Compiler Project #3 - Type Checker

2012-11269 Dongjae Lim

2013-11403 Chansu Park

1. Fill missing GetType() functions

We have some AST nodes that don't perform GetType, so we have to finish them.

1) `CAstBinaryOp, CAstUnaryOp`

Operators take such types and make a value in such type. If an operator is one of the "+", "-", "*", or "/", its return type is integer, so return integer type. Otherwise, return boolean type.

2) `CAstSpecialOp`

SnuPL/1 applies these operations only in such special cases. Moreover, we cannot explicitely use these operations in the source code, so it is complicated to decide types on them.

First, if we perform reference operator on such object, it is a pointer to that object. It means '&' makes a new pointer pointing on it. Therefore, for the operator '&', return type which is a pointer to the type of the operand .

Similary, '*' takes pointer. If it is not a pointer, return NULL. Otherwise return its basetype.

Finally, if we perform type casting with a specific type `t`, return type `t`.

3) `CAstArrayDesignator`

Perform `CAstArrayDesignator` is bit more difficult. First, if it takes a pointer, we have to take its innertype and perform `GetType`. If it is still not an array, it is an error.

Also, if # of indices of it over its dimension, we also have to return an error.

After it ends, run a loop (# of indices) times. For each loop, take basetype of current type.

After it ends, return that type.

4) `CAstStringConstant`

For our language description, strings are immutable constants, but internally represented as character arrays. Therefore its type is a character array which length is (length of the string + 1). Return this.

2. Type Checking

1) `CAstScope`

It checks all of its statements and child scopes recursively. Likewise, if a node have child nodes, it checks all of them.

Ex. `CAstStatAssign` contains expression `LHS` and `RHS`, so `CAstStatAssign::TypeCheck()` calls `LHS->TypeCheck()` and `RHS->TypeCheck()` inside.

2) `CAstStatAssign`

Assignments have two conditions.

- SnuPL/1 doesn't support assignments between compound types. Therefore, if LHS or RHS returns array type, it returns error.
- LHS and RHS must have same type.

3) `CAstStatCall`

It is just a wrapper of `CAstFunctionCall`, so its process is trivial.

4) `CAstStatReturn`

Return contains three conditions.

- The return type of its function/procedure must be valid.
- The returning expression must be valid.
- The returning expression's type must be matched with return type of the function/procedure.

5) `CAstStatIf, CAstStatWhile`

The condition statement must have boolean type.

6) `CAstBinaryOp`

It performs different acts depend on its operator.

Operator	Condition
`opAdd, opSub, opMul, opDiv, opLessThan, opLessEqual, opBiggerThan, opBiggerEqual`	LHS and RHS expressions must have integer type.
`opAnd, opOr`	LHS and RHS expressions must have boolean type.
`opEqual, opNotEqual`	LHS and RHS expression must have scalar(integer, boolean, character) type. Also, they must have same type.

7) `CAstUnaryOp`

Similar to `CAstBinaryOp`, it performs different acts depend on its operator. Also we check boundary of integer on this AST - we will explain this logic on `CAstConstant`.

- If its operator is `opNot`, check its operand is boolean type or not. If not, return error.
- It its operator is `opPos` or `opNeg`, its operand must have integer type. Otherwise, return error.

8) `CAstSpecialOp`

Its process is similar to `CAstUnaryOp`.

- If its operator is `opDeref`, its operand must be a pointer type.

9) `CAstFunctionCall`

Take its procedure symbol, then check following conditions.

- The number of parameters must be fit into the function.
- The type of parameters must also be fit into the function. If we got an array as a parameter, we always wrap it by `CAstSpecialOp` with `opAddress`. Therefore, if a parameter is recognized as an array, we wrap it with pointer and compare.

10) `CArrayDesignator`

We took its type from symbol table. Only valid types can be put into the symbol table, so return true.

11) `CAstArrayDesignator`

It also have some conditions.

- The base type must be valid.
- Each indexes must be valid integer type.

12) `CAstConstant`

If its type is boolean or character, it already guaranteed to have valid value. But if it is integer type, we need special routine for check whether its value fits into the range or not.

a) Check integer's range

In the case integer, however, we have to check whether or not its value is in the range, [-2^{31}, $2^{31}-1$]. However, abs(lower bound) != abs(upper bound), so we cannot perform boundary checking during parsing. We solved this problem by using `TypeCheck()`'s value as a flag.

Value	Behavior
$> 2^{31}$	The parser can decide that the value is invalid.
$= 2^{31}$	`CAstConstant::TypeCheck()` returns false. If we got `opNeg` and the operand is constant integer which returns false on `TypeCheck()`, the value is -2^{31}. This is valid value, so we ignore the result.
$< 2^{31}$	This is always valid value.

13) `CAstStringConstant`

We construct this AST only if it is a valid string. Therefore only valid string can be a value of this AST, so return true.

3. parser

1) Repair tokens for correct line/column number

We have such bugs on parser. When we send token in each node, some functions sent dummy token which was not assigned, which make AST to print invalid locations on error messages. Therefore, we usually used first token as a token for each AST node to set valid location.

2) Handling for array type

There are many restrictions about using array type on the source codes:

- If it is not a parameter, every indices of that cannot be opened.
- Every function cannot return array.

Therefore, we added parameters onto `ReadType` function to represent these restrictions(`openArray`: allow opened array only if it is true, `beArray`: allow array only if it is true). After that, whenever we have to read types using `VarDecl` / `VarDeclaration`, use booleans to perform these restrictions.

We already treated duplicated declaration about variables, use variables without declaration, and so on when we have made parser last time.

4. Others

Reference parser showed a critical bug about sending size of arrays which is such a parameter of subroutine declaration. Our parser was shown it, either. We checked many source codes which is given, and we found that this bug occurs since duplication checking on pointer is performed using `Match` function. We changed it to use `Compare` function instead, and thus this bug disappeared.