CSE-504 Programming Assignment #4

Type Checking

Due: Wed., Apr. 25, 2012

Description

In this assignment, you will write a type checker for the `Decaf` compiler. You will simply methods "typecheck" and "typeinfer" to the AST classes. The AST `Decaf` has refined slightly (from Assignment #3) to make type checking code more straightforward (see below for the differences). You are also provided with the object code for AST generation and the full set of header files needed to compile your type checker. You are also provided with a driver program that generates the AST and invokes your type checker.

Refinement of AST Structure

The AST data structures has been refined in the following ways.

- **Types:** Four new types have been added: `InstanceType`, `ArrayType` and `UniverseType`. The type `ClassType` is used to denote the type of a class identifier. E.g., if you have a class in a Decaf program called `Foo`, and you use `Foo.x` access some static field `x` in `Foo`, then the identifier `Foo` will be marked with `ClassType`.

  `ArrayType` is used to denote the type of an array variable (or field). In the old AST, the "type" of the variable used to refer to the *element* type of the array; there was a separate field, called "dimensions" that was used to record the dimensions of the array. In the refined AST, this has been simplified: the type of the variable will be an `ArrayType`. For instance, a variable declared as `int a[];` will have `ArrayType(IntType)` as its type; and a variable declared as `float x[][]` will have `ArrayType(ArrayType(FloatType))` as its type.

  `InstanceType` is used to denote the type of an instance identifier. E.g., if you have a variable `bar` declared to be of type `Foo`, note that `bar` will refer to an object of type `Foo`. This is denoted by an instance type (with its `classtype` attribute set to `Foo`).

  `UniverseType` is used to represent an arbitrary type (more on this later, when the subtype relation is defined).

  `NullType` is used to represent the type of `null`: i.e. it is a subtype of every instance type.

- **lineno:** Statements and expressions have a line number field that has the source line number. This is useful for printing error messages.

- **Superclass:** Two changes have been made to the handling of superclasses:

  - Every class will extend from a new top "Object" class (similar to Java). This means that every class (except "Object" itself) will have a superclass. The symbol table is initialized with this "Object" class, and there is a global variable `objectclass` that refers to the "Object" class entity.
  - The superclass field of a class entity was a `Entity*` field in the old AST. This has been changed to a `ClassEntity*` field.

You have been provided with code that constructs the refined AST. The above discussion is for your information as you write the type checker.

Type Checking and Inference Rules

Your task in this assignment is to write the following functions:

- `Type* typeinfer()`: This is a member function in each `Expression` class (e.g. `BinaryExpression`, `AssignExpression`, etc.) that infers the type of the given expression.
- `void typecheck()`: This is a method in every *Statement* class, as well as in `ClassEntity`, `MethodEntity` and `ConstructorEntity` classes.

The type inference and type checking rules are described below.

typeinfer()

This method computes the type of an expression's result, based on the types of the expression's arguments.

- `BinaryExpression`:
  - *add, sub, mul, div*: Both arguments should be numeric (i.e. integers or floats). The result is an int if both arguments are int; and float otherwise.
  - *lt, leq, gt, geq*: Both arguments should be numeric (i.e. integers or floats). The result is a boolean.
  - *eq, neq*: The two arguments should be compatible: i.e one should be a subtype of another. The result is a boolean.
  - *and, or*: The two arguments should be boolean. The result is a boolean.
- `AssignExpression`: The right hand side should be a subtype of the left hand side. The result type is same as that of the right hand side.
- `UnaryExpression`:
  - *neg*: The argument should be boolean; the result is boolean.
  - *uminus*: The argument should be numeric; the result type is same as the argument type.
- `AutoExpression`: The argument should be an integer; the result is an integer.
- `ArrayAccess`: The base of array access should be an array type, say `ArrayType(`α`)`. The index should be an integer. The result then is α.
- `NewArrayInstance`: For an one-dimensional array, the `type` field gives the element type of the array (say α). Then the result of new array instance creation is of type `ArrayType(`α`)`. When an n-dimensional array with element type α is created, the result type is `ArrayType(ArrayType(`$\cdots$ $<$ n `times` $>$ `ArrayType(`α`)`$\cdots$`))`. Note that all array bounds should be integers.
- `FieldAccess`: This AST node has one of the more complicated typing rules, due to overriding and inheritance. A field access is of the form `b.x` where `b` is an expression, called the *base* and `x` is a field name. There are two cases we need to consider; one when we're accessing *non-static* fields and one when we're accessing *static* fields.
  - *Non-static field access*: Let the type of `b` be α; for a non-static field access, this type should be an `InstanceType`, say `InstanceType(`β`)` where β is a class. Then `x` should be a non-static field in class β, or some superclass of β. Let γ be the most specific super class of β (including β itself) in which a non-static `x` is defined. The field `x` of γ should either be *public*, or the this field access should be an expression in some method or constructor in class γ.

    The result of this field access is the type of field `x` in class γ.

– *Static field access*: Let the type of `b` be α; for a static field access, this type should be a `ClassType`, say `ClassType(`β`)` where β is a class. Let γ be the most specific super class of β (including β itself) in which a static `x` is defined. The field `x` of γ should either be *public*, or the this field access should be an expression in some method in class γ.

The result of this field access is the type of field `x` in class γ.

- `MethodInvocation`:This AST node has one of the more complicated typing rules due to overriding, inheritance and *overloading*.

  A method invocation is of the form `b.f(`*args*`)` where `b` is an expression called *base*, `f` is the method name, and *args* are actual arguments passed for this invocation.

  The resolution of an method invocations type is very similar to that of field access. In addition to searching for a method of the given name, we will also take into account the types of formal parameters and actual arguments, in order to take care of overloading.

  Let the sequence of types of actual arguments be *argtypes*. Let the sequence of types of formal parameters of the candidate method be *paramtypes*. We say that the candidate matches the invocation if (i) the method name in the invocation is same as the name of the candidate method, and (ii) each element of *argtypes* is a subtype of the corresponding element of *paramtypes*.

  Let β be the class corresponding to the type of *base*. Let γ be the most specific super class of β such that γ contains a method `f` matching the invocation. The class γ should contain exactly one such method. Then the result of the method invocation is the return type of `f` in class γ.

  The consideration of static/non-static and public/private attributes of methods is same as done for FieldAccess.

- `NewInstance`: This AST node corresponds to the invocation of a *constructor*. Overloading is resolved the same way as MethodInvocation, but only constructors in the given class (and not super classes) are considered. For example, if B is a subclass of A, then a new instance creation of the form `new B(`*args*`)` will consider only the constructors of `B`. Let the AST node represent creation of a new instance of class β (given by `class_entity()`). Then the result of the new instance creation is of type `InstanceType(`β`)`.

- `ThisExpression, SuperExpression`: For inferring the type of `this`, let the current class be γ. (When inferring the type of `super` let the super class of the current class be γ). If the current method is a *static* method, then the result type is `ClassType(`γ`)`; if the current method is *non-static*, then the result type is `InstanceType(`γ`)`.

- `IdExpression`: If the expression refers to a variable, then the result is that variable's type. If the expression refers to a class, say γ, then the result is `ClassType(`γ`)`.

- `NullExpression`: The type of `null` is `NullType`.

- *constants*: The type should be appropriate for the constant (e.g. int for integer constants, float for floating point constants, etc.).

When the above rules are violated (e.g. the arguments are of wrong type) then your type checker should print an appropriate error message; and the result should be an `ErrorType`.

Your type checker should also take care not to print cascaded error messages. For instance, let `b` be a boolean variable, and `i`, `j` be integer variables. Then the expression `(i*b)+j` has type error. Your type checker should print only one error message for this error.

typecheck() for Statements

This method checks the type correctness of a statement. The following is a description of the rules that define a type correct statement.

- `IfStatement`: The expression part must have a boolean type; the then and else parts must be type correct.
- `WhileStatement`: The expression part must have a boolean type; the body must be type correct.
- `ForStatement`: The guard must have a boolean type; the other parts, init, update, and body, must be type correct.
- `ReturnStatement`: The expression type must be a subtype of the current method's return type. A return statement is valid only when it occurs in a method (not a constructor).
- `BlockStatement`: Each statement in the block must be type correct.
- `ExprStatement`: The expression must be type correct (the type of the expression itself will be ignored).
- `Decl,Break,Continue,Skip`: These statements have no type checking done on them.

If a statement is not type correct, your type checker should output a suitable error message.

typecheck() for classes, methods and constructors

- `ClassEntity`: A class entity is type checked when every one of its method and constructor members are type checked.
- `MethodEntity`: A method entity is type checked when its body is type checked.
- `ConstructorEntity`: A constructor entity is type checked when its body is type checked.

The driver does type checking of the program by type checking all top-level (class) entities.

Subtypes

There are three special cases for subtypes:

- *Universe*: is a super type of everything.
- *Null*: is a subtype of every instance type.
- *Error*: is a subtype as well as a super type of every type.

  (Note that this definition makes the subtype relation not a partial order, but this is a useful hack: this ensures that error type is compatible with any type and hence errors do not cascade through the type checker).

Three kinds of types have non-trivial subtype relation:

- *ClassType*: `ClassType(`t_1`)` is a subtype of `ClassType(`t_2`)` if t_1 is a subclass of (or same as) t_2. Thus the subtype relationship among class types follows the subclass relationship.
- *InstanceType*: `InstanceType(`t_1`)` is a subtype of `InstanceType(`t_2`)` if t_1 is a subclass of (or same as) t_2. Thus the subtype relationship among instance types follows the subclass relationship.
- *ArrayType*: `ArrayType(`t_1`)` is a subtype of `ArrayType(`t_2`)` if t_1 is a subtype of t_2. Hence the subtype relationship among the element types induces the subtype relationship among array types.

For all other types, the subtype relationship is identity: e.g. `IntType` is a subtype of `IntType`, `FloatType` is a subtype of `FloatType`, etc.

The subtype relation between types is defined in `Types.cc`. So you do not need to encode this. The above discussion is for your information only.

Project Path

In the set of files given to you, you'll see a file called `TypeCheck.cc`. That file contains stubs for defining your type checking/inference methods (all these methods currently signal type error).

Start with classes containing methods with only assignment statements (and that too with constants). You'll then need to do type inference only for assignments. Add different kinds of expressions one by one. At some point, you can introduce other statements, and add type checking methods for those kinds of statements. This way, you can build the type checking/inference routines step by step, until all the methods are written.

In principle, you must be able to complete this assignment by modifying only `TypeCheck.cc`. If you need any additional functions, you can add these to `TypeCheck.cc`.

Available Material

The following material is available in the accompanying `.tgz` archive:

- A `src` directory containing source files for the driver, lexical analyzer, symbol table, AST and Type data structures. This directory also has the template `TypeCheck.cc` file that you will edit and submit.
- A `include` directory containing header files for the symbol table and AST data structures.
- A `tests` directory containing sample input files `in*` and the respective output files `out*`.
- An `obj` directory containing `decaf.tab.o` (parsing, AST construction) for Linux, Cygwin and Solaris.

Deliverables

You need to only submit the modified `TypeCheck.cc` file. If you added code to any other file, then archive the entire directory structure (include and src directories) along with the modified Makefile using tar and submit that archive.

Please use the assignment form on Blackboard to upload your submission.