*
2. Inference. *What type is that ?*

The four operations of type checking

The type checking of an expression is a precise sequence of steps which can be of 4 kinds:

1. Checking. *What function is that ?*
2. Inference. *What type is that ?*
3. Equivalence. *Are those types the same ?*

The four operations of type checking

The type checking of an expression is a precise sequence of steps which can be of 4 kinds:

1. Checking. *What function is that ?*
2. Inference. *What type is that ?*
3. Equivalence. *Are those types the same ?*
4. Coercion. *Can we fix things when equivalence fails ?*

1 Checking. *What function is that ?*

Consider the following function:

```
int foo(int x) { ... }
```

1 Checking. *What function is that ?*

Consider the following function:

```
int foo(int x) { ... }
```

Here are examples of when to use *checking*:

```
foo(3);  
CHECK foo(int) -> int  
// good
```

```
foo(4.5);  
CHECK foo(int) -> int  
// bad, there is no foo which takes a double
```

1 Checking. *What function is that ?*

```
foo(3,4,5);  
CHECK foo(int) -> int  
// bad, there is no foo with 3 parameters
```

```
foo();  
CHECK foo(int) -> int  
// bad, there is no foo with no parameters
```

```
bar();  
CHECK foo(int) -> int  
// bad, there is no function bar
```

1 Checking. *What function is that ?*

3 + 8

```
CHECK int + int -> int // good
```

3.4 + 4.5

```
CHECK int + int -> int
```

```
// bad, the arguments are not int, try again
```

```
CHECK double + double -> double
```

```
// good, this is the one we keep
```

"hi" + "there"

```
CHECK int + int -> int
```

```
// bad, the arguments are not int, try again
```

```
CHECK double + double -> double
```

```
// bad, no doubles neither
```

```
CHECK String + String ->String
```

```
// good this is the one we keep
```

1 Checking. *What function is that ?*

3 + 8

```
CHECK int + int -> int // good
```

3.4 + 4.5

```
CHECK int + int -> int
```

```
// bad, the arguments are not int, try again
```

```
CHECK double + double -> double
```

```
// good, this is the one we keep
```

"hi" + "there"

```
CHECK int + int -> int
```

```
// bad, the arguments are not int, try again
```

```
CHECK double + double -> double
```

```
// bad, no doubles neither
```

```
CHECK String + String ->String
```

```
// good this is the one we keep
```

Java has a specific order to try functions with the same name (like the + function).

2 Inference. *What type is that ?*

It consists in finding the type of a variable, values, or the return type of a function, an operator.

2 Inference. *What type is that ?*

It consists in finding the type of a variable, values, or the return type of a function, an operator.

Example:

```
int foo(int x){ ... }  
int x;
```

Here is what gets inferred:

2 Inference. *What type is that ?*

It consists in finding the type of a variable, values, or the return type of a function, an operator.

Example:

```
int foo(int x){ ... }  
int x;
```

Here is what gets inferred:

```
x = 5;  
INFER x int  
INFER 5 int
```

```
foo(4)  
INFER 4 int  
INFER foo returns int
```

2 Inference. *What type is that ?*

4 + 5

INFER 4 int

INFER 5 int

INFER + returns int

3.4 + 6.3

INFER 3.4 double

INFER 6.3 double

INFER + returns double

Remark that many steps are missing, only the inference steps are written.

3 Equivalence. *Are those types the same ?*

Find out if two types are actually the same type.

Reminder: there are 2 type equivalence, *name equivalence* and *structural equivalence*.

Java uses *name equivalence*.

3 Equivalence. *Are those types the same ?*

Find out if two types are actually the same type.

Reminder: there are 2 type equivalence, *name equivalence* and *structural equivalence*.

Java uses *name equivalence*.

Example:

```
int foo (int z) { ... }  
void bar (int z, String w) { ... }  
int x;
```

3 Equivalence. *Are those types the same ?*

Find out if two types are actually the same type.

Reminder: there are 2 type equivalence, *name equivalence* and *structural equivalence*.

Java uses *name equivalence*.

Example:

```
int foo (int z) { ... }  
void bar (int z, String w) { ... }  
int x;
```

```
x = 5;
```

```
INFER 5 int
```

```
INFER x int
```

```
EQUIV int (the 5) and int (the x)? yes
```

3 Equivalence. *Are those types the same ?*

```
foo(5)
```

```
INFER 5 int
```

```
CHECK foo(int) -> int
```

```
  EQUIV int (the 5) and int (first param)? yes
```

3 Equivalence. *Are those types the same ?*

```
foo(5)
```

```
INFER 5 int
```

```
CHECK foo(int) -> int
```

```
  EQUIV int (the 5) and int (first param)? yes
```

```
x + 4
```

```
INFER x int
```

```
INFER 4 int
```

```
CHECK int + int -> int
```

```
  EQUIV int(the x) and int(first param) ? yes
```

```
  EQUIV int(the 4) and int(snd param) ? yes
```

```
INFER + returns int
```

3 Equivalence. *Are those types the same ?*

```
bar(4, "hi")
```

3 Equivalence. *Are those types the same ?*

```
bar(4,"hi")  
INFER 4 int
```

3 Equivalence. *Are those types the same ?*

```
bar(4,"hi")
```

```
INFER 4 int
```

```
INFER "hi" String
```

3 Equivalence. *Are those types the same ?*

```
bar(4,"hi")  
INFER 4 int  
INFER "hi" String  
CHECK bar(int, String) -> void
```

3 Equivalence. *Are those types the same ?*

```
bar(4,"hi")  
INFER 4 int  
INFER "hi" String  
CHECK bar(int, String) -> void
```

```
EQUIV int (4) int (first param)? yes
```

```
EQUIV String ("hi") String (snd param)? yes
```

3 Equivalence. *Are those types the same ?*

```
bar(4,"hi")
```

```
INFER 4 int
```

```
INFER "hi" String
```

```
CHECK bar(int, String) -> void
```

```
    EQUIV int (4) int (first param)? yes
```

```
    EQUIV String ("hi") String (snd param)? yes
```

```
INFER bar returns void
```

4 Coercion. *Can we fix things when equivalence fails ?*

Tries to change the types as needed.

Example:

4 Coercion. *Can we fix things when equivalence fails ?*

Tries to change the types as needed.

Example:

```
int x;  
double y;
```

4 Coercion. *Can we fix things when equivalence fails ?*

Tries to change the types as needed.

Example:

```
int x;
```

```
double y;
```

```
y = 4;
```

4 Coercion. *Can we fix things when equivalence fails ?*

Tries to change the types as needed.

Example:

```
int x;
```

```
double y;
```

```
y = 4;
```

```
INFER 4 int
```

4 Coercion. *Can we fix things when equivalence fails ?*

Tries to change the types as needed.

Example:

```
int x;
```

```
double y;
```

```
y = 4;
```

```
INFER 4 int
```

```
INFER y double
```

4 Coercion. *Can we fix things when equivalence fails ?*

Tries to change the types as needed.

Example:

```
int x;
```

```
double y;
```

```
y = 4;
```

```
INFER 4 int
```

```
INFER y double
```

```
EQUIV double (the y) and int (the 4)?
```

4 Coercion. *Can we fix things when equivalence fails ?*

Tries to change the types as needed.

Example:

```
int x;
```

```
double y;
```

```
y = 4;
```

```
INFER 4 int
```

```
INFER y double
```

```
EQUIV double (the y) and int (the 4)? no
```

4 Coercion. *Can we fix things when equivalence fails ?*

Tries to change the types as needed.

Example:

```
int x;
```

```
double y;
```

```
y = 4;
```

```
INFER 4 int
```

```
INFER y double
```

```
EQUIV double (the y) and int (the 4)? no
```

```
COERCE int to double ?
```

4 Coercion. *Can we fix things when equivalence fails ?*

Tries to change the types as needed.

Example:

```
int x;
```

```
double y;
```

```
y = 4;
```

```
INFER 4 int
```

```
INFER y double
```

```
EQUIV double (the y) and int (the 4)? no
```

```
COERCE int to double ? yes
```

4 Coercion. *Can we fix things when equivalence fails ?*

Tries to change the types as needed.

Example:

```
int x;
```

```
double y;
```

```
y = 4;
```

```
INFER 4 int
```

```
INFER y double
```

```
EQUIV double (the y) and int (the 4)? no
```

```
COERCE int to double ? yes
```

```
Success
```

4 Coercion. *Can we fix things when equivalence fails ?*

```
x = 5.5;
```

4 Coercion. *Can we fix things when equivalence fails ?*

```
x = 5.5;  
INFER 5.5 double
```

4 Coercion. *Can we fix things when equivalence fails ?*

```
x = 5.5;  
INFER 5.5 double  
INFER x int
```

4 Coercion. *Can we fix things when equivalence fails ?*

```
x = 5.5;  
INFER 5.5 double  
INFER x int  
EQUIV int (the x) and double (the 5.5)?
```

4 Coercion. *Can we fix things when equivalence fails ?*

```
x = 5.5;  
INFER 5.5 double  
INFER x int  
EQUIV int (the x) and double (the 5.5)? no
```

4 Coercion. *Can we fix things when equivalence fails ?*

```
x = 5.5;
```

```
INFER 5.5 double
```

```
INFER x int
```

```
EQUIV int (the x) and double (the 5.5)? no
```

```
COERCE double to int ?
```

4 Coercion. *Can we fix things when equivalence fails ?*

```
x = 5.5;
```

```
INFER 5.5 double
```

```
INFER x int
```

```
EQUIV int (the x) and double (the 5.5)? no
```

```
COERCE double to int ? no
```

4 Coercion. *Can we fix things when equivalence fails ?*

```
x = 5.5;  
INFER 5.5 double  
INFER x int  
EQUIV int (the x) and double (the 5.5)? no  
COERCE double to int ? no  
Failure
```

Type checking: All the 4 steps together

3.4 + 5

Type checking: All the 4 steps together

```
3.4 + 5
```

```
INFER 3.4 double
```

Type checking: All the 4 steps together

```
3.4 + 5
```

```
INFER 3.4 double
```

```
INFER 5 int
```

Type checking: All the 4 steps together

3.4 + 5

INFER 3.4 double

INFER 5 int

CHECK int + int -> int

Type checking: All the 4 steps together

3.4 + 5

INFER 3.4 double

INFER 5 int

CHECK int + int -> int

EQUIV double (3.4) int (the parameter) ? no

COERCE double to int ? no

Type checking: All the 4 steps together

3.4 + 5

INFER 3.4 double

INFER 5 int

CHECK int + int -> int

EQUIV double (3.4) int (the parameter) ? no

COERCE double to int ? no

CHECK double + double -> double

Type checking: All the 4 steps together

3.4 + 5

INFER 3.4 double

INFER 5 int

CHECK int + int -> int

EQUIV double (3.4) int (the parameter) ? no

COERCE double to int ? no

CHECK double + double -> double

EQUIV double (3.4) double (the parameter) ? yes

EQUIV int (5) double (the parameter) ? no

COERCE int to double ? yes

Type checking: All the 4 steps together

3.4 + 5

INFER 3.4 double

INFER 5 int

CHECK int + int -> int

EQUIV double (3.4) int (the parameter) ? no

COERCE double to int ? no

CHECK double + double -> double

EQUIV double (3.4) double (the parameter) ? yes

EQUIV int (5) double (the parameter) ? no

COERCE int to double ? yes

INFER returns double

Literals

Type check the following values, also known as literals:

3

3.4

"hi"

'c'

true

0x88f

The compiler has a list of possible + functions. They are in order so we will end up picking the right one.

```
+(int,int): int // takes 2 int, returns an int
```

```
+(double,double): double // takes two double, returns a double
```

```
+(String,String): String // takes two String, returns a String
```

The compiler has a list of possible `+` functions. They are in order so we will end up picking the right one.

```
+(int,int): int // takes 2 int, returns an int
```

```
+(double,double): double // takes two double, returns a double
```

```
+(String,String): String // takes two String, returns a String
```

Type check the following assignment and expressions:

```
x = 3;
```

```
3 + 4
```

```
3 + 2.3
```

```
"hi" + "there"
```

Solution for "hi" + "There"

```
INFER "hi" String
INFER "there" String
CHECK +(int,int): int
  EQUIV String ("hi") and int (first param) ? No
  COERCE String to int ? No
CHECK +(double,double): double
  EQUIV String ("hi") and double (first param) ? No
  COERCE String to double ? No
CHECK +(String,String): String
  EQUIV String ("hi") and String (first param) ? Yes
  EQUIV String ("There") and String (second param) ? Yes
INFER + returns String
```

Combining math and assignment

```
double y = 3 + 2.3;
```

Combining math and assignment

```
double y = 3 + 2.3;
```

Combining function call and assignment

```
int fact(int n){ ... }  
x = fact(4);
```

Combining math and assignment

```
double y = 3 + 2.3;
```

Combining function call and assignment

```
int fact(int n){ ... }  
x = fact(4);
```

INFER 4 int

CHECK fact(int): int

EQUIV int (4) and int (fact param) ? yes

INFER fact returns int

INFER x int

EQUIV int (x) and int (result of fact(4)) ? yes

Success for assignment

Typechecking of return

```
return 1;
```

Typechecking of return

```
return 1;
```

```
INFER return type of this function is int
```

```
INFER 1 int
```

```
EQUIV int int ? Yes
```

Typechecking of return

```
return 1;
```

```
INFER return type of this function is int
```

```
INFER 1 int
```

```
EQUIV int int ? Yes
```

```
return;
```

```
INFER return type of this function is void
```

```
INFER blank void
```

```
EQUIV void void ? Yes
```

Typechecking selection (a.k.a if-statement)

```
if (true) { ... }
```

Typechecking selection (a.k.a if-statement)

```
if (true) { ... }
```

```
INFER true boolean
```

```
EQUIV boolean boolean ? Yes
```

```
Success
```

Typechecking selection (a.k.a if-statement)

```
if (true) { ... }
```

```
INFER true boolean
```

```
EQUIV boolean boolean ? Yes
```

```
Success
```

Combining selection (or iteration) and test

```
if (x < 2) { ... }
```

Typechecking selection (a.k.a if-statement)

```
if (true) { ... }
```

```
INFER true boolean
```

```
EQUIV boolean boolean ? Yes
```

```
Success
```

Combining selection (or iteration) and test

```
if (x < 2) { ... }
```

```
INFER x int
```

Typechecking selection (a.k.a if-statement)

```
if (true) { ... }
```

```
INFER true boolean
```

```
EQUIV boolean boolean ? Yes
```

```
Success
```

Combining selection (or iteration) and test

```
if (x < 2) { ... }
```

```
INFER x int
```

```
INFER 2 int
```

Typechecking selection (a.k.a if-statement)

```
if (true) { ... }
```

```
INFER true boolean
```

```
EQUIV boolean boolean ? Yes
```

```
Success
```

Combining selection (or iteration) and test

```
if (x < 2) { ... }
```

```
INFER x int
```

```
INFER 2 int
```

```
CHECK <(int,int) -> boolean
```

Typechecking selection (a.k.a if-statement)

```
if (true) { ... }
```

```
INFER true boolean
```

```
EQUIV boolean boolean ? Yes
```

```
Success
```

Combining selection (or iteration) and test

```
if (x < 2) { ... }
```

```
INFER x int
```

```
INFER 2 int
```

```
CHECK <(int,int) -> boolean
```

```
EQUIV int int ? yes
```

Typechecking selection (a.k.a if-statement)

```
if (true) { ... }
```

```
INFER true boolean
```

```
EQUIV boolean boolean ? Yes
```

```
Success
```

Combining selection (or iteration) and test

```
if (x < 2) { ... }
```

```
INFER x int
```

```
INFER 2 int
```

```
CHECK <(int,int) -> boolean
```

```
EQUIV int int ? yes
```

```
EQUIV int int ? yes
```

Typechecking selection (a.k.a if-statement)

```
if (true) { ... }
```

```
INFER true boolean
```

```
EQUIV boolean boolean ? Yes
```

```
Success
```

Combining selection (or iteration) and test

```
if (x < 2) { ... }
```

```
INFER x int
```

```
INFER 2 int
```

```
CHECK <(int,int) -> boolean
```

```
EQUIV int int ? yes
```

```
EQUIV int int ? yes
```

```
INFER < returns boolean
```

Back on our problem

What is the output of this program ? (Hint: typecheck the instruction)

```
class TestTC {  
  
    public static void main(String args[]){  
        int a = 3;  
        int    five1 = 5;  
        double five2 = 5.0;  
        double r1;  
        double r2;  
  
        r1 = (five1 / (five2 / a)); /* Typecheck this */  
        r2 = (five2 / (five1 / a)); /* Typecheck this */  
  
        System.out.println("R" + 1 + " = " + r1);  
        System.out.println("R" + 2 + " = " + r2);  
    }  
}
```

Ackermann function

Typecheck the instructions of the following Ackermann function:

```
int ack(int m, int n) {
    int res;

    if (m == 0){
        res = n + 1;
    } else {
        if (n == 0 && m > 0) {
            res = ack(m-1, 0);
        } else {
            res = ack(m-1, ack(m, n-1));
        }
    }

    return res;
}
```