

# Konstrukcija kompilatora

— Semantička analiza: Određivanje tipova —

Milena Vujošević Jančić

[www.matf.bg.ac.rs/~milena](http://www.matf.bg.ac.rs/~milena)

Matematički fakultet, Univerzitet u Beogradu

# Pregled

- 1 Određivanje tipova u izrazima
- 2 Provera tipova u okviru naredbi
- 3 Literatura

## Review from Last Time

```
class MyClass implements MyInterface {
    string myInteger;

    void doSomething() {
        int[] x;
        x = new string;

        x[5] = myInteger * y;
    }
    void doSomething() {

    }
    int fibonacci(int n) {
        return doSomething() + fibonacci(n - 1);
    }
}
```

## Review from Last Time

```
class MyClass implements MyInterface {  
    string myInteger;  
  
    void doSomething() {  
        int[] x;  
        x = new string;  
        x[5] → myInteger * y;  
    }  
    void doSomething() {  
    }  
    int fibonacci(int n) {  
        return doSomething() + fibonacci(n - 1);  
    }  
}
```

Interface not declared

Wrong type

Can't multiply strings

Variable not declared

Can't redefine functions

Can't add void

No main function

## Review from Last Time

```
class MyClass implements MyInterface {  
    string myInteger;  
  
    void doSomething() {  
        int[] x;  
        x = new string;  
        x[5] = myInteger * y;  
    }  
    void doSomething() {  
    }  
    int fibonacci(int n) {  
        return doSomething() + fibonacci(n - 1);  
    }  
}
```

Can't multiply strings

Wrong type

Variable not declared

Can't redefine functions

Can't add void

No main function

## Review from Last Time

```
class MyClass implements MyInterface {  
    string myInteger;  
  
    void doSomething() {  
        int[] x;  
        x = new string; ← Wrong type  
        x[5] → myInteger * y; ← Variable not declared  
    }  
    void doSomething() {  
  
    }  
    int fibonacci(int n) {  
        return doSomething() + fibonacci(n - 1); ← Can't add void  
    }  
}
```

Can't multiply strings

No main function

## Review from Last Time

```
class MyClass implements MyInterface {
    string myInteger;

    void doSomething() {
        int[] x;
        x = new string;
        x[5] = myInteger * y;
    }
    void doSomething() {

    }
    int fibonacci(int n) {
        return doSomething() + fibonacci(n - 1);
    }
}
```

Can't multiply strings

Wrong type

Can't add void

No main function

## Review from Last Time

```
class MyClass implements MyInterface {  
    string myInteger;  
  
    void doSomething() {  
        int[] x;  
        x = new string;  
        x[5] = myInteger * y;  
    }  
    void doSomething() {  
  
    }  
    int fibonacci(int n) {  
        return doSomething() + fibonacci(n - 1);  
    }  
}
```

Can't multiply strings

Wrong type

Can't add void



# Pregled

## 1 Određivanje tipova u izrazima

- Sistem tipova
- Pravila zaključivanja
- Dodavanje dosega
- Tipovi i nasleđivanje
- Tip null
- Ternarni operator

## 2 Provera tipova u okviru naredbi

## 3 Literatura

# Tipovi

- Notacija tipova zavisi od programskog jezika ali uključuje:
  - Skup vrednosti
  - Skup operacija nad tim vrednostima
- Greška u radu sa tipovima se javlja onda kada se primenjuje operacija nad vrednostima koje ne podržavaju tu operaciju

## Vrste provere tipova

- Statička provera tipova
  - Analiziraj program u fazi kompilacije kako bi pokazao da nema grešaka u tipovima
  - Nikada ne dozvoli da se nešto loše desi u fazi izvršavanja
- Dinamička provera tipova:
  - Proveri operacije u fazi izvršavanja, neposredno pre konkretnog izvršavanja
  - Preciznije od statičkog određivanja tipova, ali manje efikasno
- Bez analize tipova
- Dinamičko određivanje tipova omogućava lakše pisanje prototipova, ali statičko određivanje tipova ima manje grešaka

## Sistem tipova

- Pravila koja definišu šta je dozvoljena operacija nad tipovima formiraju sistem tipova.
- U jako tipiziranim jezici svi tipovi moraju da se poklapaju
- U slabo tipiziranim jezicima mogu se desiti greške u tipovima u fazi izvršavanja
- Jako tipizirani jezici su robusni, ali slabo tipizirani jezici su obično brži

## Zaključivanje tipova

- Dva osnovna koraka statičkog zaključivanja tipova:
  - Zaključivanje tipa za svaki izraz na osnovu tipova komponenti izraza
  - Potvrđivanje da tipovi izraza u određenom kontekstu se poklapaju sa očekivanim
- Ovi koraci se često kombinuju u jednom prolazu

## An Example

```
while (numBitsSet(x + 5) <= 10) {  
  
    if (1.0 + 4.0) {  
        /* ... */  
    }  
  
    while (5 == null) {  
        /* ... */  
    }  
  
}
```

## An Example

```
while (numBitsSet(x + 5) <= 10) {  
  
    if (1.0 + 4.0) {  
        /* ... */  
    }  
  
    while (5 == null) {  
        /* ... */  
    }  
  
}
```

## An Example

```
while (numBitsSet(x + 5) <= 10) {  
  
    if (1.0 + 4.0) {  
        /* ... */  
    }  
  
    while (5 == null) {  
        /* ... */  
    }  
  
}
```



## An Example

```
while (numBitsSet(x + 5) <= 10) {  
    if (1.0 + 4.0) {  
        /* ... */  
    }  
    while (5 == null) {  
        /* ... */  
    }  
}
```

*Well-typed  
expression with  
wrong type.*

## An Example

```
while (numBitsSet(x + 5) <= 10) {  
  
    if (1.0 + 4.0) {  
        /* ... */  
    }  
  
    while (5 == null) {  
        /* ... */  
    }  
  
}
```

## An Example

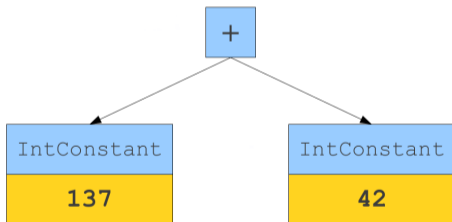
```
while (numBitsSet(x + 5) <= 10) {  
  
    if (1.0 + 4.0) {  
        /* ... */  
    }  
  
    while (5 == null) {  
        /* ... */  
    }  
  
}
```

Expression with  
type error

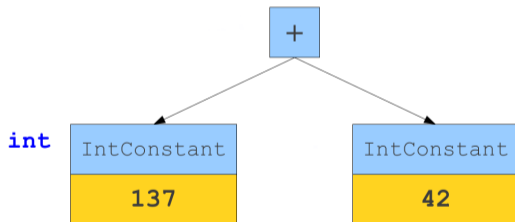
## Zaključivanje tipova

- Kako odrediti tip izraza?
- Ovaj proces odgovara logičkom zaključivanju: počinjemo od skupa aksioma i onda primenjujemo pravila zaključivanja da bi odredili tip izraza.
- Mnogi sistemi tipova se mogu posmatrati kao sistemi za dokazivanje

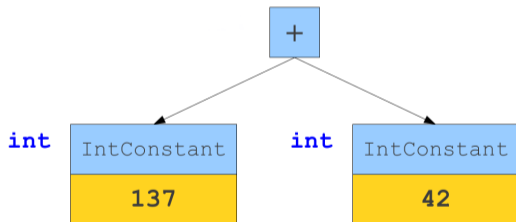
## Primer 1



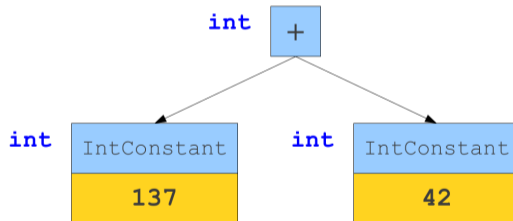
## Primer 1



## Primer 1

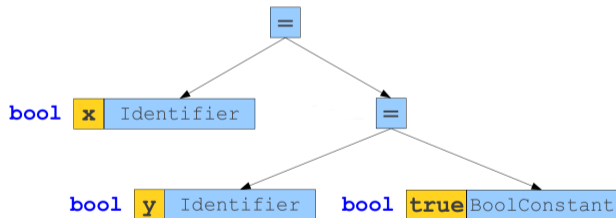


## Primer 1

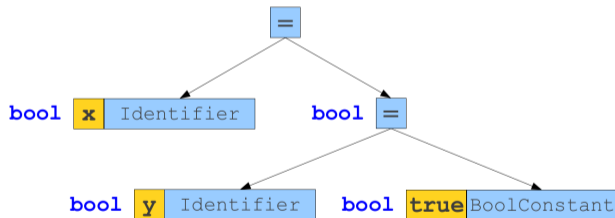




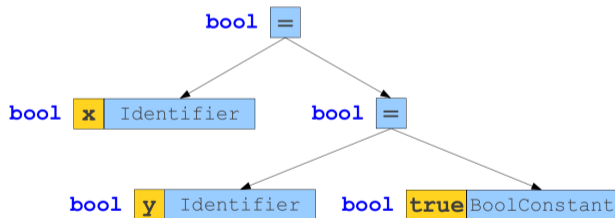
## Primer 2



## Primer 2



## Primer 2



## Jednostavna pravila zaključivanja

- Ako je  $x$  promenljiva koja ima tip  $t$ , izraz  $x$  ima tip  $t$ .
- Ako je  $e$  celobrojna konstanta, onda  $e$  ima tip `int`.
- Ako operandi  $e_1$  i  $e_2$  izraza  $e_1+e_2$  imaju tip `int` i `int`, onda i izraz  $e_1+e_2$  ima tip `int`
- Ovakav zapis treba da se formalizuje i skrati

## Formalan zapis

- Aksiome i pravila zaključivanja mogu se zapisati na sledeći način:

$$\frac{\textit{Preduslov}}{\textit{Postuslov}}$$

**Ako je preduslov tačan, možemo da zaključimo postuslov.**

- Pišemo  $\vdash e : T$  kada možemo da zaključimo da izraz  $e$  ima tip  $T$
- Simbol  $\vdash$  čitamo **možemo da zaključimo**

## Our Starting Axioms

---

$\vdash \text{true} : \text{bool}$

---

$\vdash \text{false} : \text{bool}$

## Some Simple Inference Rules

## Some Simple Inference Rules

$i$  is an integer constant

---

$\vdash i : \text{int}$

$s$  is a string constant

---

$\vdash s : \text{string}$

$d$  is a double constant

---

$\vdash d : \text{double}$



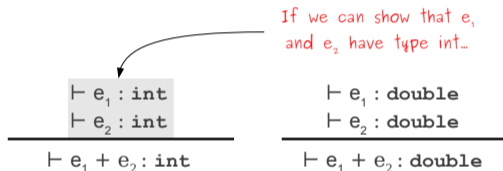
## More Complex Inference Rules

## More Complex Inference Rules

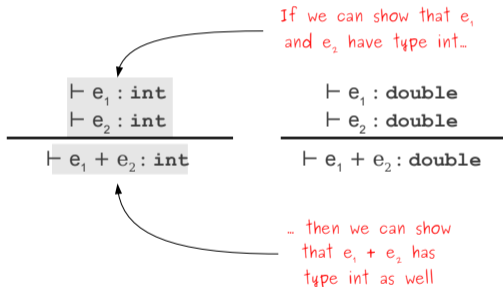
$$\frac{\begin{array}{l} \vdash e_1 : \text{int} \\ \vdash e_2 : \text{int} \end{array}}{\vdash e_1 + e_2 : \text{int}}$$

$$\frac{\begin{array}{l} \vdash e_1 : \text{double} \\ \vdash e_2 : \text{double} \end{array}}{\vdash e_1 + e_2 : \text{double}}$$

## More Complex Inference Rules



## More Complex Inference Rules



## Even More Complex Inference Rules

## Even More Complex Inference Rules

$$\frac{\begin{array}{c} \vdash e_1 : T \\ \vdash e_2 : T \\ \text{T is a primitive type} \end{array}}{\vdash e_1 == e_2 : \text{bool}}$$

$$\frac{\begin{array}{c} \vdash e_1 : T \\ \vdash e_2 : T \\ \text{T is a primitive type} \end{array}}{\vdash e_1 != e_2 : \text{bool}}$$

## Zašto zadavati tipove na ovaj način?

- Ovakve definicije su precizne i ne zavise od bilo koje konkretne implementacije
- Daju maksimalnu fleksibilnost za implementaciju: provera tipova se može implementirati bilo kako sve dok prati zadata pravila
- Ovako zadata pravila omogućavaju formalno dokazivanje svojstva ispravnosti programa

# A Problem



# A Problem

$x$  is an identifier.

---

$\vdash x : ??$

## A Problem

$x$  is an identifier.

---

$\vdash x : ??$

How do we know the  
type of  $x$  if we don't  
know what it refers to?

## An Incorrect Solution

## An Incorrect Solution

x is an identifier.  
x is in scope with type T.

---

$\vdash x : T$

## An Incorrect Solution

x is an identifier.  
x is in scope with type T.

$\vdash x : T$

```
int MyFunction(int x) {  
    {  
        double x;  
    }  
  
    if (x == 1.5) {  
        /* ... */  
    }  
}
```

## An Incorrect Solution

x is an identifier.  
x is in scope with type T.

$\vdash x : T$

```
int MyFunction(int x) {  
  {  
    double x;  
  }  
  if (x == 1.5) {  
    /* ... */  
  }  
}
```

Facts

## An Incorrect Solution

x is an identifier.  
x is in scope with type T.

$\vdash x : T$

```
int MyFunction(int x) {  
  {  
    double x;  
  }  
  if (x == 1.5) {  
    /* ... */  
  }  
}
```

Facts

## An Incorrect Solution

x is an identifier.  
x is in scope with type T.

$\vdash x : T$

```
int MyFunction(int x) {  
  {  
    double x;  
  }  
  
  if (x == 1.5) {  
    /* ... */  
  }  
}
```

Facts

$\vdash x : \text{double}$



## An Incorrect Solution

x is an identifier.  
x is in scope with type T.

$\vdash x : T$

```
int MyFunction(int x) {  
    {  
        double x;  
    }  
    if (x == 1.5) {  
        /* ... */  
    }  
}
```

Facts

$\vdash x : \text{double}$

## An Incorrect Solution

$x$  is an identifier.  
 $x$  is in scope with type  $T$ .

$\vdash x : T$

```
int MyFunction(int x) {  
  {  
    double x;  
  }  
  
  if (x == 1.5) {  
    /* ... */  
  }  
}
```

### Facts

$\vdash x : \text{double}$

$\vdash x : \text{int}$

## An Incorrect Solution

x is an identifier.  
x is in scope with type T.

$\vdash x : T$

```
int MyFunction(int x) {  
    {  
        double x;  
    }  
  
    if (x == 1.5) {  
        /* ... */  
    }  
}
```

### Facts

$\vdash x : \text{double}$

$\vdash x : \text{int}$

## An Incorrect Solution

$x$  is an identifier.  
 $x$  is in scope with type  $T$ .

$\vdash x : T$

$d$  is a double constant

$\vdash d : \text{double}$

```
int MyFunction(int x) {  
  {  
    double x;  
  }  
  
  if (x == 1.5) {  
    /* ... */  
  }  
}
```

### Facts

$\vdash x : \text{double}$

$\vdash x : \text{int}$

## An Incorrect Solution

$x$  is an identifier.  
 $x$  is in scope with type  $T$ .

$\vdash x : T$

$d$  is a double constant

$\vdash d : \text{double}$

```
int MyFunction(int x) {  
    {  
        double x;  
    }  
    if (x == 1.5) {  
        /* ... */  
    }  
}
```

### Facts

$\vdash x : \text{double}$

$\vdash x : \text{int}$

$\vdash 1.5 : \text{double}$

## An Incorrect Solution

x is an identifier.  
x is in scope with type T.

$\vdash x : T$

```
int MyFunction(int x) {  
    {  
        double x;  
    }  
    if (x == 1.5) {  
        /* ... */  
    }  
}
```

### Facts

$\vdash x : \text{double}$

$\vdash x : \text{int}$

$\vdash 1.5 : \text{double}$

## An Incorrect Solution

x is an identifier.  
x is in scope with type T.

$\vdash x : T$

```
int MyFunction(int x) {  
    {  
        double x;  
    }  
    if (x == 1.5) {  
        /* ... */  
    }  
}
```

### Facts

$\vdash x : \text{double}$

$\vdash x : \text{int}$

$\vdash 1.5 : \text{double}$

## An Incorrect Solution

$x$  is an identifier.  
 $x$  is in scope with type  $T$ .

---

$\vdash x : T$

$\vdash e_1 : T$

$\vdash e_2 : T$

$T$  is a primitive type

---

$\vdash e_1 == e_2 : \text{bool}$

```
int MyFunction(int x) {  
    {  
        double x;  
    }  
    if (x == 1.5) {  
        /* ... */  
    }  
}
```

### Facts

$\vdash x : \text{double}$

$\vdash x : \text{int}$

$\vdash 1.5 : \text{double}$



## An Incorrect Solution

$x$  is an identifier.  
 $x$  is in scope with type  $T$ .

---

$\vdash x : T$

$\vdash e_1 : T$

$\vdash e_2 : T$

$T$  is a primitive type

---

$\vdash e_1 == e_2 : \text{bool}$

```
int MyFunction(int x) {  
    {  
        double x;  
    }  
    if (x == 1.5) {  
        /* ... */  
    }  
}
```

### Facts

$\vdash x : \text{double}$

$\vdash x : \text{int}$

$\vdash 1.5 : \text{double}$

$\vdash x == 1.5 : \text{bool}$

## An Incorrect Solution

$x$  is an identifier.  
 $x$  is in scope with type  $T$ .

---

$\vdash x : T$

$\vdash e_1 : T$

$\vdash e_2 : T$

$T$  is a primitive type

---

$\vdash e_1 == e_2 : \text{bool}$

```
int MyFunction(int x) {  
    {  
        double x;  
    }  
    if (x == 1.5) {  
        /* ... */  
    }  
}
```

### Facts

$\vdash x : \text{double}$

$\vdash x : \text{int}$

$\vdash 1.5 : \text{double}$

$\vdash x == 1.5 : \text{bool}$

## Dodavanje dosega u zaključivanje

- Potrebno je da ojačamo naša pravila zaključivanja sa kontekstom tako da se pamti u kojim situacijama su rezultati validni
- Dodavanje dosega:

$$S \vdash e : T$$

**U dosegu  $S$ , izraz  $e$  ima tip  $T$ .**

- Tipove sada dokazujemo relativno za doseg u kojem se nalazimo

## Old Rules Revisited

$$\frac{}{S \vdash \text{true} : \text{bool}}$$
$$\frac{}{S \vdash \text{false} : \text{bool}}$$

$i$  is an integer constant

$$\frac{}{S \vdash i : \text{int}}$$

$s$  is a string constant

$$\frac{}{S \vdash s : \text{string}}$$

$d$  is a double constant

$$\frac{}{S \vdash d : \text{double}}$$
$$S \vdash e_1 : \text{double}$$
$$S \vdash e_2 : \text{double}$$
$$\frac{}{S \vdash e_1 + e_2 : \text{double}}$$
$$S \vdash e_1 : \text{int}$$
$$S \vdash e_2 : \text{int}$$
$$\frac{}{S \vdash e_1 + e_2 : \text{int}}$$

## A Correct Rule

$$\frac{\begin{array}{l} x \text{ is an identifier.} \\ x \text{ is a variable in scope } S \text{ with type } T. \end{array}}{S \vdash x : T}$$

## A Correct Rule

$$\frac{\begin{array}{l} x \text{ is an identifier.} \\ \mathbf{x \text{ is a variable in scope } S} \text{ with type } T. \end{array}}{S \vdash x : T}$$

## Rules for Functions

---

$$S \vdash f(e_1, \dots, e_n) : ??$$

## Rules for Functions

$f$  is an identifier.

---

$S \vdash f(e_1, \dots, e_n) : ??$



## Rules for Functions

$f$  is an identifier.

$f$  is a non-member function in scope  $S$ .

---

$$S \vdash f(e_1, \dots, e_n) : ??$$

## Rules for Functions

$f$  is an identifier.  
 $f$  is a non-member function in scope  $S$ .  
 $f$  has type  $(T_1, \dots, T_n) \rightarrow U$

---

$S \vdash f(e_1, \dots, e_n) : ??$

## Rules for Functions

$f$  is an identifier.  
 $f$  is a non-member function in scope  $S$ .  
 $f$  has type  $(T_1, \dots, T_n) \rightarrow U$   
 $S \vdash e_i : T_i$  for  $1 \leq i \leq n$   

---

 $S \vdash f(e_1, \dots, e_n) : ??$

## Rules for Functions

$f$  is an identifier.  
 $f$  is a non-member function in scope  $S$ .  
 $f$  has type  $(T_1, \dots, T_n) \rightarrow U$   
 $S \vdash e_i : T_i$  for  $1 \leq i \leq n$   

---

 $S \vdash f(e_1, \dots, e_n) : U$

## Rules for Functions

*f* is an identifier.  
*f* is a non-member function in scope *S*.  
*f* has type  $(T_1, \dots, T_n) \rightarrow U$   
 $S \vdash e_i : T_i$  for  $1 \leq i \leq n$   

---

 $S \vdash f(e_1, \dots, e_n) : U$

Read rules like this

## Rules for Arrays

$$S \vdash e_1 : T[]$$
$$S \vdash e_2 : \mathbf{int}$$

---

$$S \vdash e_1[e_2] : T$$

## Rule for Assignment

$$\frac{\begin{array}{l} S \vdash e_1 : T \\ S \vdash e_2 : T \end{array}}{S \vdash e_1 = e_2 : T}$$

## Rule for Assignment

$$\frac{\begin{array}{l} S \vdash e_1 : T \\ S \vdash e_2 : T \end{array}}{S \vdash e_1 = e_2 : T}$$

If **Derived** extends **Base**, will this rule work for this code?

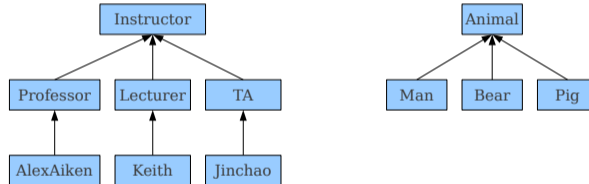
```
Base    myBase;  
Derived myDerived;  
  
myBase = myDerived;
```



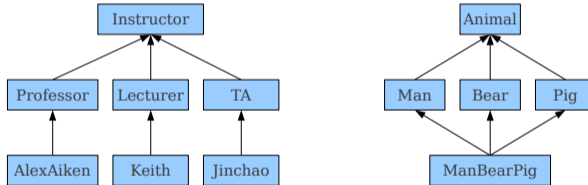
## Tipovi i nasleđivanje

- Potrebno je ubaciti nasleđivanje u sistem tipova
- Važno je uzeti u obzir oblik hijerarhije klasa

## Single Inheritance



## Multiple Inheritance



## Osobine nasleđivanja

**Refleksivnost** Svaki tip nasleđuje samog sebe

**Tranzitivnost** Ako A nasleđuje B i ako B nasleđuje C, onda i A nasleđuje C.

Primer: mačka, sisar, životinja (mačka posredno nasleđuje životinju)

**Antisimetričnost** Ako A nasleđuje B i ako B nasleđuje A, onda su A i B istog tipa.

## Osobine nasleđivanja

**Refleksivnost** Svaki tip nasleđuje samog sebe

**Tranzitivnost** Ako A nasleđuje B i ako B nasleđuje C, onda i A nasleđuje C.

Primer: mačka, sisar, životinja (mačka posredno nasleđuje životinju)

**Antisimetričnost** Ako A nasleđuje B i ako B nasleđuje A, onda su A i B istog tipa.

Dakle, nasleđivanje ima osobine **parcijalnog uređenja na tipovima**.

## Osobine pretvaranja (konvertovanja)

- Ako A nasleđuje B, onda se objekat klase A može **pretvoriti** (konvertovati) u objekat klase B
- Pretvaranje (konvertovanje) je malo širi pojam i ne mora da obuhvata samo klasno nasleđivanje

**Refleksivnost** Svaki tip se može pretvoriti (konvertovati) u samog sebe.

**Tranzitivnost** Ako se A može pretvoriti u B i ako se B može pretvoriti u C, onda se i A može pretvoriti u C.

**Antisimetričnost** Ako se A može pretvoriti u B i ako se B može pretvoriti u A, onda su A i B istog tipa.

## Tipovi i parcijalna uređenja

- Kažemo da je  $A \leq B$  ako se  $A$  može pretvoriti (konvertovati) u  $B$ . Važi:
  - $A \leq A$
  - $A \leq B$  i  $B \leq C$  povlači  $A \leq C$
  - $A \leq B$  i  $B \leq A$  povlači  $A = B$

## Updated Rule for Assignment

---

$$S \vdash e_1 = e_2 : ??$$



## Updated Rule for Assignment

$$\begin{array}{l} S \vdash e_1 : T_1 \\ S \vdash e_2 : T_2 \end{array}$$

---

$$S \vdash e_1 = e_2 : ??$$

## Updated Rule for Assignment

$$\frac{\begin{array}{l} S \vdash e_1 : T_1 \\ S \vdash e_2 : T_2 \\ T_2 \leq T_1 \end{array}}{S \vdash e_1 = e_2 : ??}$$

## Updated Rule for Assignment

$$\frac{\begin{array}{l} S \vdash e_1 : T_1 \\ S \vdash e_2 : T_2 \\ T_2 \leq T_1 \end{array}}{S \vdash e_1 = e_2 : T_1}$$

## Updated Rule for Comparisons

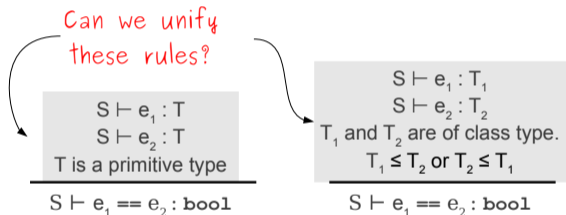
$$\frac{\begin{array}{l} S \vdash e_1 : T \\ S \vdash e_2 : T \\ T \text{ is a primitive type} \end{array}}{S \vdash e_1 == e_2 : \mathbf{bool}}$$

## Updated Rule for Comparisons

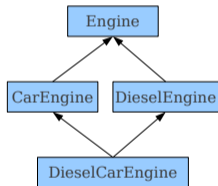
$$\frac{\begin{array}{l} S \vdash e_1 : T \\ S \vdash e_2 : T \\ T \text{ is a primitive type} \end{array}}{S \vdash e_1 == e_2 : \mathbf{bool}}$$

$$\frac{\begin{array}{l} S \vdash e_1 : T_1 \\ S \vdash e_2 : T_2 \\ T_1 \text{ and } T_2 \text{ are of class type.} \\ T_1 \leq T_2 \text{ or } T_2 \leq T_1 \end{array}}{S \vdash e_1 == e_2 : \mathbf{bool}}$$

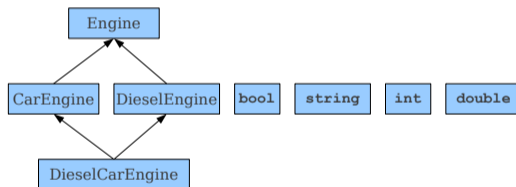
## Updated Rule for Comparisons



## The Shape of Types

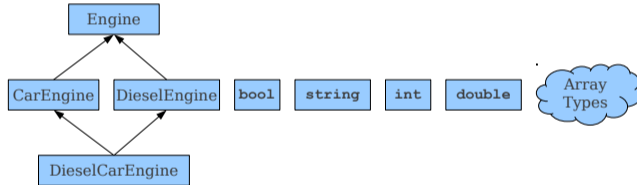


## The Shape of Types





## The Shape of Types



## Proširivanje pretvaranja (konvertovanja)

- Ako je  $A$  osnovni tip ili niz,  $A$  se može pretvoriti samo u samog sebe
- Odnosno, ako su  $A$  i  $B$  neki tipovi, i  $A$  je osnovni tip ili niz, onda važi:
  - $A \leq B$  povlači  $A = B$
  - $B \leq A$  povlači  $A = B$

## Updated Rule for Comparisons

$$\frac{\begin{array}{l} S \vdash e_1 : T \\ S \vdash e_2 : T \\ T \text{ is a primitive type} \end{array}}{S \vdash e_1 == e_2 : \mathbf{bool}}$$

$$\frac{\begin{array}{l} S \vdash e_1 : T_1 \\ S \vdash e_2 : T_2 \\ T_1 \text{ and } T_2 \text{ are of class type.} \\ T_1 \leq T_2 \text{ or } T_2 \leq T_1 \end{array}}{S \vdash e_1 == e_2 : \mathbf{bool}}$$

## Updated Rule for Comparisons

$$\frac{\begin{array}{l} S \vdash e_1 : T \\ S \vdash e_2 : T \\ T \text{ is a primitive type} \end{array}}{S \vdash e_1 == e_2 : \mathbf{bool}}$$
$$\frac{\begin{array}{l} S \vdash e_1 : T_1 \\ S \vdash e_2 : T_2 \\ T_1 \text{ and } T_2 \text{ are of class type.} \\ T_1 \leq T_2 \text{ or } T_2 \leq T_1 \end{array}}{S \vdash e_1 == e_2 : \mathbf{bool}}$$

$$\frac{\begin{array}{l} S \vdash e_1 : T_1 \\ S \vdash e_2 : T_2 \\ T_1 \leq T_2 \text{ or } T_2 \leq T_1 \end{array}}{S \vdash e_1 == e_2 : \mathbf{bool}}$$

## Updated Rule for Comparisons

$$\frac{\begin{array}{c} S \vdash e_1 : T \\ S \vdash e_2 : T \\ T \text{ is a primitive type} \end{array}}{S \vdash e_1 == e_2 : \text{bool}}$$
$$\frac{\begin{array}{c} S \vdash e_1 : T_1 \\ S \vdash e_2 : T_2 \\ T_1 \text{ and } T_2 \text{ are of class type.} \\ T_1 \leq T_2 \text{ or } T_2 \leq T_1 \end{array}}{S \vdash e_1 == e_2 : \text{bool}}$$
$$\frac{\begin{array}{c} S \vdash e_1 : T_1 \\ S \vdash e_2 : T_2 \\ T_1 \leq T_2 \text{ or } T_2 \leq T_1 \end{array}}{S \vdash e_1 == e_2 : \text{bool}}$$

## Updated Rule for Function Calls

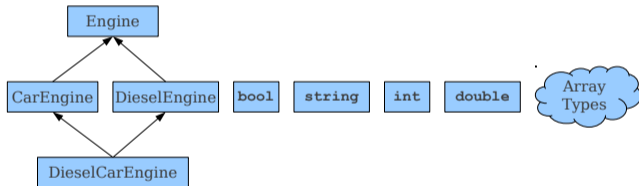
$$\begin{array}{l} f \text{ is an identifier.} \\ f \text{ is a non-member function in scope } S. \\ f \text{ has type } (T_1, \dots, T_n) \rightarrow U \\ S \vdash e_i : R_i \text{ for } 1 \leq i \leq n \\ R_i \leq T_i \text{ for } 1 \leq i \leq n \\ \hline S \vdash f(e_1, \dots, e_n) : U \end{array}$$

## A Tricky Case

---

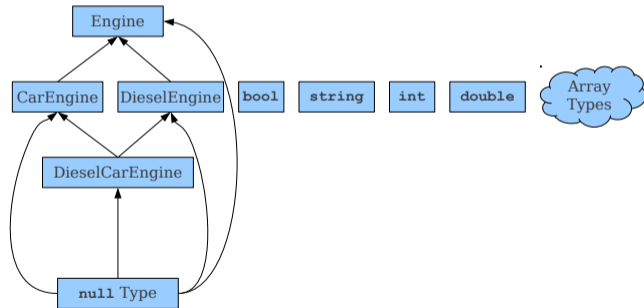
$S \vdash \text{null} : ??$

## Back to the Drawing Board





## Back to the Drawing Board



## Tip null

- Definišemo novi tip koji odgovara literalu null i zovemo ga **null tip**
- Definišemo da važi da je null tip  $\leq A$  za svaki klasni tip  $A$
- Ovaj tip obično nije dostupan programerima, već se koristi samo interno u okviru sistema tipova
- Mnogi programski jezici imaju ovakav tip u svom sistemu tipova

## A Tricky Case

---

$S \vdash \text{null} : ??$

## A Tricky Case

---

$S \vdash \text{null} : \text{null type}$

## A Tricky Case

---

$S \vdash \text{null} : \text{null type}$

## Object-Oriented Considerations

S is in scope of class T.

---

$S \vdash \text{this} : T$

T is a class type.

---

$S \vdash \text{new } T : T$

$S \vdash e : \text{int}$

---

$S \vdash \text{NewArray}(e, T) : T[]$

## Šta nedostaje?

- Za vežbu možete da pokušate da definišete proveru tipova za naredne jezičke konstrukte:
  - Funkcije članice klase
  - Pristup polju klase
  - Različite operatore

## Određivanje tipova za ternarni operator

- Ternarni uslovni operator `?` : izračunava izraz `i` vraća jednu od dve vrednosti
- Može se koristiti sa osnovnim tipovima  
`int x = random() ? 137 : 42;`
- Može se koristiti sa nasleđivanjem  
`Base b = isB ? new Base : new Derived;`
- Kako bi izgledalo određivanje tipa za ovaj izraz?



## A Proposed Rule

---

$S \vdash \text{cond } ? e_1 : e_2 : ??$

## A Proposed Rule

$S \vdash cond : bool$

---

$S \vdash cond \ ? e_1 : e_2 : ??$

## A Proposed Rule

$$S \vdash \text{cond} : \text{bool}$$
$$S \vdash e_1 : T_1$$
$$S \vdash e_2 : T_2$$

---

$$S \vdash \text{cond } ? e_1 : e_2 : ??$$

## A Proposed Rule

$$\frac{\begin{array}{l} S \vdash \text{cond} : \text{bool} \\ S \vdash e_1 : T_1 \\ S \vdash e_2 : T_2 \\ T_1 \leq T_2 \text{ or } T_2 \leq T_1 \end{array}}{S \vdash \text{cond} ? e_1 : e_2 : ??}$$

## A Proposed Rule

$$\frac{\begin{array}{l} S \vdash \text{cond} : \text{bool} \\ S \vdash e_1 : T_1 \\ S \vdash e_2 : T_2 \\ T_1 \leq T_2 \text{ or } T_2 \leq T_1 \end{array}}{S \vdash \text{cond} ? e_1 : e_2 : \max(T_1, T_2)}$$

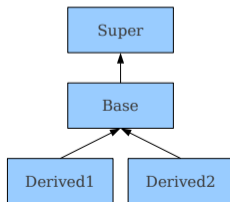
## A Proposed Rule

$$\frac{\begin{array}{l} S \vdash \text{cond} : \text{bool} \\ S \vdash e_1 : T_1 \\ S \vdash e_2 : T_2 \\ T_1 \leq T_2 \text{ or } T_2 \leq T_1 \end{array}}{S \vdash \text{cond} ? e_1 : e_2 : \mathbf{\max(T_1, T_2)}}$$

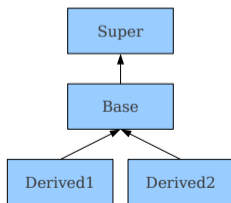
## A Proposed Rule

$$S \vdash cond : \text{bool}$$
$$S \vdash e_1 : T_1$$
$$S \vdash e_2 : T_2$$
$$T_1 \leq T_2 \text{ or } T_2 \leq T_1$$

---

$$S \vdash cond ? e_1 : e_2 : \mathbf{\max(T_1, T_2)}$$


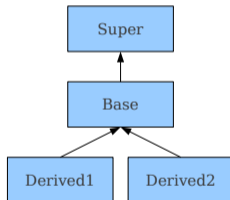
## A Proposed Rule

$$S \vdash cond : \text{bool}$$
$$S \vdash e_1 : T_1$$
$$S \vdash e_2 : T_2$$
$$T_1 \leq T_2 \text{ or } T_2 \leq T_1$$
$$S \vdash cond ? e_1 : e_2 : \text{max}(T_1, T_2)$$


Is this really  
what we want?

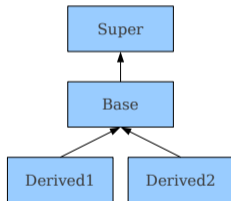


## A Small Problem



$$\frac{\begin{array}{l} S \vdash \text{cond} : \text{bool} \\ S \vdash e_1 : T_1 \\ S \vdash e_2 : T_2 \\ T_1 \leq T_2 \text{ or } T_2 \leq T_1 \end{array}}{S \vdash \text{cond} ? e_1 : e_2 : \max(T_1, T_2)}$$

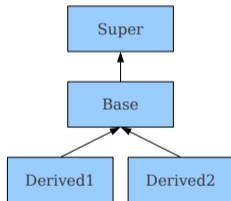
## A Small Problem



$$\frac{\begin{array}{l} S \vdash cond : \text{bool} \\ S \vdash e_1 : T_1 \\ S \vdash e_2 : T_2 \\ T_1 \leq T_2 \text{ or } T_2 \leq T_1 \end{array}}{S \vdash cond ? e_1 : e_2 : \max(T_1, T_2)}$$

```
Base = random() ?  
    new Derived1 : new Derived2;
```

## A Small Problem



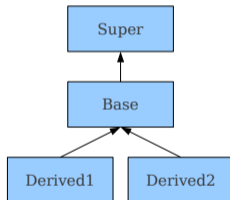
$$\begin{array}{l} S \vdash cond : \text{bool} \\ S \vdash e_1 : T_1 \\ S \vdash e_2 : T_2 \\ \hline T_1 \leq T_2 \text{ or } T_2 \leq T_1 \\ \hline S \vdash cond ? e_1 : e_2 : \max(T_1, T_2) \end{array}$$

```
Base = random() ?  
    new Derived1 : new Derived2;
```

## Najmanja gornja granica

- **Gornja granica** za dva tipa  $A$  i  $B$  je tip  $C$  takav da važi  $A \leq C$  i  $B \leq C$
- **Najmanja gornja granica** za tipove  $A$  i  $B$  je tip  $C$  takav da važi:
  - $C$  je gornja granica za  $A$  i  $B$
  - Ako je  $C'$  gornja granica za  $A$  i  $B$ , onda je  $C \leq C'$
- Kada postoji najmanja gornja granica za  $A$  i  $B$ , to označavamo sa  $A \vee B$
- U kojim situacijama može da ne postoji najmanja gornja granica za  $A$  i  $B$ ?

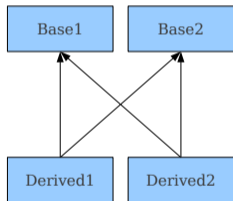
## A Better Rule



$$\frac{\begin{array}{l} S \vdash \text{cond} : \text{bool} \\ S \vdash e_1 : T_1 \\ S \vdash e_2 : T_2 \\ T = T_1 \vee T_2 \end{array}}{S \vdash \text{cond} ? e_1 : e_2 : T}$$

```
Base = random() ?  
    new Derived1 : new Derived2;
```

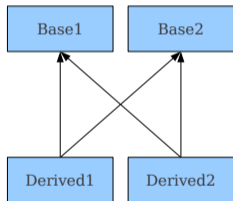
... that still has problems



$$\frac{\begin{array}{l} S \vdash \text{cond} : \text{bool} \\ S \vdash e_1 : T_1 \\ S \vdash e_2 : T_2 \\ T = T_1 \vee T_2 \end{array}}{S \vdash \text{cond} ? e_1 : e_2 : T}$$

```
Base = random() ?  
    new Derived1 : new Derived2;
```

... that still has problems



$$\frac{\begin{array}{l} S \vdash \text{cond} : \text{bool} \\ S \vdash e_1 : T_1 \\ S \vdash e_2 : T_2 \\ \mathbf{T = T_1 \vee T_2} \end{array}}{S \vdash \text{cond} ? e_1 : e_2 : T}$$

```
Base = random() ?  
    new Derived1 : new Derived2;
```

## Višestruko nasleđivanje

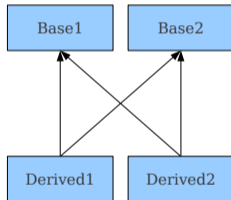
- Hijerarhija tipova u višestrukome nasleđivanju više nema drvoliku strukturu
- Dve klase mogu da nemaju najmanje gornje ograničenje
- To se dešava u C++-u jer imamo višestruko nasleđivanje, ali i u Javi zbog interfejsa



## Minimalno gornje ograničenje

- Gornje ograničenje tipova  $A$  i  $B$  je tip  $C$  ako važi da je  $A \leq C$  i  $B \leq C$
- Minimalno gornje ograničenje tipova  $A$  i  $B$  je tip  $C$  takav da važi
  - $C$  je gornje ograničenje za  $A$  i  $B$
  - Ako je  $C'$  gornje ograničenje od  $A$  i  $B$ , onda ne važi da je  $C' < C$  (dakle,  $C'$  ako je uporedivo, ne sme da bude manje, ali ne mora da bude uporedivo sa  $C$ !)
- Minimalno gornje ograničenje ne mora da bude jedinstveno
- Najmanje gornje ograničenje mora da bude minimalno gornje ograničenje, ali ne važi obratno

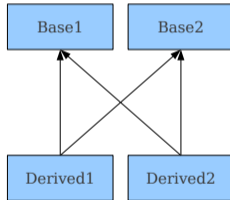
## A Correct Rule



$$\frac{\begin{array}{l} S \vdash \text{cond} : \text{bool} \\ S \vdash e_1 : T_1 \\ S \vdash e_2 : T_2 \\ T \text{ is a minimal upper bound of } T_1 \text{ and } T_2 \end{array}}{S \vdash \text{cond} ? e_1 : e_2 : T}$$

```
Base1 = random() ?  
    new Derived1 : new Derived2;
```

## A Correct Rule


$$\frac{\begin{array}{l} S \vdash cond : \text{bool} \\ S \vdash e_1 : T_1 \\ S \vdash e_2 : T_2 \\ \text{T is a minimal upper bound of } T_1 \text{ and } T_2 \end{array}}{S \vdash cond ? e_1 : e_2 : T}$$

Can prove both that  
expression has type **Base1**  
and that expression has  
type **Base2**.

```
Base1 = random() ?  
new Derived1 : new Derived2;
```

## Zaključak

- Određivanje tipova može da bude veoma nezgodno
- Na određivanje tipova veoma utiče izbor operatora koji se koriste u jeziku
- Na određivanje tipova veoma utiču i dozvoljene konverzije između tipova

# Pregled

- 1 Određivanje tipova u izrazima
- 2 Provera tipova u okviru naredbi
  - Ispravnost naredbi
  - Praktični problemi
  - Preopterećivanje funkcija
  - Kompletnosti i saglasnost sistema tipova
- 3 Literatura

## Provera tipova u okviru naredbi

- Da li su tipovi izraza u okviru naredbi ispravni?
  - Da li `if` naredba ima dobro formiran uslov?
  - Da li `return` naredba vraća adekvatan tip?
  - ...
- Kako se to praktično implementira?
- Kako se proveravaju preopterećene funkcije (isto ime, različiti potpisi, engl. *overloading*)?
- Šta je dozvoljeno za predefinisane funkcije (u okviru nasleđivanja, engl. *overriding*)

## Ispravnost naredbi

- Ideja: proširiti sistem izvođenja zaključaka tako da modeluje naredbe
- Kažemo da je

$$S \vdash WF(stmt)$$

ako je naredba *stmt* dobro formirana (engl. *well formed*) u doseg *S*.

- Sistem tipova je zadovoljen ako za svaku funkciju *f* sa telom *B* koji je u doseg *S*, možemo da pokažemo da važi

$$S \vdash WF(B)$$

## A Simple Well-Formedness Rule

$$\frac{S \vdash expr : T}{S \vdash WF(expr;)}$$

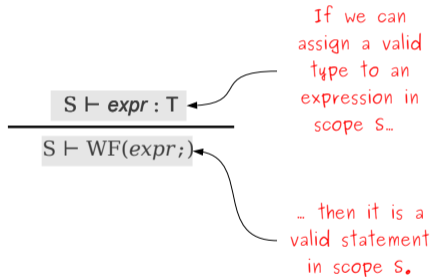


## A Simple Well-Formedness Rule

$$\frac{S \vdash expr : T}{S \vdash WF(expr;)}$$

If we can  
assign a valid  
type to an  
expression in  
scope S...

## A Simple Well-Formedness Rule



## A More Complex Rule

## A More Complex Rule

$$\frac{\begin{array}{l} S \vdash \text{WF}(\text{stmt}_1) \\ S \vdash \text{WF}(\text{stmt}_2) \end{array}}{S \vdash \text{WF}(\text{stmt}_1 \text{ stmt}_2)}$$

## Rules for **break**

## Rules for **break**

S is in a **for** or **while** loop.

$S \vdash \text{WF}(\text{break};)$

## A Rule for Loops

## A Rule for Loops

$$\frac{\begin{array}{l} S \vdash \text{expr} : \text{bool} \\ S' \text{ is the scope inside the loop.} \\ S' \vdash \text{WF}(\text{stmt}) \end{array}}{S \vdash \text{WF}(\text{while } (\text{expr}) \text{ stmt})}$$



## Rules for Block Statements

## Rules for Block Statements

$S'$  is the scope formed by adding *decls* to  $S$

$$\frac{S' \vdash \text{WF}(\textit{stmt})}{S \vdash \text{WF}(\{ \textit{decls} \textit{ stmt} \})}$$

## Rules for **return**

## Rules for `return`

S is in a function returning T

$$\frac{S \vdash \text{expr} : T' \quad T' \leq T}{S \vdash \text{WF}(\text{return expr};)}$$

$$S \vdash \text{WF}(\text{return expr};)$$

S is in a function returning `void`

$$S \vdash \text{WF}(\text{return};)$$

## Kako proveriti da li je program ispravno formiran?

- Proći rekurzivno kroz AST stablo, za svaku naredbu proveriti tip svakog podizraza koji sadrži:
  - Prijavi grešku ako ne može tip da se dodeli izrazu
  - Prijavi grešku ako je pogrešan tip dodeljen izrazu

## Something is Very Wrong Here

```
int x, y, z;  
if ((x == y) > 5 && x + y < z) || x == z) {  
    /* ... */  
}
```

## Something is Very Wrong Here

```
int x, y, z;  
if (((x == y) > 5 && x + y < z) || x == z) {  
    /* ... */  
}
```

## Something is Very Wrong Here

```
int x, y, z;  
if (((x == y) > 5 && x + y < z) || x == z) {  
    /* ... */  
}
```

Facts



## Something is Very Wrong Here

```
int x, y, z;  
if ((x == y) > 5 && x + y < z) || x == z) {  
    /* ... */  
}
```

Facts

## Something is Very Wrong Here

```
int x, y, z;  
if ((x == y) > 5 && x + y < z) || x == z) {  
    /* ... */  
}
```

### Facts

$x$  is an identifier.

$x$  is a variable in scope  $S$  with type  $T$ .

$S \vdash x : T$

## Something is Very Wrong Here

```
int x, y, z;  
if ((x == y) > 5 && x + y < z) || x == z) {  
    /* ... */  
}
```

Facts

$S \vdash x : \text{int}$

x is an identifier.  
x is a variable in scope S with type T.

$S \vdash x : T$

## Something is Very Wrong Here

```
int x, y, z;  
if ((x == y) > 5 && x + y < z) || x == z) {  
    /* ... */  
}
```

Facts

$S \vdash x : \text{int}$

x is an identifier.  
x is a variable in scope S with type T.

$S \vdash x : T$

## Something is Very Wrong Here

```
int x, y, z;  
if ((x == y) > 5 && x + y < z) || x == z) {  
    /* ... */  
}
```

$x$  is an identifier.  
 $x$  is a variable in scope  $S$  with type  $T$ .

$S \vdash x : T$

Facts
$S \vdash x : \text{int}$
$S \vdash y : \text{int}$
$S \vdash z : \text{int}$

## Something is Very Wrong Here

```
int x, y, z;  
if ((x == y) > 5 && x + y < z) || x == z) {  
    /* ... */  
}
```

$x$  is an identifier.  
 $x$  is a variable in scope  $S$  with type  $T$ .

$S \vdash x : T$

Facts
$S \vdash x : \text{int}$
$S \vdash y : \text{int}$
$S \vdash z : \text{int}$

## Something is Very Wrong Here

```
int x, y, z;  
if ((x == y) > 5 && x + y < z) || x == z) {  
    /* ... */  
}
```

$$\begin{array}{l} S \vdash e_1 : T_1 \\ S \vdash e_2 : T_2 \\ T_1 \leq T_2 \text{ or } T_2 \leq T_1 \end{array}$$

---

$$S \vdash e_1 == e_2 : \text{bool}$$

### Facts

$$S \vdash x : \text{int}$$
$$S \vdash y : \text{int}$$
$$S \vdash z : \text{int}$$

## Something is Very Wrong Here

```
int x, y, z;  
if ((x == y) > 5 && x + y < z) || x == z) {  
    /* ... */  
}
```

$$\frac{\begin{array}{l} S \vdash e_1 : T_1 \\ S \vdash e_2 : T_2 \\ T_1 \leq T_2 \text{ or } T_2 \leq T_1 \end{array}}{S \vdash e_1 == e_2 : \text{bool}}$$

Facts
$S \vdash x : \text{int}$
$S \vdash y : \text{int}$
$S \vdash z : \text{int}$
$S \vdash x == y : \text{bool}$



## Something is Very Wrong Here

```
int x, y, z;  
if ((x == y) > 5 && x + y < z) || x == z) {  
    /* ... */  
}
```

$$\frac{\begin{array}{l} S \vdash e_1 : T_1 \\ S \vdash e_2 : T_2 \\ T_1 \leq T_2 \text{ or } T_2 \leq T_1 \end{array}}{S \vdash e_1 == e_2 : \text{bool}}$$

Facts
$S \vdash x : \text{int}$
$S \vdash y : \text{int}$
$S \vdash z : \text{int}$
$S \vdash x == y : \text{bool}$

## Something is Very Wrong Here

```
int x, y, z;  
if ((x == y) > 5 && x + y < z) || x == z) {  
    /* ... */  
}
```

$i$  is an integer constant

---

$S \vdash i : \text{int}$

Facts
$S \vdash x : \text{int}$
$S \vdash y : \text{int}$
$S \vdash z : \text{int}$
$S \vdash x == y : \text{bool}$

## Something is Very Wrong Here

```
int x, y, z;  
if ((x == y) > 5 && x + y < z) || x == z) {  
    /* ... */  
}
```

$i$  is an integer constant

---

$S \vdash i : \text{int}$

### Facts

$S \vdash x : \text{int}$

$S \vdash y : \text{int}$

$S \vdash z : \text{int}$

$S \vdash x == y : \text{bool}$

$S \vdash 5 : \text{int}$

## Something is Very Wrong Here

```
int x, y, z;  
if ((x == y) > 5 && x + y < z) || x == z) {  
    /* ... */  
}
```

$i$  is an integer constant

---

$S \vdash i : \text{int}$

Facts
$S \vdash x : \text{int}$
$S \vdash y : \text{int}$
$S \vdash z : \text{int}$
$S \vdash x == y : \text{bool}$
$S \vdash 5 : \text{int}$

## Something is Very Wrong Here

```
int x, y, z;  
if ((x == y) > 5 && x + y < z) || x == z) {  
    /* ... */  
}
```

$$\frac{S \vdash e_1 : \text{int} \quad S \vdash e_2 : \text{int}}{S \vdash e_1 + e_2 : \text{int}}$$

Facts
$S \vdash x : \text{int}$
$S \vdash y : \text{int}$
$S \vdash z : \text{int}$
$S \vdash x == y : \text{bool}$
$S \vdash 5 : \text{int}$

## Something is Very Wrong Here

```
int x, y, z;  
if ((x == y) > 5 && x + y < z) || x == z) {  
    /* ... */  
}
```

$$\frac{S \vdash e_1 : \text{int} \quad S \vdash e_2 : \text{int}}{S \vdash e_1 + e_2 : \text{int}}$$

Facts
$S \vdash x : \text{int}$
$S \vdash y : \text{int}$
$S \vdash z : \text{int}$
$S \vdash x == y : \text{bool}$
$S \vdash 5 : \text{int}$
$S \vdash x + y : \text{int}$

## Something is Very Wrong Here

```
int x, y, z;  
if ((x == y) > 5 && x + y < z || x == z) {  
    /* ... */  
}
```

$$\frac{S \vdash e_1 : \text{int} \quad S \vdash e_2 : \text{int}}{S \vdash e_1 + e_2 : \text{int}}$$

Facts
$S \vdash x : \text{int}$
$S \vdash y : \text{int}$
$S \vdash z : \text{int}$
$S \vdash x == y : \text{bool}$
$S \vdash 5 : \text{int}$
$S \vdash x + y : \text{int}$

## Something is Very Wrong Here

```
int x, y, z;  
if ((x == y) > 5 && x + y < z || x == z) {  
    /* ... */  
}
```

$$\frac{S \vdash e_1 : \text{int} \quad S \vdash e_2 : \text{int}}{S \vdash e_1 < e_2 : \text{bool}}$$

Facts
$S \vdash x : \text{int}$
$S \vdash y : \text{int}$
$S \vdash z : \text{int}$
$S \vdash x == y : \text{bool}$
$S \vdash 5 : \text{int}$
$S \vdash x + y : \text{int}$



## Something is Very Wrong Here

```
int x, y, z;  
if ((x == y) > 5 && x + y < z || x == z) {  
    /* ... */  
}
```

$$\frac{S \vdash e_1 : \text{int} \quad S \vdash e_2 : \text{int}}{S \vdash e_1 < e_2 : \text{bool}}$$

Facts
$S \vdash x : \text{int}$
$S \vdash y : \text{int}$
$S \vdash z : \text{int}$
$S \vdash x == y : \text{bool}$
$S \vdash 5 : \text{int}$
$S \vdash x + y : \text{int}$
$S \vdash x + y < z : \text{bool}$

## Something is Very Wrong Here

```
int x, y, z;  
if ((x == y) > 5 && x + y < z) || x == z {  
    /* ... */  
}
```

$$\frac{S \vdash e_1 : \text{int} \quad S \vdash e_2 : \text{int}}{S \vdash e_1 < e_2 : \text{bool}}$$

Facts
$S \vdash x : \text{int}$
$S \vdash y : \text{int}$
$S \vdash z : \text{int}$
$S \vdash x == y : \text{bool}$
$S \vdash 5 : \text{int}$
$S \vdash x + y : \text{int}$
$S \vdash x + y < z : \text{bool}$

## Something is Very Wrong Here

```
int x, y, z;  
if ((x == y) > 5 && x + y < z) || x == z {  
    /* ... */  
}
```

$$\frac{\begin{array}{l} S \vdash e_1 : T_1 \\ S \vdash e_2 : T_2 \\ T_1 \leq T_2 \text{ or } T_2 \leq T_1 \end{array}}{S \vdash e_1 == e_2 : \text{bool}}$$

Facts
$S \vdash x : \text{int}$
$S \vdash y : \text{int}$
$S \vdash z : \text{int}$
$S \vdash x == y : \text{bool}$
$S \vdash 5 : \text{int}$
$S \vdash x + y : \text{int}$
$S \vdash x + y < z : \text{bool}$

## Something is Very Wrong Here

```
int x, y, z;  
if ((x == y) > 5 && x + y < z) || x == z {  
    /* ... */  
}
```

$$\frac{\begin{array}{l} S \vdash e_1 : T_1 \\ S \vdash e_2 : T_2 \\ T_1 \leq T_2 \text{ or } T_2 \leq T_1 \end{array}}{S \vdash e_1 == e_2 : \text{bool}}$$

Facts
$S \vdash x : \text{int}$
$S \vdash y : \text{int}$
$S \vdash z : \text{int}$
$S \vdash x == y : \text{bool}$
$S \vdash 5 : \text{int}$
$S \vdash x + y : \text{int}$
$S \vdash x + y < z : \text{bool}$
$S \vdash x == z : \text{bool}$

## Something is Very Wrong Here

```
int x, y, z;  
if (((x == y) > 5 && x + y < z) || x == z) {  
    /* ... */  
}
```

$$\frac{\begin{array}{l} S \vdash e_1 : T_1 \\ S \vdash e_2 : T_2 \\ T_1 \leq T_2 \text{ or } T_2 \leq T_1 \end{array}}{S \vdash e_1 == e_2 : \text{bool}}$$

Facts
$S \vdash x : \text{int}$
$S \vdash y : \text{int}$
$S \vdash z : \text{int}$
$S \vdash x == y : \text{bool}$
$S \vdash 5 : \text{int}$
$S \vdash x + y : \text{int}$
$S \vdash x + y < z : \text{bool}$
$S \vdash x == z : \text{bool}$

## Something is Very Wrong Here

```
int x, y, z;  
if (((x == y) > 5 && x + y < z) || x == z) {  
    /* ... */  
}
```

$$\frac{S \vdash e_1 : \text{int} \quad S \vdash e_2 : \text{int}}{S \vdash e_1 > e_2 : \text{bool}}$$

Facts
$S \vdash x : \text{int}$
$S \vdash y : \text{int}$
$S \vdash z : \text{int}$
$S \vdash x == y : \text{bool}$
$S \vdash 5 : \text{int}$
$S \vdash x + y : \text{int}$
$S \vdash x + y < z : \text{bool}$
$S \vdash x == z : \text{bool}$

## Something is Very Wrong Here

```
int x, y, z;  
if (((x == y) > 5 && x + y < z) || x == z) {  
    /* ... */  
}
```

$$\frac{S \vdash e_1 : \text{int} \quad S \vdash e_2 : \text{int}}{S \vdash e_1 > e_2 : \text{bool}}$$



Facts
$S \vdash x : \text{int}$
$S \vdash y : \text{int}$
$S \vdash z : \text{int}$
$S \vdash x == y : \text{bool}$
$S \vdash 5 : \text{int}$
$S \vdash x + y : \text{int}$
$S \vdash x + y < z : \text{bool}$
$S \vdash x == z : \text{bool}$

## Something is Very Wrong Here

```
int x, y, z;  
if (((x == y) > 5 && x + y < z) || x == z) {  
    /* ... */  
}
```

$$\frac{S \vdash e_1 : \text{int} \quad S \vdash e_2 : \text{int}}{S \vdash e_1 > e_2 : \text{bool}}$$

```
> Error: Cannot compare int and bool
```

Facts
$S \vdash x : \text{int}$
$S \vdash y : \text{int}$
$S \vdash z : \text{int}$
$S \vdash x == y : \text{bool}$
$S \vdash 5 : \text{int}$
$S \vdash x + y : \text{int}$
$S \vdash x + y < z : \text{bool}$
$S \vdash x == z : \text{bool}$



## Something is Very Wrong Here

```
int x, y, z;  
if (((x == y) > 5 && x + y < z) || x == z) {  
    /* ... */  
}
```

$$\frac{S \vdash e_1 : \text{int} \quad S \vdash e_2 : \text{int}}{S \vdash e_1 > e_2 : \text{bool}}$$

```
> Error: Cannot compare int and bool
```

Facts
$S \vdash x : \text{int}$
$S \vdash y : \text{int}$
$S \vdash z : \text{int}$
$S \vdash x == y : \text{bool}$
$S \vdash 5 : \text{int}$
$S \vdash x + y : \text{int}$
$S \vdash x + y < z : \text{bool}$
$S \vdash x == z : \text{bool}$

## Something is Very Wrong Here

```
int x, y, z;  
if (((x == y) > 5 && x + y < z) || x == z) {  
    /* ... */  
}
```

$$\frac{S \vdash e_1 : \text{bool} \quad S \vdash e_2 : \text{bool}}{S \vdash e_1 \ \&\& \ e_2 : \text{bool}}$$

```
> Error: Cannot compare int and bool
```

Facts
$S \vdash x : \text{int}$
$S \vdash y : \text{int}$
$S \vdash z : \text{int}$
$S \vdash x == y : \text{bool}$
$S \vdash 5 : \text{int}$
$S \vdash x + y : \text{int}$
$S \vdash x + y < z : \text{bool}$
$S \vdash x == z : \text{bool}$

## Something is Very Wrong Here

```
int x, y, z;  
if (((x == y) > 5 && x + y < z) || x == z) {  
    /* ... */  
}
```

$$\frac{S \vdash e_1 : \text{bool} \quad S \vdash e_2 : \text{bool}}{S \vdash e_1 \ \&\& \ e_2 : \text{bool}}$$

```
> Error: Cannot compare int and bool  
Error: Cannot compare ??? and bool
```

Facts
$S \vdash x : \text{int}$
$S \vdash y : \text{int}$
$S \vdash z : \text{int}$
$S \vdash x == y : \text{bool}$
$S \vdash 5 : \text{int}$
$S \vdash x + y : \text{int}$
$S \vdash x + y < z : \text{bool}$
$S \vdash x == z : \text{bool}$

## Something is Very Wrong Here

```
int x, y, z;  
if ((x == y) > 5 && x + y < z) || x == z) {  
    /* ... */  
}
```

$$\frac{S \vdash e_1 : \text{bool} \quad S \vdash e_2 : \text{bool}}{S \vdash e_1 \ \&\& \ e_2 : \text{bool}}$$

```
> Error: Cannot compare int and bool  
Error: Cannot compare ??? and bool
```

Facts
$S \vdash x : \text{int}$
$S \vdash y : \text{int}$
$S \vdash z : \text{int}$
$S \vdash x == y : \text{bool}$
$S \vdash 5 : \text{int}$
$S \vdash x + y : \text{int}$
$S \vdash x + y < z : \text{bool}$
$S \vdash x == z : \text{bool}$

## Something is Very Wrong Here

```
int x, y, z;  
if ((x == y) > 5 && x + y < z) || x == z) {  
    /* ... */  
}
```

$$\frac{S \vdash e_1 : \text{bool} \quad S \vdash e_2 : \text{bool}}{S \vdash e_1 \parallel e_2 : \text{bool}}$$

```
> Error: Cannot compare int and bool  
Error: Cannot compare ??? and bool
```

Facts
$S \vdash x : \text{int}$
$S \vdash y : \text{int}$
$S \vdash z : \text{int}$
$S \vdash x == y : \text{bool}$
$S \vdash 5 : \text{int}$
$S \vdash x + y : \text{int}$
$S \vdash x + y < z : \text{bool}$
$S \vdash x == z : \text{bool}$

## Something is Very Wrong Here

```
int x, y, z;  
if ((x == y) > 5 && x + y < z) || x == z) {  
    /* ... */  
}
```

$$\frac{S \vdash e_1 : \text{bool} \quad S \vdash e_2 : \text{bool}}{S \vdash e_1 \parallel e_2 : \text{bool}}$$

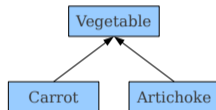
```
> Error: Cannot compare int and bool  
Error: Cannot compare ??? and bool  
Error: Cannot compare ??? and bool
```

Facts
$S \vdash x : \text{int}$
$S \vdash y : \text{int}$
$S \vdash z : \text{int}$
$S \vdash x == y : \text{bool}$
$S \vdash 5 : \text{int}$
$S \vdash x + y : \text{int}$
$S \vdash x + y < z : \text{bool}$
$S \vdash x == z : \text{bool}$

## Propagiranje greške

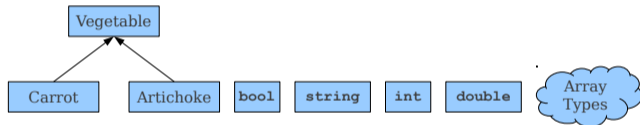
- Prethodni primer je demonstrirao propagaciju greške
- Statička greška tipova se dešava kada ne možemo da dokažemo da izraz ima odgovarajući tip
- Greške u tipovima se lako propagiraju: na primer, ako ne možemo da dokažemo tip od  $e_1$  onda ne možemo da dokažemo ni tip za  $e_1 + e_2$ , ni tip za  $(e_1 + e_2) + e_3$  i tako redom.
- Kako to prevazići?

## The Shape of Types

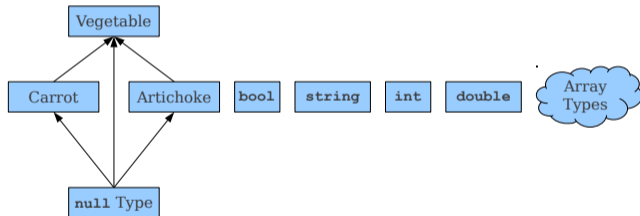




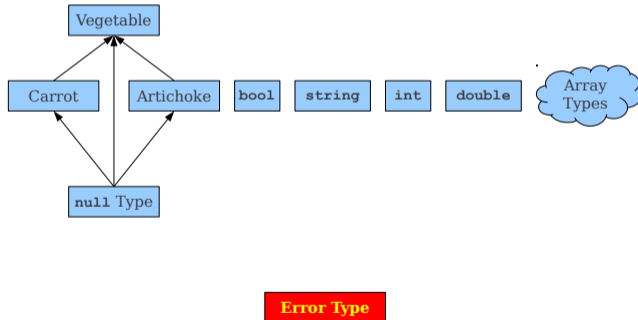
## The Shape of Types



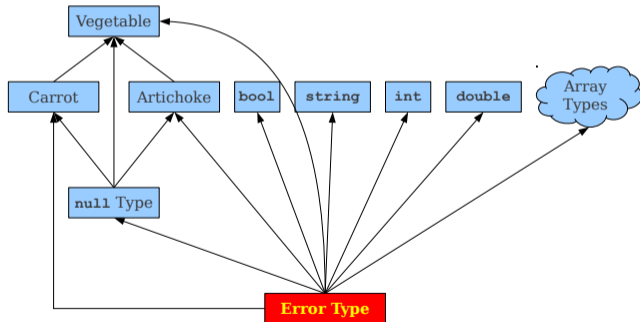
## The Shape of Types



## The Shape of Types



## The Shape of Types



## Tip Error

- U sistem tipova uvodimo novi tip koji predstavlja grešku
- Ovaj tip je manji od svih drugih tipova i označava se sa  $\perp$
- Nekada se zove i *bottom* tip
- Po definiciji, važi  $\perp \leq A$ , za svako  $A$
- Kada otkriješ tip Error, pretvaraj se kao da je dokazan izraz tipa  $\perp$
- Potrebno je samo unaprediti pravila zaključivanja tako da sadrže  $\perp$

## Updated Rules for Addition

$$S \vdash e_1 : \text{double}$$
$$S \vdash e_2 : \text{double}$$

---

$$S \vdash e_1 + e_2 : \text{double}$$

## Updated Rules for Addition

$$\begin{array}{l} S \vdash e_1 : T_1 \\ S \vdash e_2 : T_2 \end{array}$$

---

$$S \vdash e_1 + e_2 : \mathbf{double}$$

## Updated Rules for Addition

$$\frac{\begin{array}{l} S \vdash e_1 : T_1 \\ S \vdash e_2 : T_2 \\ T_1 \leq \text{double} \\ T_2 \leq \text{double} \end{array}}{S \vdash e_1 + e_2 : \text{double}}$$



## Updated Rules for Addition

$$\frac{\begin{array}{l} S \vdash e_1 : T_1 \\ S \vdash e_2 : T_2 \\ T_1 \leq \text{double} \\ T_2 \leq \text{double} \end{array}}{S \vdash e_1 + e_2 : \text{double}}$$

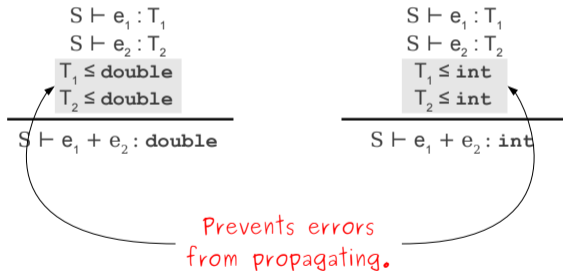
what does  
this mean?

## Updated Rules for Addition

$$\frac{\begin{array}{l} S \vdash e_1 : T_1 \\ S \vdash e_2 : T_2 \\ T_1 \leq \mathbf{double} \\ T_2 \leq \mathbf{double} \end{array}}{S \vdash e_1 + e_2 : \mathbf{double}}$$

$$\frac{\begin{array}{l} S \vdash e_1 : T_1 \\ S \vdash e_2 : T_2 \\ T_1 \leq \mathbf{int} \\ T_2 \leq \mathbf{int} \end{array}}{S \vdash e_1 + e_2 : \mathbf{int}}$$

## Updated Rules for Addition



## Updated Rules for Addition

$$\frac{\begin{array}{l} S \vdash e_1 : T_1 \\ S \vdash e_2 : T_2 \\ T_1 \leq \mathbf{double} \\ T_2 \leq \mathbf{double} \end{array}}{S \vdash e_1 + e_2 : \mathbf{double}}$$

$$\frac{\begin{array}{l} S \vdash e_1 : T_1 \\ S \vdash e_2 : T_2 \\ T_1 \leq \mathbf{int} \\ T_2 \leq \mathbf{int} \end{array}}{S \vdash e_1 + e_2 : \mathbf{int}}$$

## Updated Rules for Addition

$$\frac{\begin{array}{l} S \vdash e_1 : T_1 \\ S \vdash e_2 : T_2 \\ T_1 \leq \mathbf{double} \\ T_2 \leq \mathbf{double} \end{array}}{S \vdash e_1 + e_2 : \mathbf{double}}$$

$$\frac{\begin{array}{l} S \vdash e_1 : T_1 \\ S \vdash e_2 : T_2 \\ T_1 \leq \mathbf{int} \\ T_2 \leq \mathbf{int} \end{array}}{S \vdash e_1 + e_2 : \mathbf{int}}$$

What happens if  
both operands have  
error type?

## Oporavljanje od grešaka

- U semantičkom analizatoru neophodno je da postoji neka vrsta oporavka od greške, kako bi mogla da se nastavi dalje analiza
- Jedan vid oporavljanja od grešaka je tip `Error` koji smo uveli, ali postoje i drugi slučajevi koje treba rešiti, na primer:
  - Poziv nepostojeće funkcije
  - Deklaracija promenljive nekog neispravnog tipa
  - Tretiranje promenljive koja nije niz kao da jeste niz
  - ...
- Ne postoje ispravni i pogrešni odgovori na ova pitanja, već samo bolji i lošiji izbori

## Preopterećivanje funkcija (engl. *overloading*)

- Preopterećivanje funkcija: više funkcija sa istim imenom ali različitim argumentima  
`int saberi(int a, int b)` i `int saberi(int a, int b, int c)`
- Preopterećivanje funkcija povećava čitljivost koda
- Kod preopterećivanja imamo pozive funkcija koje se razlikuju po broju argumenata ili po tipovima argumenata ili po oba
- Nije dozvoljena razlika samo u povratnom tipu funkcije, na primer, ovako nešto nije dozvoljeno:  
`int saberi(int a, int b)` i `long saberi(int a, int b)`
- Naziva se i statički polimorfizam (ili *compile time* polimorfizam)

## Preopterećivanje funkcija (engl. *overloading*)

- U fazi kompilacije, analizom tipova argumenata, potrebno je odrediti koju funkciju treba pozvati
- U slučaju da ne može da se odredi funkcija koja se najbolje uklapa, treba prijaviti grešku



## Overloading Example

```
void Function();  
void Function(int x);  
void Function(double x);  
void Function(Base b);  
void Function(Derived d);
```

```
Function();  
Function(137);  
Function(42.0);  
Function(new Base);  
Function(new Derived);
```

## Overloading Example

```
void Function();  
void Function(int x);  
void Function(double x);  
void Function(Base b);  
void Function(Derived d);
```

```
Function();  
Function(137);  
Function(42.0);  
Function(new Base);  
Function(new Derived);
```

## Implementing Overloading

```
void Function();  
void Function(int x);  
void Function(double x);  
void Function(Base b);  
void Function(Derived d);
```

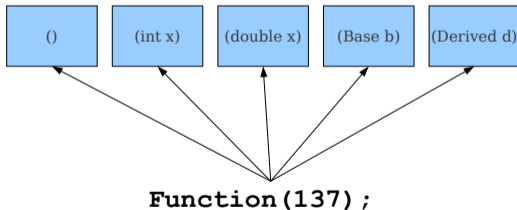
## Implementing Overloading

```
void Function();  
void Function(int x);  
void Function(double x);  
void Function(Base b);  
void Function(Derived d);
```

**Function(137);**

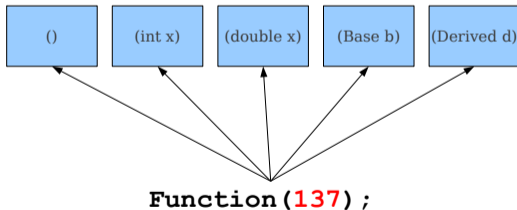
## Implementing Overloading

```
void Function();  
void Function(int x);  
void Function(double x);  
void Function(Base b);  
void Function(Derived d);
```



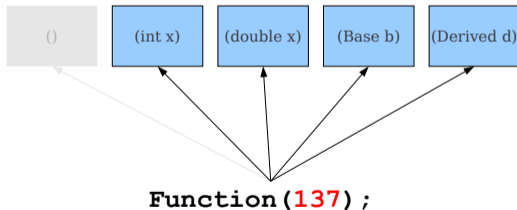
## Implementing Overloading

```
void Function();  
void Function(int x);  
void Function(double x);  
void Function(Base b);  
void Function(Derived d);
```



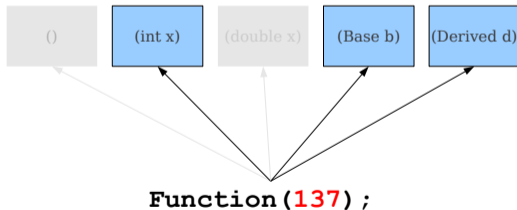
## Implementing Overloading

```
void Function();  
void Function(int x);  
void Function(double x);  
void Function(Base b);  
void Function(Derived d);
```



## Implementing Overloading

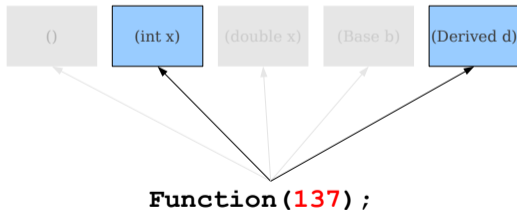
```
void Function();  
void Function(int x);  
void Function(double x);  
void Function(Base b);  
void Function(Derived d);
```





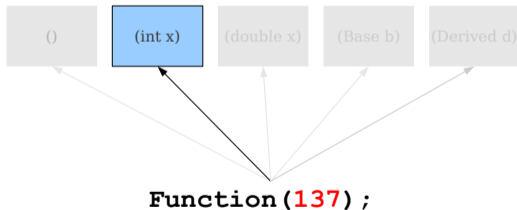
## Implementing Overloading

```
void Function();  
void Function(int x);  
void Function(double x);  
void Function(Base b);  
void Function(Derived d);
```



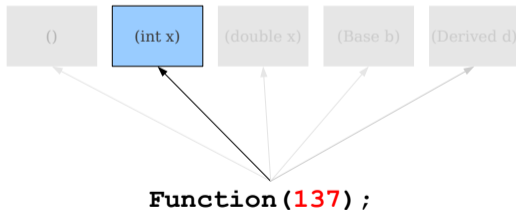
## Implementing Overloading

```
void Function();  
void Function(int x);  
void Function(double x);  
void Function(Base b);  
void Function(Derived d);
```



## Implementing Overloading

```
void Function();  
void Function(int x);  
void Function(double x);  
void Function(Base b);  
void Function(Derived d);
```



## Jednostavno preopterećivanje

- Počinjemo od skupa preopterećenih funkcija
- Najpre isfiltriramo funkcije koje se ne poklapaju i tako dobijamo skup kandidata (C++ terminologija) ili skup potencijalno primenljivih metoda (Java terminologija)
- Ako je skup prazan, prijavi grešku
- Ako skup sadrži samo jednu funkciju, izaberi nju
- Ako skup sadrži više funkcija... (izaberi *najbolju*)

## Overloading with Inheritance

```
void Function();  
void Function(int x);  
void Function(double x);  
void Function(Base b);  
void Function(Derived d);
```

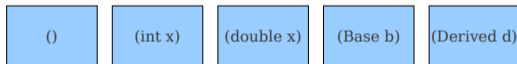
## Overloading with Inheritance

```
void Function();  
void Function(int x);  
void Function(double x);  
void Function(Base b);  
void Function(Derived d);
```

```
Function(new Derived);
```

## Overloading with Inheritance

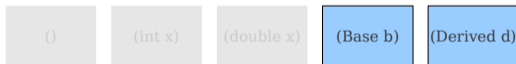
```
void Function();  
void Function(int x);  
void Function(double x);  
void Function(Base b);  
void Function(Derived d);
```



**Function(new Derived);**

## Overloading with Inheritance

```
void Function();  
void Function(int x);  
void Function(double x);  
void Function(Base b);  
void Function(Derived d);
```



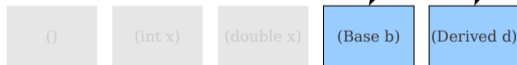
```
Function(new Derived);
```



## Overloading with Inheritance

```
void Function();  
void Function(int x);  
void Function(double x);  
void Function(Base b);  
void Function(Derived d);
```

How do we  
compare  
these?



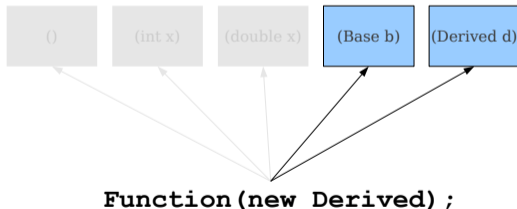
```
Function(new Derived);
```

## Kako odrediti najbolje poklapanje

- Ako postoji više od jednog izbora, izaberi funkciju koja najspecifičnije odgovara datom pozivu
- Ako imamo dve funkcije koje su kandidati,  $A$  i  $B$ , sa argumentima  $A_1, A_2, \dots, A_n$  i  $B_1, B_2, \dots, B_n$ , kažemo da je  $A <: B$  ako važi  $A_i \leq B_i$  za svako  $i \in \{1, 2, \dots, n\}$
- Relacija  $<:$  je parcijalno uređenje
- **Funkcija  $A$  je najbolji izbor ako za svaku drugu funkciju  $B$  koja je kandidat za izbor važi  $A <: B$  (tj, ona je bar onoliko dobra koliko i svaki drugi izbor)**
- Ako postoji najbolji izbor (ili najbolje poklapanje), onda se bira ta funkcija
- U suprotnom, poziv je višesmislen i to mora nekako da se razreši

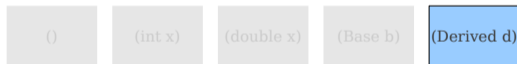
## Overloading with Inheritance

```
void Function();  
void Function(int x);  
void Function(double x);  
void Function(Base b);  
void Function(Derived d);
```



## Overloading with Inheritance

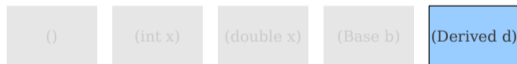
```
void Function();  
void Function(int x);  
void Function(double x);  
void Function(Base b);  
void Function(Derived d);
```



```
Function(new Derived);
```

## Overloading with Inheritance

```
void Function();  
void Function(int x);  
void Function(double x);  
void Function(Base b);  
void Function(Derived d);
```



```
Function(new Derived);
```

## Ambiguous Calls

```
void Function(Base b1, Base b2);  
void Function(Derived d1, Base b2);  
void Function(Base b1, Derived d2);
```

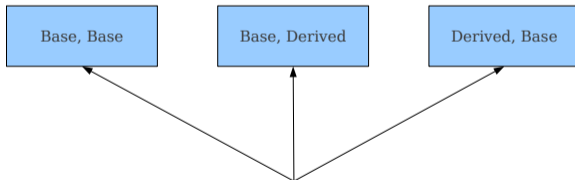
## Ambiguous Calls

```
void Function(Base b1, Base b2);  
void Function(Derived d1, Base b2);  
void Function(Base b1, Derived d2);
```

```
Function(new Derived, new Derived);
```

## Ambiguous Calls

```
void Function(Base b1, Base b2);  
void Function(Derived d1, Base b2);  
void Function(Base b1, Derived d2);
```

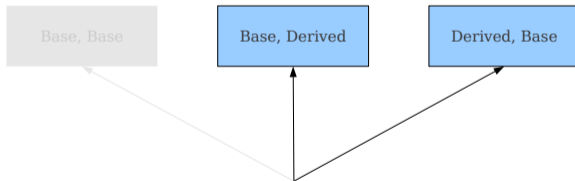


```
Function(new Derived, new Derived);
```



## Ambiguous Calls

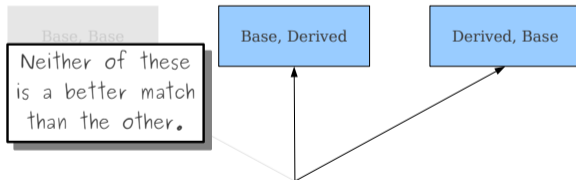
```
void Function(Base b1, Base b2);  
void Function(Derived d1, Base b2);  
void Function(Base b1, Derived d2);
```



```
Function(new Derived, new Derived);
```

## Ambiguous Calls

```
void Function(Base b1, Base b2);  
void Function(Derived d1, Base b2);  
void Function(Base b1, Derived d2);
```



```
Function(new Derived, new Derived);
```

## Preopterećivanje - u realnosti

- Preopterećivanje je obično značajno kompleksnije
- Na primer, neki jezici (npr C, C++, Java) dozvoljavaju funkcije sa promenljivim brojem argumenata (engl. *variadic functions*)

## Overloading with Variadic Functions

```
void Function(Base b1, Base b2);  
void Function(Derived d1, Base b2);  
void Function(Derived d1, ...);
```

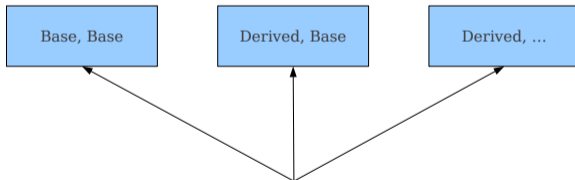
## Overloading with Variadic Functions

```
void Function(Base b1, Base b2);  
void Function(Derived d1, Base b2);  
void Function(Derived d1, ...);
```

```
Function(new Derived, new Derived);
```

## Overloading with Variadic Functions

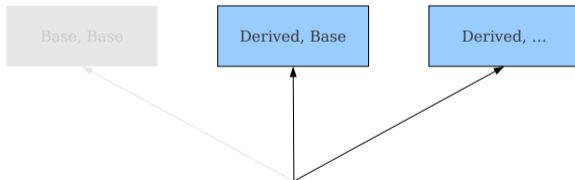
```
void Function(Base b1, Base b2);  
void Function(Derived d1, Base b2);  
void Function(Derived d1, ...);
```



```
Function(new Derived, new Derived);
```

## Overloading with Variadic Functions

```
void Function(Base b1, Base b2);  
void Function(Derived d1, Base b2);  
void Function(Derived d1, ...);
```



```
Function(new Derived, new Derived);
```

## Preopterećivanje u kontekstu variadic funkcija

- Prva opcija: smatraj poziv višesmislenim
- Druga opcija: smatraj boljom funkciju koja nije variadic: funkcija koja je specifično dizajnirana da podrži neki konkretan skup argumenata je verovatno bolji izbor od one koja je dizajnirana da može da podrži proizvoljno mnogo parametara
- Druga opcija se koristi u C++-u i (sa malim modifikacijama) u Javi

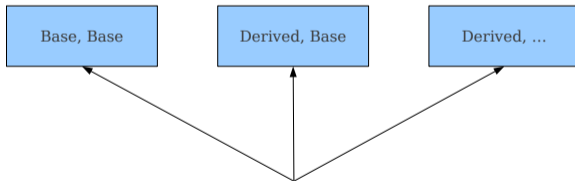


## Hijerarhijsko preopterećivanje funkcija

- Ideja: Napraviti hijerarhiju funkcija kandidata
- Konceptualno to je vrlo slično dosezima
  - Počni sa najnižim nivoom hijerarhije, i traži u okviru njega poklapanje
  - Ako pronađeš jedinstveno poklapanje, izaberi ga
  - Ako pronađeš višestruka poklapanja, prijavi višesmislenost
  - Ako ne pronađeš poklapanje, idi na naredni nivo u hijerarhiji
- Slične tehnike se koriste i u drugim situacijama, npr
  - Šablonske/generičke funkcije
  - Implicitne konverzije

## Overloading with Variadic Functions

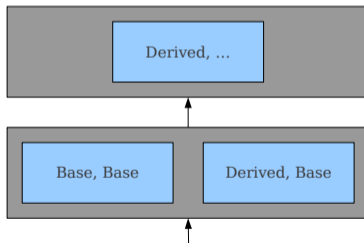
```
void Function(Base b1, Base b2);  
void Function(Derived d1, Base b2);  
void Function(Derived d1, ...);
```



```
Function(new Derived, new Derived);
```

## Overloading with Variadic Functions

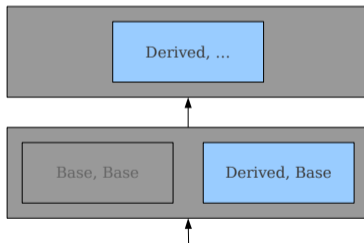
```
void Function(Base b1, Base b2);  
void Function(Derived d1, Base b2);  
void Function(Derived d1, ...);
```



```
Function(new Derived, new Derived);
```

## Overloading with Variadic Functions

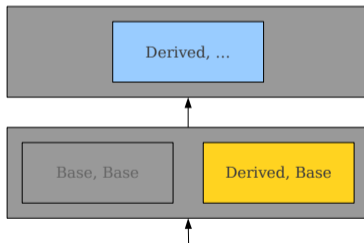
```
void Function(Base b1, Base b2);  
void Function(Derived d1, Base b2);  
void Function(Derived d1, ...);
```



```
Function(new Derived, new Derived);
```

## Overloading with Variadic Functions

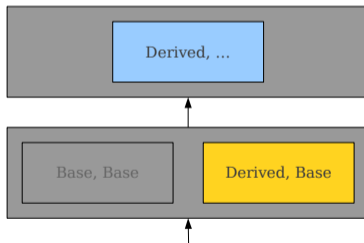
```
void Function(Base b1, Base b2);  
void Function(Derived d1, Base b2);  
void Function(Derived d1, ...);
```



```
Function(new Derived, new Derived);
```

## Overloading with Variadic Functions

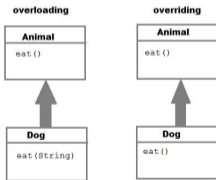
```
void Function(Base b1, Base b2);  
void Function(Derived d1, Base b2);  
void Function(Derived d1, ...);
```



```
Function(new Derived, new Derived);
```

## Predefinisanje (engl. *overriding*) funkcija

Preopterećivanje može da bude u okviru iste klase ili između klasa koje su odnosu nasleđivanja. Predefinisanje se odnosi samo na funkcije koje su u okviru klasa koje su u odnosu nasleđivanja.



Najjednostavnija odluka: kod predefinisanja, svi tipovi moraju da se u potpunosti poklapaju. Međutim, to uvodi previše ograničenja i potrebno je biti fleksibilniji.

## Različite varijante funkcija

- Kako izbor odluka vezanih za predefinisane (engl. *overriding*) utiče na sistem tipova i na karakteristike programskih jezika?
- Primer: Neka klasa *Mačka* nasleđuje klasu *Sisar* i neka klasa *Sisar* nasleđuje klasu *Životinja*. Neka je funkcija  $f$  definisana u klasi *Životinja* i predefinisana u klasama *Sisar* i *Mačka*.
- Kovarijante i kontravarijante



## Različite varijante funkcija

**Kovarijanta po povratnom tipu** Da li u okviru klasa možemo da dozvolimo da predefinisana funkcija ima kao povratni tip svoj klasni tip umesto nadklasnog tipa?

**Primer** Da li za predefinisanu funkciju  $f$  povratni tip može da bude *Mačka* umesto *Sisar*?

Na primer, ako je  $f$  u klasi *Sisar* definisana sa

```
Sisar f()
```

da li u klasi *Mačka* funkcija  $f$  može da bude predefinisana sa

```
Mačka f()?
```

## Različite varijante funkcija

**Kovarijanta po argumentu** Da li u okviru klasa možemo da dozvolimo da predefinisana funkcija ima kao argument svoj klasni tip umesto nadklasnog tipa?

**Primer** Da li za predefinisanu funkciju  $f$  argument funkcije može da bude *Mačka* umesto *Sisar*?

Ako je  $f$  u klasi *Sisar* definisana sa

```
void f(Sisar s)
```

da li u klasi *Mačka* funkcija  $f$  može da bude predefinisana sa

```
void f(Mačka m)?
```

## Različite varijante funkcija

**Kontravarijanta po argumentu** Da li u okviru klase možemo da dozvolimo da predefinisana funkcija ima kao argument tip superklase umesto nadklasnog tipa?

**Primer** Da li za predefinisanu funkciju  $f$  argument funkcije može da bude *Životinja* umesto *Sisar*?

Ako je  $f$  u klasi *Sisar* definisana sa

```
void f(Sisar s)
```

da li u klasi *Mačka* funkcija  $f$  može da bude predefinisana sa

```
void f(Životinja m)?
```

## A Rule for Member Functions

$$\frac{}{S \vdash e_0.f(e_1, \dots, e_n) : ?}$$

## A Rule for Member Functions

$f$  is an identifier.

---

$$S \vdash e_0.f(e_1, \dots, e_n) : ?$$

## A Rule for Member Functions

$f$  is an identifier.

$S \vdash e_0 : M$

---

$S \vdash e_0.f(e_1, \dots, e_n) : ?$

## A Rule for Member Functions

$f$  is an identifier.

$S \vdash e_0 : M$

$f$  is a member function in class  $M$ .

---

$S \vdash e_0.f(e_1, \dots, e_n) : ?$

## A Rule for Member Functions

$f$  is an identifier.

$S \vdash e_0 : M$

$f$  is a member function in class  $M$ .

$f$  has type  $(T_1, \dots, T_n) \rightarrow U$

---

$S \vdash e_0.f(e_1, \dots, e_n) : ?$



## A Rule for Member Functions

$f$  is an identifier.

$S \vdash e_0 : M$

$f$  is a member function in class  $M$ .

$f$  has type  $(T_1, \dots, T_n) \rightarrow U$

$S \vdash e_i : R_i$  for  $1 \leq i \leq n$

$R_i \leq T_i$  for  $1 \leq i \leq n$

---

$S \vdash e_0.f(e_1, \dots, e_n) : ?$

## A Rule for Member Functions

$$\begin{array}{l} f \text{ is an identifier.} \\ S \vdash e_0 : M \\ f \text{ is a member function in class } M. \\ f \text{ has type } (T_1, \dots, T_n) \rightarrow U \\ S \vdash e_i : R_i \text{ for } 1 \leq i \leq n \\ R_i \leq T_i \text{ for } 1 \leq i \leq n \\ \hline S \vdash e_0.f(e_1, \dots, e_n) : U \end{array}$$

## Izazovi sistema tipova

- Da li je prethodno definisano pravilo ispravno?
  - Da li sa tako definisanim pravilom imamo garanciju da nam neće promaći ni jedna greška u tipovima u fazu izvršavanja, tj, da li je pravilo bezbedno (engl. *safe*)?
  - Da li se sa tako definisanim pravilom može desiti da neki program, iako validan u fazi izvršavanja može da bude odbijen u fazi prevođenja, tj, da li je pravilo legalno (engl. *legal*)?

## Legality and Safety

```
class Id {
    Id me() {
        return this;
    }
    void beSelfish() {
        /* ... */
    }
}

class Ego extends Id {
    void bePractical() {
        /* ... */
    }
}

int main() {
    (new Ego).me().bePractical();
}
```

## Legality and Safety

```
class Id {
    Id me() {
        return this;
    }
    void beSelfish() {
        /* ... */
    }
}

class Ego extends Id {
    void bePractical() {
        /* ... */
    }
}

int main() {
    (new Ego).me().bePractical();
}
```

## Legality and Safety

```
class Id {
    Id me() {
        return this;
    }
    void beSelfish() {
        /* ... */
    }
}

class Ego extends Id {
    void bePractical() {
        /* ... */
    }
}

int main() {
    (new Ego).me().bePractical();
}
    {
        Ego
    }
```

## Legality and Safety

```
class Id {
    Id me() {
        return this;
    }
    void beSelfish() {
        /* ... */
    }
}

class Ego extends Id {
    void bePractical() {
        /* ... */
    }
}

int main() {
    (new Ego) .me () .bePractical ();
}
    {
        Ego
```

## Legality and Safety

```
class Id {
  Id me() {
    return this;
  }
  void beSelfish() {
    /* ... */
  }
}

class Ego extends Id {
  void bePractical() {
    /* ... */
  }
}

int main() {
  (new Ego).me().bePractical();
}
```

Ego

$f$  is an identifier.  
 $S \vdash e_0 : M$   
 $f$  is a member function in class  $M$ .  
 $f$  has type  $(T_1, \dots, T_n) \rightarrow U$   
 $S \vdash e_i : R_i$  for  $1 \leq i \leq n$   
 $R_i \leq T_i$  for  $1 \leq i \leq n$   

---

 $S \vdash e_0.f(e_1, \dots, e_n) : U$



## Legality and Safety

```
class Id {  
  Id me() {  
    return this;  
  }  
  void beSelfish() {  
    /* ... */  
  }  
}  
  
class Ego extends Id {  
  void bePractical() {  
    /* ... */  
  }  
}  
  
int main() {  
  (new Ego).me().bePractical();  
}
```

Ego

$f$  is an identifier.

$S \vdash e_0 : M$

$f$  is a member function in class  $M$ .

$f$  has type  $(T_1, \dots, T_n) \rightarrow U$

$S \vdash e_i : R_i$  for  $1 \leq i \leq n$

$R_i \leq T_i$  for  $1 \leq i \leq n$

---

$S \vdash e_0.f(e_1, \dots, e_n) : U$

## Legality and Safety

```
class Id {  
    Id me() {  
        return this;  
    }  
    void beSelfish() {  
        /* ... */  
    }  
}  
  
class Ego extends Id {  
    void bePractical() {  
        /* ... */  
    }  
}  
  
int main() {  
    (new Ego).me().bePractical();  
}
```

                          
Ego                Id

$f$  is an identifier.  
 $S \vdash e_0 : M$   
 $f$  is a member function in class  $M$ .  
 $f$  has type  $(T_1, \dots, T_n) \rightarrow U$   
 $S \vdash e_i : R_i$  for  $1 \leq i \leq n$   
 $R_i \leq T_i$  for  $1 \leq i \leq n$   

---

 $S \vdash e_0.f(e_1, \dots, e_n) : U$

## Legality and Safety

```
class Id {
  Id me() {
    return this;
  }
  void beSelfish() {
    /* ... */
  }
}

class Ego extends Id {
  void bePractical() {
    /* ... */
  }
}

int main() {
  (new Ego).me().bePractical();
}
```

Ego      Id

$f$  is an identifier.  
 $S \vdash e_0 : M$   
 $f$  is a member function in class  $M$ .  
 $f$  has type  $(T_1, \dots, T_n) \rightarrow U$   
 $S \vdash e_i : R_i$  for  $1 \leq i \leq n$   
 $R_i \leq T_i$  for  $1 \leq i \leq n$   

---

 $S \vdash e_0.f(e_1, \dots, e_n) : U$

## Legality and Safety

```
class Id {  
  Id me() {  
    return this;  
  }  
  void beSelfish() {  
    /* ... */  
  }  
}  
  
class Ego extends Id {  
  void bePractical() {  
    /* ... */  
  }  
}  
  
int main() {  
  (new Ego).me().bePractical();  
}
```

**Ego**      **Id**

$f$  is an identifier.

$S \vdash e_0 : M$

$f$  is a member function in class  $M$ .

$f$  has type  $(T_1, \dots, T_n) \rightarrow U$

$S \vdash e_i : R_i$  for  $1 \leq i \leq n$

$R_i \leq T_i$  for  $1 \leq i \leq n$

$S \vdash e_0.f(e_1, \dots, e_n) : U$

**bePractical**  
is not in  
**Id!**

## Statički i dinamički tipovi

- Na osnovu definisanog pravila, prethodni program bi, iako se može ispravno izvršiti, bio odbijen: statički sistemi tipova su često nepotpuni (nekompletni, engl. *incomplete*)
- To se dešava zato što postoji razlika između statičkih i dinamičkih tipova:
  - Statički tip je deklarisan u izvornom kodu
  - Dinamički tip je pravi tip objekta u fazi izvršavanja

## Kompletnost (potpunost) i saglasnost

- Idealan sistem tipova: kompletan (potpun) i saglasan
  - **Kompletnost (potpunost):**  
ako se program može ispravno izvršiti — sistem tipova kaže da je on ispravan u smislu tipova
  - **Saglasnost:**  
ako sistem tipova kaže da je program ispravan — onda važi da se program može ispravno izvršiti
- Nažalost, za većinu programskih jezika, **nemoguće je ostvariti i potpunost i saglasnost** (rešavanje problema provere tipova svodi se na rešavanje halting problema, koji je neodlučiv)

## Dobar sistem tipova

- Teško je napraviti dobar sistem tipova, tj, lako je napraviti pravila koja su nesaglasna, a često je nemoguće prihvatiti sve ispravne programe.
- Da bi se izgradio dobar statički sistem za proveru tipova obično je **cilj da se napravi jezik koji je što "kompletniji"** (tj da ima što manje situacija u kojima ne važi uslov kompletnosti) a da pritom ima saglasan sistem pravila za proveru tipova.
- Saglasnost se može obezbediti dokazivanjem narednog svojstva za svaki izraz  $E$ :

$$DynamicType(E) \leq StaticType(E)$$

- Kao što je već pomenuto, statički sistemi tipova mogu nekada da odbiju program koji bi se mogao ispravno izvršiti (zato što nisu u mogućnosti da dokažu odsustvo greške u tipovima). Takav sistem tipova naziva se nekompletan (nepotpun).

## Različite varijante funkcija

**Kovarijanta po povratnom tipu** Da li u okviru klase možemo da dozvolimo da predefinisana funkcija ima kao povratni tip svoj klasni tip umesto nadklasnog tipa?

**Primer** Da li za predefinisanu funkciju  $f$  povratni tip može da bude *Mačka* umesto *Sisar*?

Na primer, ako je  $f$  u klasi *Sisar* definisana sa

```
Sisar f()
```

da li u klasi *Mačka* funkcija  $f$  može da bude predefinisana sa

```
Mačka f()?
```



## Kovarijanta po povratnom tipu

- Neka funkcija A predefiniše funkciju B, ali neka se ove funkcije razlikuju po povratnom tipu: funkcije A i B su kovarijante po povratnom tipu ukoliko se tipovi argumenata poklapaju, a povratni tip funkcije A se može pretvoriti (konvertovati) u povratni tip funkcije B.
- Da li je bezbedno dozvoliti kovarijante po povratnom tipu?

## Relaxing our Restrictions

```
class Base {
    Base clone() {
        return new Base;
    }
}

class Derived extends Base {
    Base clone() {
        return new Derived;
    }
}
```

## Relaxing our Restrictions

```
class Base {  
    Base clone() {  
        return new Base;  
    }  
}  
  
class Derived extends Base {  
    Base clone() {  
        return new Derived;  
    }  
}
```

## Relaxing our Restrictions

```
class Base {  
    Base clone() {  
        return new Base;  
    }  
}  
  
class Derived extends Base {  
    Derived clone() {  
        return new Derived;  
    }  
}
```

## Relaxing our Restrictions

```
class Base {  
    Base clone() {  
        return new Base;  
    }  
}  
  
class Derived extends Base {  
    Derived clone() {  
        return new Derived;  
    }  
}
```

Is this safe?

## The Intuition

```
Base b = new Base;  
Derived d = new Derived;
```

## The Intuition

```
Base b = new Base;  
Derived d = new Derived;  
  
Base b2 = b.clone();
```

## The Intuition

```
Base b = new Base;  
Derived d = new Derived;  
  
Base b2 = b.clone();  
Base b3 = d.clone();
```



## The Intuition

```
Base b = new Base;  
Derived d = new Derived;  
  
Base b2 = b.clone();  
Base b3 = d.clone();  
Derived d2 = b.clone();
```

## The Intuition

```
Base b = new Base;  
Derived d = new Derived;  
  
Base b2 = b.clone();  
Base b3 = d.clone();  
Derived d2 = b.clone();
```

## The Intuition

```
Base b = new Base;  
Derived d = new Derived;  
  
Base b2 = b.clone();  
Base b3 = d.clone();  
Derived d2 = b.clone();  
Derived d3 = d.clone();
```

## The Intuition

```
Base b = new Base;  
Derived d = new Derived;  
  
Base b2 = b.clone();  
Base b3 = d.clone();  
Derived d2 = b.clone();  
Derived d3 = d.clone();  
  
Base reallyD = new Derived;
```

## The Intuition

```
Base b = new Base;  
Derived d = new Derived;  
  
Base b2 = b.clone();  
Base b3 = d.clone();  
Derived d2 = b.clone();  
Derived d3 = d.clone();  
  
Base reallyD = new Derived;  
Base b4 = reallyD.clone();
```

## The Intuition

```
Base b = new Base;  
Derived d = new Derived;  
  
Base b2 = b.clone();  
Base b3 = d.clone();  
Derived d2 = b.clone();  
Derived d3 = d.clone();  
  
Base reallyD = new Derived;  
Base b4 = reallyD.clone();  
Derived d4 = reallyD.clone();
```

## The Intuition

```
Base b = new Base;  
Derived d = new Derived;  
  
Base b2 = b.clone();  
Base b3 = d.clone();  
Derived d2 = b.clone();  
Derived d3 = d.clone();  
  
Base reallyD = new Derived;  
Base b4 = reallyD.clone();  
Derived d4 = reallyD.clone();
```

## Is this Safe?

$f$  is an identifier.

$S \vdash e_0 : M$

$f$  is a member function in class  $M$ .

$f$  has type  $(T_1, \dots, T_n) \rightarrow U$

$S \vdash e_i : R_i$  for  $1 \leq i \leq n$

$R_i \leq T_i$  for  $1 \leq i \leq n$

---

$S \vdash e_0.f(e_1, \dots, e_n) : U$



## Is this Safe?

$f$  is an identifier.  
 $S \vdash e_0 : M$

$f$  is a member function in class  $M$ .  
 $f$  has type  $(T_1, \dots, T_n) \rightarrow U$   
 $S \vdash e_i : R_i$  for  $1 \leq i \leq n$   
 $R_i \leq T_i$  for  $1 \leq i \leq n$

---

$S \vdash e_0.f(e_1, \dots, e_n) : U$

This refers to the  
static type of the  
function.

## Is this Safe?

$f$  is an identifier.  
 $S \vdash e_0 : M$

$f$  is a member function in class  $M$ .  
 $f$  has type  $(T_1, \dots, T_n) \rightarrow U$

$S \vdash e_i : R_i$  for  $1 \leq i \leq n$   
 $R_i \leq T_i$  for  $1 \leq i \leq n$

---

$S \vdash e_0.f(e_1, \dots, e_n) : U$

This refers to the  
static type of the  
function.

$f$  has dynamic type  
 $(T_1, T_2, \dots, T_n) \rightarrow V$   
and we know that  
 $V \leq U$

## Is this Safe?

$f$  is an identifier.  
 $S \vdash e_0 : M$

$f$  is a member function in class  $M$ .  
 $f$  has type  $(T_1, \dots, T_n) \rightarrow U$

$S \vdash e_i : R_i$  for  $1 \leq i \leq n$   
 $R_i \leq T_i$  for  $1 \leq i \leq n$

---

$S \vdash e_0.f(e_1, \dots, e_n) : U$

This refers to the  
static type of the  
function.

$f$  has dynamic type  
 $(T_1, T_2, \dots, T_n) \rightarrow V$   
and we know that

$V \leq U$

so the rule is sound!

## Kovarijanta povratnog tipa

- Šta se dešava ako dozvolimo kovarijante povratnog tipa, tj ako dozvolimo da predefinisana funkcija ima kao povratni tip svoj klasni tip umesto nadklasnog tipa?
- Kovarijante povratnog tipa su bezbedne, tj pravilo koje definiše tipove je i dalje saglasno jer je sa kovarijantom povratnog tipa obezbeđeno važenje uslova  $DynamicType(E) \leq StaticType(E)$
- Mnogi programski jezici obezbeđuju kovarijante povratnog tipa, na primer C++ i Java

## Različite varijante funkcija

**Kovarijanta po argumentu** Da li u okviru klasa možemo da dozvolimo da predefinisana funkcija ima kao argument svoj klasni tip umesto nadklasnog tipa?

**Primer** Da li za predefinisanu funkciju  $f$  argument funkcije može da bude *Mačka* umesto *Sisar*?

Ako je  $f$  u klasi *Sisar* definisana sa

```
void f(Sisar s)
```

da li u klasi *Mačka* funkcija  $f$  može da bude predefinisana sa

```
void f(Mačka m)?
```

## Kovarijanta po argumentu

- Neka funkcija A predefiniše funkciju B, ali neka se ove funkcije razlikuju po bar jednom argumentu: Funkcija A je *kovarijanta po argumentu* funkciji B ukoliko se argumenti funkcija poklapaju ili se neki argumenti funkcije A mogu pretvoriti (konvertovati) u odgovarajuće argumente funkcije B.
- Da li je bezbedno dozvoliti kovarijante po argumentu?

## Relaxing our Restrictions (Again)

```
class Base {
    bool equalTo(Base B) {
        /* ... */
    }
}

class Derived extends Base {
    bool equalTo(Base B) {
        /* ... */
    }
}
```

## Relaxing our Restrictions (Again)

```
class Base {
    bool equalTo(Base B) {
        /* ... */
    }
}

class Derived extends Base {
    bool equalTo(Base B) {
        /* ... */
    }
}
```



## Relaxing our Restrictions (Again)

```
class Base {
    bool equalTo(Base B) {
        /* ... */
    }
}

class Derived extends Base {
    bool equalTo(Derived B) {
        /* ... */
    }
}
```

## Relaxing our Restrictions (Again)

```
class Base {
    bool equalTo(Base B) {
        /* ... */
    }
}

class Derived extends Base {
    bool equalTo(Derived D) {
        /* ... */
    }
}
```

## Relaxing our Restrictions (Again)

```
class Base {  
    bool equalTo(Base B) {  
        /* ... */  
    }  
}  
  
class Derived extends Base {  
    bool equalTo(Derived D) {  
        /* ... */  
    }  
}
```

Is this safe?

## Is this Safe?

$f$  is an identifier.

$S \vdash e_0 : M$

$f$  is a member function in class  $M$ .

$f$  has type  $(T_1, \dots, T_n) \rightarrow U$

$S \vdash e_i : R_i$  for  $1 \leq i \leq n$

$R_i \leq T_i$  for  $1 \leq i \leq n$

---

$S \vdash e_0.f(e_1, \dots, e_n) : U$

## Is this Safe?

$f$  is an identifier.  
 $S \vdash e_0 : M$

$f$  is a member function in class  $M$ .  
 $f$  has type  $(T_1, \dots, T_n) \rightarrow U$

$S \vdash e_i : R_i$  for  $1 \leq i \leq n$   
 $R_i \leq T_i$  for  $1 \leq i \leq n$

---

$S \vdash e_0.f(e_1, \dots, e_n) : U$

This refers to the  
static type of the  
function.

## Is this Safe?

$f$  is an identifier.  
 $S \vdash e_0 : M$

$f$  is a member function in class  $M$ .  
 $f$  has type  $(T_1, \dots, T_n) \rightarrow U$

$S \vdash e_i : R_i$  for  $1 \leq i \leq n$   
 $R_i \leq T_i$  for  $1 \leq i \leq n$

---

$S \vdash e_0.f(e_1, \dots, e_n) : U$

This refers to the  
static type of the  
function.

$f$  has dynamic type  
 $(V_1, V_2, \dots, V_n) \rightarrow U$   
and we know that  
 $V_i \leq T_i$  for  $1 \leq i \leq n$

## Is this Safe?

$f$  is an identifier.  
 $S \vdash e_0 : M$

$f$  is a member function in class  $M$ .  
 $f$  has type  $(T_1, \dots, T_n) \rightarrow U$

$S \vdash e_i : R_i$  for  $1 \leq i \leq n$

$R_i \leq T_i$  for  $1 \leq i \leq n$

---

$S \vdash e_0.f(e_1, \dots, e_n) : U$

$R_i \leq T_i$  for  $1 \leq i \leq n$   
 $V_i \leq T_i$  for  $1 \leq i \leq n$

This refers to the static type of the function.

$f$  has dynamic type  $(V_1, V_2, \dots, V_n) \rightarrow U$  and we know that

$V_i \leq T_i$  for  $1 \leq i \leq n$

## Kovarijante po argumentu su nebezbedne!

- Da li u okviru klasa možemo da dozvolimo da predefinisana funkcija ima kao argument svoj klasni tip umesto nadklasnog tipa?
- Dozvoliti potklasi da napravi restrikciju tipa svog parametra funkcije je **suštinski nebezbedno!**
- Pozivi kroz baznu klasu mogu da pošalju objekat pogrešnog tipa
- Zbog toga, Javin metod `Object.equals` uzima kao argument drugi `Object`
- U nekim jezicima ovo nije dobro urađeno: na primer, Eiffel dozvoljava kovarijante po argumentima i posledica toga je da na taj način neispravni programi mogu da promaknu sistemu tipova i da zato dođe do greške u fazi izvršavanja
- Sledi konkretan primer kako kovarijante po argumentu mogu da naprave problem u fazi izvršavanja



## A Concrete Example

## A Concrete Example

```
class Fine {  
    void nothingFancy(Fine f) {  
        /* ... do nothing ... */  
    }  
}
```

## A Concrete Example

```
class Fine {  
    void nothingFancy(Fine f) {  
        /* ... do nothing ... */  
    }  
}  
  
class Borken extends Fine {  
    int missingFn() {  
        return 137;  
    }  
    void nothingFancy(Borken b) {  
        Print(b.missingFn());  
    }  
}
```

## A Concrete Example

```
class Fine {
    void nothingFancy(Fine f) {
        /* ... do nothing ... */
    }
}

class Borken extends Fine {
    int missingFn() {
        return 137;
    }
    void nothingFancy(Borken b) {
        Print(b.missingFn());
    }
}

int main() {
    Fine f = new Borken;
    f.nothingFancy(new Fine);
}
```

## A Concrete Example

```
class Fine {
    void nothingFancy(Fine f) {
        /* ... do nothing ... */
    }
}

class Borken extends Fine {
    int missingFn() {
        return 137;
    }
    void nothingFancy(Borken b) {
        Print(b.missingFn());
    }
}

int main() {
    Fine f = new Borken();
    f.nothingFancy(new Fine());
}
```

## A Concrete Example

```
class Fine {
    void nothingFancy(Fine f) {
        /* ... do nothing ... */
    }
}

class Borken extends Fine {
    int missingFn() {
        return 137;
    }
    void nothingFancy(Borken b) {
        Print(b.missingFn());
    }
}

int main() {
    Fine f = new Borken();
    f.nothingFancy(new Fine());
}
```

(That calls this one)

## A Concrete Example

```
class Fine {  
    void nothingFancy(Fine f) {  
        /* ... do nothing ... */  
    }  
}  
  
class Borken extends Fine {  
    int missingFn() {  
        return 137;  
    }  
    void nothingFancy(Borken b) {  
        Print(b.missingFn());  
    }  
}  
  
int main() {  
    Fine f = new Borken;  
    f.nothingFancy(new Fine);  
}
```

(That calls this one)

## Različite varijante funkcija

**Kontravarijanta po argumentu** Da li u okviru klase možemo da dozvolimo da predefinisana funkcija ima kao argument tip superklase umesto nadklasnog tipa?

**Primer** Da li za predefinisanu funkciju  $f$  argument funkcije može da bude *Životinja* umesto *Sisar*?

Ako je  $f$  u klasi *Sisar* definisana sa

```
void f(Sisar s)
```

da li u klasi *Mačka* funkcija  $f$  može da bude predefinisana sa

```
void f(Životinja m)?
```



## Kontravarijanta po argumentu

- Neka važi  $C_a \leq C_b \leq C_c$ , tj neka je  $C_c$  bazna klasa za klasu  $C_b$ , a  $C_b$  bazna klasa za klasu  $C_a$ . Neka funkcija A (klase  $C_a$ ) predefiniše funkciju B (klase  $C_b$ ), ali neka se razlikuju tipovi arugmenata, tj: funkcija A je kontravarijanta po argumentu funkciji B ukoliko A ima za argument tip superklase  $C_c$  umesto tip  $C_b$ .
- Da li je bezbedno dozvoliti kontravarijante po argumentu?

## Contravariant Arguments

```
class Super {}  
class Base extends Super {  
    bool equalTo(Base B) {  
        /* ... */  
    }  
}  
  
class Derived extends Base {  
    bool equalTo(Base B) {  
        /* ... */  
    }  
}
```

## Contravariant Arguments

```
class Super {}  
class Base extends Super {  
    bool equalTo(Base B) {  
        /* ... */  
    }  
}  
  
class Derived extends Base {  
    bool equalTo(Base B) {  
        /* ... */  
    }  
}
```

## Contravariant Arguments

```
class Super {}  
class Base extends Super {  
    bool equalTo(Base B) {  
        /* ... */  
    }  
}  
  
class Derived extends Base {  
    bool equalTo(Super B) {  
        /* ... */  
    }  
}
```

## Contravariant Arguments

```
class Super {}  
class Base extends Super {  
    bool equalTo(Base B) {  
        /* ... */  
    }  
}  
  
class Derived extends Base {  
    bool equalTo(Super B) {  
        /* ... */  
    }  
}
```

## Kontravarijante po argumentima su bezbedne

- Da li u okviru klasa možemo da dozvolimo da predefinisana funkcija ima kao argument tip superklase umesto nadklasnog tipa?
- **Kontravarijante po argumentima su bezbedne**: intuitivno, kada zovemo tu funkciju kroz baznu klasu, funkcija će prihvatiti bilo šta što bi bazna klasa već prihvatila
- Ipak, većina jezika ne podržava kontravarijante po argumentima
  - Povećavaju kompleksnost kompajlera i specifikacije jezika
  - Povećavaju kompleksnost proveravanja predefinisanih metoda

## Contravariant Overrides

```
class Super {}  
class Duper extends Super {}  
class Base extends Super {  
    bool equalTo(Base B) {  
        /* ... */  
    }  
}  
  
class Derived extends Base {  
    bool equalTo(Super B) {  
        /* ... */  
    }  
    bool equalTo(Duper B) {  
        /* ... */  
    }  
}
```

## Contravariant Overrides

```
class Super {}  
class Duper extends Super {}  
class Base extends Super {  
    bool equalTo(Base B) {  
        /* ... */  
    }  
}  
  
class Derived extends Base {  
    bool equalTo(Super B) {  
        /* ... */  
    }  
    bool equalTo(Duper B) {  
        /* ... */  
    }  
}
```

Two overrides?  
Or an overload  
and an override?



## Zaključak

- Potrebno je biti jako pažljiv u dizajnu programskih jezika i u uvođenju novih karakteristika u statički tipizirane jezike
- Lako je dizajnirati karakteristike jezika ali je teško dizajnirati karakteristike jezika koje omogućavaju da se kreiraju bezbedni sistemi tipova
- Sistem tipova nam može pomoći da detektujemo greške u dizajnu jezika

## Zaključak

- Proširili smo sistem tipova da može da uključi i provere da li su naredbe dobro formirane
- Tip greške se može pretvoriti u sve druge tipove i on nam služi za sprečavanje širenja grešaka
- Preopterećivanje se razrešava u fazi kompilacije i određuje koja od različitih funkcija će biti pozvana
- Zbog preopterećivanja je potrebno rangirati funkcije kako bi se odredio najbolji izbor
- Bezbedno je da funkcije budu kovarijantne po povratnom tipu ili kontravarijantne po tipu argumenta

# Pregled

- 1 Određivanje tipova u izrazima
- 2 Provera tipova u okviru naredbi
- 3 Literatura**

## Literatura

- (The Dragon Book) Compilers: Principles, Techniques, and Tools Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman
- Kompajleri Stanford  
<https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/>