COM SCI 132 Week 3

Aidan Jan

April 15, 2024

Type Checking Continued

Review: we have expressions $\texttt{A} \vdash e : t$ and statements $\texttt{A} \vdash \texttt{s}$, where

- A represents the symbol table (type environment)
  - Must be searched in the order: local variables, parameters, then fields
- s represents a statement
- e represents an expression
- t represents a data type (out of {`int, bool, int[], C`})
  - `C` represents some user-defined class

Type Checking Methods

Methods are written in the format:

$$t_r \texttt{ m (}t_a\texttt{ a) \{}t_l\texttt{ x; s; return e\}}$$

- t_r is the return type
- `m` is the method name
- t_a is the type of the parameter `a`
- `a` is the parameter
- t_l is the type of the local variable
- `x` is a local variable
- `s` is a statement
- `e` is the return value (which must have type t_r)

Additionally,

$$\frac{\texttt{A = fields} \cdot \texttt{(a : }\ t_a\texttt{, k : }\ l_l\texttt{), A} \vdash \texttt{s, A} \vdash \texttt{e : }\ t_r}{t_r\texttt{ m (}t_a\texttt{ a) \{}t_l\texttt{ x; s; return e\}}}$$

For a method call,

$$\frac{A \vdash e_0 : \texttt{C}, \mathbf{c}, A \vdash e : t_a}{A \vdash e_0 \cdot m(e) : t_r}$$

where **c** refers to

```
class c {
    // fields
    ...
    t_r m (t_a a) { s }
}
```

and t_a represents the type of the parameter in **c**.

Objects

- In Java (and miniJava), objects are created with the `new` keyword.
- This stores the object in the symbol table, along with any object variables (fields) and their types.

Subtyping

Consider the following representations of a number: byte, short, int, long, double. In increasing order, byte has 8 bits of storage, a short 16, an int 32, and a long and double 64. Due to the increasing bit lengths, a 'bigger' data type can contain 'smaller' types. For example,

```
int a = 0;
long b = 0;
b = a;
```

The above is possible since a long is big enough to store all the data an int contains. However,

```
a = b;
```

is not possible, because an int cannot contain a long.

Subtyping with Classes

A class can inherit another class with the keyword `extends`. When a class is inherited, the class inheriting gains all the functions and private variables (fields) of the inherited class. For example,

```
class A { ... }
class B extends A { ... }

A a = new A(...);
B b = new B(...);

A = B;
```

Setting A to B is valid since A can contain the data B has, in that all of A's fields will be filled. However, setting `B = A` is invalid since B is a subtype of A, and B has less functionality than A.
Example: (ColorPoint $\subseteq$ Point)

```
class Point {
    public Point() { ... }
    public void move() { ... }


}
class ColorPoint extends Point {
```

```
    public ColorPoint() { ... }
    public void color() { this.move(); ... }

}

class Main {
    public static void main(String[] args) {
        Point p;
        ColorPoint q;

        p = q; // legal!
        q = p; // illegal!

        q.color();
    }

}
```

Remember that if $t_e \subseteq t_x$, then

$$\frac{x : t_x, e : t_e}{\vdash x = e}$$

Everything done on a `p` can be done on a `q`, but not the reverse, because `q` extends `p`.