CSE 504 Programming Assignment #5

Type Checker

1 Description

In this assignment, you will perform type checking for E--. As described in the class, this process involves two main tasks – inferring the types of expressions from that of its operands, and then checking if each operator or function in the program is being used with compatible argument types. In some cases, coercions will need to be introduced, e.g., when an integer is multiplied by a float. These coercions must be explicitly indicated in the AST.

You will do these tasks by providing the implementation of two virtual functions `const Type* typeCheck()` and `void typePrint()` for each of the subclasses of the `Ast` and `STEClasses` classes. The driver will call `typeCheck()` first to perform the above operations, and then call `typePrint()` to print the result. The output of `typePrint` will be similar to `print`, with the following differences: (a) types (rather than values) will be printed in the case of variables and values; (b) wherever expression coercion is needed, you will prefix the original type of the expression with "(*newtype*)" where *newtype* stands for coerced type.

When `typeCheck` is invoked, it should uncover as many distinct type errors as possible. For each error, print an error message that succinctly describes the kind of error detected. Your grade will depend on whether you can detect all type errors (a good thing) and whether you report the effects of the same error multiple times (not a good thing).

1.1 Detailed Description of Type Checking

Type checking involves two principal components. First we need to identify if two types are compatible. This is done using the `isSubType` method of the `Type` object. The implementation of this function is trivial for class types, i.e., A is a subtype of B iff A is the same as B. For basic types, subtype notion is a bit more involved, but still quite simple: byte is a subtype of int, int is a subtype of double, and so on. Among primitive types, integers are a subtype of floats. But booleans aren't a subtype of any type. Unsigned integers are a subtype of integers.

The second component of type checking are the rules that describe the types of arguments expected by operators and functions in the language. For E--, these rules are described below.

1.1.1 Typing Rules for Arithmetic Operators

- The binary operator % accepts two integer arguments. Its output type is an integer.
- All binary arithmetic operators, except %, take either an integral or floating point argument. The output type is the most general of the input types. (The definition of "most general" is consistent with the definition of subtype discussed above.) If the arguments have unequal types, then the less general type should be explicitly coerced to the more general type before the operator is applied. This is indicated by setting the field "coercedType" of the appropriate ExprNode.
- Relational operators take either integral or floating point argument. Comparison between a signed and unsigned expression **need not** produce a warning. The output type is boolean. Coercions must be introduced as mentioned in the previous case.
- Logical operators take two boolean arguments and their output type is boolean.
- `!` takes a boolean argument and its output is boolean as well.
- The unary minus operator takes either an integer or float argument. Its output type is the same as that of input type.

1.1.2 Type Checking of Expressions

- For function calls, the type of actual parameters must be a subtype of the corresponding formal argument type. Type coercions should be indicated as before.
- For literals, type checking involves setting and returning the type field associated with the `ValueNode`.

1.1.3 Statements

- For assignment statements, the type of the rhs must be a subtype of lhs. Integer expressions can be assigned to floating point numbers, and you required to indicate the necessary coercions. In addition, for assignments in rules, the lhs must be a global variable.

  You should set the output type of an assignment statement to be a boolean, so that uses of assignments within patterns will be permitted.

- For if-then-else statements, the condition must be boolean.

- For the return statement, ensure that the value being returned is a subtype of the return type declared for the function.

1.1.4 Rules

- Actual parameters to an event will have the types given by the declaration for that event.

- There cannot be any assignments to actual event parameters; assignments are permitted only on global variables.

- The pattern negation operation is permitted only on those patterns that don't have any sequencing operators (':' or '**'). Thus, `!e(x)`, `!(e1(x) \/ e2(y))`, `!(e1(x) \/ !e2(y))` and `!e1(x):e2(y)` are all type-correct, but `!(e1(x):e2(y))` and `!(e1(x) \/ e2(y)**)` are not. Virtual methods such as `hasSeqOps` have been declared for the `BasePatNode` to help perform this check.

2 Tips

- As usual, start small. Build up your type checker gradually. Simple expressions are again a good place to start. Once you get this right, add the rules for type checking statements, complex expressions and patterns.

- The type checker will perform an entire traversal of the AST and symbol tables that together capture the entire program.

- Comment out all but small sections of input test files when you start debugging. As you get your program working, slowly uncomment other portions of the test files.

- DO NOT WASTE TOO MUCH TIME TRYING TO GET YOUR OUTPUT TO EXACTLY MATCH THE STANDARD OUTPUT. The content, of course, should match whenever your program works right. The form may not match, although it is probably useful to get the form to match as much as possible, while making sure that you do not spend far too much time on it.

- MAKE SURE THAT YOU SET TYPE COERCIONS FLAGS where appropriate.

- MAKE SURE THAT TYPE ERRORS DO NOT PROPAGATE. As far as possible, try to identify the output type of an operator or function even if there is a type error in one of the arguments.

3 Available Material

Same as the last assignment, except that the header files have been modified slightly for type checking. You should put the new functions in separate files, e.g., nAst.C and nSTEClasses.C.