Type Checking Java

Last lecture, we saw different kind of errors, and I indicated that some of those errors can be detected at compile-time, that is, before executing the program, and some can only be detected at run-time.

The purpose of this lecture is to illustrate how Java can check the absence of type errors and method inexistence errors at compile-time, by seeing how types and especially *type checking* is performed.

What we are after is an algorithm that runs over the source code of our program, and without actually running the code, returns either (1) a guarantee that when the code will execute it will not cause either a type error or a method-does-not-exist error, of (2) point out that there is a possible type error or method-does-not-exist error, and indicate where the error may happen.

Note my use of qualification, such as "*possible* type error", and "where the error *may* happen". This is because the algorithm is good, but not that good. Consider the following snippets of code.

```
if (x==0)
  return 10 + 10;
else
  return 10 + true;
```

Does this snippet of code cause an error when executed? Well, it depends on the value of variable `x` when it executes. Meaning that it *may* cause an error. And if we don't know in advance what the value of `x` is, we cannot say anything more. (For instance, suppose that `x` is input by the user.)

```
if (x==x)
  return 10 + 10;
else
  return 10 + true;
```

This is more interesting. It never causes an error when executed, because `x==x` will always be true, no matter what `x` is. Conversely, the following code:

```
if (!(x==x))
  return 10 + 10;
else
  return 10 + true;
```

always causes an error when executed.

We want an algorithm that correctly identifies the cases above. Let's keep them in mind.

Instead of defining a type-checking algorithm for all of Java, let me instead define a "mini-Java" that captures the essence of Java. It is a very simplified programming language, but make no mistake, it is a programming language.

The type-checking algorithm works by taking a program as input. So I need to define the syntax in which we write out programs. The syntax is written using a so-called *BNF grammar*[1]. This kind of grammar lets you define the form that different syntactic elements take, and those form can be recursive.

```
// syntax for types
T,U,V ::=  void
           int
           boolean
           C          (where C is a class name)

// syntax for expressions
E ::=  true, false
       0,1,2,...,-1,-2,...
       this
       x              (where x is a variable or a name)
       E_1 + E_2
       E_1 == E_2
       E.f
       E_1.m(E_2)
       new C(E_1,...,E_k)

// syntax for statements
S ::=  E_1.m(E_2);
       return E;
       { Block }
       x = E;
       if E S_1 else S_2

// syntax for blocks of statements
```

[1]BNF = Backus-Naur Form

```
Block ::= T_1 x_1; ...; T_k X_k; S_1 ... S_n
```

A class is is of the form:

```
class C {
  T_1 f_1;
  ...
  T_k f_k;

  U_1 m_1 (V_1 x_1) { Block_1 }
  ...
  U_n m_n (V_n x_n) { Block_n }
```

Note that this is essentially the syntax of Java, but simplified. I will rely on your intuition of the execution model of Java to understand how programs in this babyJava execute. The only bit of subtlety is how instances of class are created. The `new C(E_1,...E_k)` operator first evaluates `E_1`, ..., `E_k` to values `v_1`, ..., `v_k`, and creates a new instance of class `C` with fields initialized to `v_1`, ..., `v_k`.

A program is a set *CLS* of classes. The type checker takes this set of classes, and issues a guarantee of error-freeness, or report a possible type error.

Type checking a set of classes amounts to checking each class separately, but remembering the full set of classes, because we may need to refer to other classes when checking the current class.

```
checkClasses (CLS) =
  for each class C in CLS, checkClass(CLS,C)
```

Type checking a class simply checks every method of the class separately, by checking the block of statements making up the body of the method.

```
checkClass (CLS,C) =
  * If class C is in CLS and is of the form:
      'class C {
         T_1 f_1;
         ...
         T_k f_k;
         U_1 m_1 (V_1 x_1) { Block_1 }
         ...
         U_n m_n (V_n x_n) { Block_n }
       }':
    checkBlock(CLS, C, {V_1 x_1}, Block_1, U_1)
```

```
    ...
    checkBlock(CLS, C, {V_n x_n}, Block_n, U_n)
```

Note that when checking a block, we remember that the argument to the method is assumed to have a certain type, and that the method is specified to return a specific type (which will let us check that a `return` in the body of the method returns the right type of value). We also carry through the current class `C`, remembering that class in which the method being checked lives.

Checking a block of statements amounts to checking every statement in the block. Note that we record in a *variable environment* that type of all variables declared at the top of the block. This will be handy when we read off a value from a variable, or assign to a variable, since the environment will keep track of the type of all variables that are accessible.

```
checkBlock (CLS, C, ENV, Block, T_ret) =
  * If Block is of the form 'T_1 x_1; ...; T_k x_k; S_1 ... S_n':
     let NEWENV = ENV union { T_1 x_1, ..., T_k x_k }
     checkStatement(CLS, C, NEWENV, S_1, T_ret)
     ...
     checkStatement(CLS, C, NEWENV, S_n, T_ret)
```

Checking an individual statement is where a lot of the action takes place. Depending on the statement at hand, we check different things.

```
checkStatement (CLS, C, ENV, S, T_ret) =
  * If S is of the form 'E_1.m(E_2)':
      let T_1 = typeOf(CLS, C, ENV, E_1)
      if T_1 is not a C_1 in CLS,
        then report type error ("E_1 not of a known class")
      if C_1 has no method m in its class definition,
        then report type error ("Class of E_1 does not define method m")
      let definition of m in C_1 be 'U m (V x) { Block }'
      let T_2 = typeOf(CLS, C, ENV, E_2)
      if not (T_2 = V),
        then report type error ("Argument does not agree m's signature")
  * If S is of the form 'return E':
      let T = typeOf(CLS, C, ENV, E)
      if not (T = T_ret),
        then report type error ("Return type of method does not agree")
  * If S is of the form '{ Block }':
      checkBlock(CLS, C, ENV, Block, T_ret)
  * If S is of the form 'x = E':
```

```
      let T = typeOf(CLS, C, ENV, E)
      let T' = lookup type of 'x' in ENV
      if not (T = T'),
        then report type error ("Assignment types do not agree")
  * If S is of the form 'if E S_1 else S_2':
      let T = typeOf(CLS,C,ENV,E)
      if not (T = boolean),
        then report type error ("Conditional expects boolean condition")
      checkStatement(CLS,C,ENV,S_1,T_ret)
      checkStatement(CLS,C,ENV,S_2,T_ret)
```

Finally, the following function returns the type of an expression, depending on the expression form.

```
typeOf (CLS, C, ENV, E) =
  * If E is of the form 'true' or 'false':
      return boolean
  * If E is of the form '0,1,2,...,-1,-2,...':
      return int
  * If E is of the form 'this':
      return C
  * If E is of the form 'x', where x is a variable or a name:
      let T = lookup type of 'x' in ENV
      return T
  * If E is of the form 'E_1 + E_2':
      let T_1 = typeOf(CLS, C, ENV, E_1)
      let T_2 = typeof(CLS, C, ENV, E_2)
      if not (T_1 = T_2 = int),
        then report type error ("+ operation applies to integers only")
      return int
  * If E is of the form 'E_1 == E_2':
      let T_1 = typeOf(CLS, C, ENV, E_1)
      let T_2 = typeof(CLS, C, ENV, E_2)
      if not (T_1 = T_2),
        then report type error ("== operation applies to same types")
      return boolean
  * If E is of the form 'E_1.f':
      let T_1 = typeOf(CLS, C, ENV, E_1)
      if T_1 is not a C_1 in CLS,
        then report type error ("E_1 not of a known class")
      if C_1 has no field f in its class definition,
        then report type error ("Class of E_1 does not define field f")
```

```
      let definition of f in C_1 be 'T f'
      return T
  * If E is of the form 'E_1.m(E_2):
      let T_1 = typeOf(CLS, C, ENV, E_1)
      if T_1 is not a C_1 in CLS,
        then report type error ("E_1 not of a known class")
      if C_1 has no method m in its class definition,
        then report type error ("Class of E_1 does not define method m")
      let definition of m in C_1 be 'U m (V x) { Block }'
      let T_2 = typeOf(CLS, C, ENV, E_2)
      if not (T_2 = V),
        then report type error ("Argument does not agree m's signature")
      return U
  * If E is of the form 'new C_1(E_1,...,E_n)':
      If C_1 is not in CLS,
        then report type error ("Class C_1 not known")
      let T_1 f_1; ...; T_k f_k be the fields in C_1
      if not (k = n),
        then report type error ("Wrong arguments number to construct C")
      // so, k=n
      let U_1 = typeOf(CLS, C, ENV, E_1)
      ...
      let U_n = typeOf(CLS, C, ENV, E_k)
      if not (U_1 = T_1) or not (U_2 = T_2) or ... or not (U_k = T_k)
        then report type error ("Argument types do not agree with fields")
      return C_1
```

It is possible to prove, although I won't do it here because it is long and requires a precise description of the execution model, that if the above algorithm does not report a type error when we invoke `checkClasses(CLS)` for a set `CLS` of classes, then executing any method in any class in `CLS` with an argument of the right type does not cause a type error or method-not-found error at execution.

The algorithm is conservative—it deems that some programs may have a type error even though there is none that occur when the program is executed. For instance, it is easy to check, by looking at how `checkStatement` works, that the above code snippet:

```
if (x==x)
  return 10 + 10;
else
  return 10 + true;
```

assuming it occurs in the context of a program that otherwise type checks, would fail to type

check: the algorithm reports a type error. That's simply because the algorithm does not take advantage of possible information it may have about whether there are guarantees that only one branch be taken. It always checks that both branches type check, even if one will never be taken when the code executes. This is a choice that the designers of the algorithm made: such choices depend on complexity of the resulting type checking algorithm, uniform behavior that is more easily predicted by the user, etc.