COM SCI 132 Week 3

Aidan Jan

April 17, 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 $\{\texttt{int, bool, int[], C}\}$)
  - `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`.

Sparrow

- Program `p ::=` $F_1 \ \ldots F_m$
- FunDecl `F ::= func f (`$id_1 \ \ldots id_f$`) b`
- Block `b ::=` $i_1 \ \ldots i_n$ `return id`
- Instruction `i ::= l:  | id=@f | id = id + id | ...| id = [id + c] | [id + c] = id | id = id | id = alloc(id) | print(id) | goto l | if0 id goto l | id = call id(id...id)`
  - ...includes subtract, greater than, less than, etc. operators
  - in `[id + c]`, `id` is the heap address, `c` is the offset (measured in bytes).
  - `l:` is a label
  - `if0` is a conditional check that executes the goto if the variable is zero. This is equivalent to the JZ processor command in x86 assembly.
  - Using `call` on an id works because identifiers can also be functions.

Sparrow Rules

$$\frac{\text{hypothesis}_0 \ldots \text{hypothesis}_n}{\text{conclusion}}$$

- Values: integers `c`; heap address with offset (a, c); function names `f`
- The heap: `H`: map from heap addresses to tuples of values
- Environment: `E`: map from identifiers to values
- Program state: p, H, b^*, E, b
  - p is the program (never modified; program being executed stays the same)
  - H is the heap (heap changes as program is executed)

  - b^* is the function being executed (in a way, a bigger block, changes only when change of control occurs)
  - E is the environment
  - b is the block of code currently being executed. (e.g., a loop, conditional block, etc., same as b^* at the start.)

Program States

Assignment to constant

Suppose we add an extra statement in front of b, such that the statement is executed first. Like:

$$(p, H, b^*, E, (\texttt{id = c}) \cdot b)$$

What happens on the next step?

$$(p, H, b^*, E \cdot [id \to c], b)$$

- An id is now assigned to the number c and stored in the environment
- The next statement being executed is the first statement in b

Assignment to expression

What if we have $(p, H, b^*, E, id = (id_1 - id_2) \cdot b)$?
The next step is more complex since we must check that id_1 and id_2 are both integers.

$$\begin{cases} (p, H, b^*, E \cdot [id \to (c_1 - c_2)], b) & \text{if } id_1 \text{ and } id_2 \text{ map to constants} \\ \text{Error} & \text{otherwise} \end{cases}$$

Assignment to variable on heap

If we now have $(p, H, b^*, E, (id = [id_1 + c]) \cdot b)$, where id is being assigned to another variable stored in the heap, then,

$$\begin{cases} (p, H, b^*, E \cdot [id \to ((H(a_1))(c_1 + c))], b) & \text{if E}(id_1) = (a_1, c_1) \text{ and } (c_1 + c) \in H \\ \text{Error} & \text{otherwise} \end{cases}$$

- Both variable checks for id_1 being an integer, and a range check of location $[id_1 + c]$ being in the heap must be passed to not result in an error
- $H(a_1)$ is a tuple

Assignment of location on heap to identifier

If we are assigning a location in the heap to some identifier, so $(p, H, b^*, E, ([id_1 + c]) \cdot b)$, then the next step would be

$$\begin{cases} (p, H \cdot [a_1 \to t], b^*, E, b) & \text{if } E(id_1) = (a_1, c_1) \text{ and } (c_1 + c) \in \text{dom}(H(a_1)) \text{ and } t = H(a_1) \cdot [(c_1 + c) \to E[id]] \\ \text{Error} & \text{otherwise} \end{cases}$$

- Range check still must be done
- $H(a_1)$ is a tuple
- Notice that this time, the heap changes instead of the environment

Function calls

If we have a function call: $(p, H, b^*, E, (id = call id_0(id_1)) \cdot b)$, many things change.
First, we need to perform checks for the following:

- $E[id_0] = f$
- p contains the function $f(id'_1 \ldots id'_f)$

If the checks pass, we get:

$$(p, H', b^*, E \cdot [id \to E'(id')], b)$$

where $E'(id')$ is the result of executing from the following program state:

$$(p, H', b', E', b' \cdots \texttt{return}\ id')$$

- Since a change in control occurs, the heap can be a completely new heap on the next instruction.
- However, on the `return` statement, the program must return to the original heap. As a result, an identifier in the environment is assigned to the **return value** of the function, if there is one.
- Note that the H' carries over into the original program state, since the other function can also modify the same heap.