CS323 Project Phrase 2 Report

12112421 王德涵 12112620 徐霄阳 12112822 孔繁初

Semantic rules

Assumptions

As described in the requirement document, we have implemented all the assumptions:

Assumption 1 Char variables only occur in assignment operations or function parameters/arguments.
Assumption 2 Only int variables can do boolean operations.
Assumption 3 Only int and float variables can do arithmetic operations.
Assumption 4 No nested function definitions.
Assumption 5 Field names in struct definitions are unique (in any scope), i.e., the names of struct fields, and variables never overlap.
Assumption 6 There is only the global scope, i.e., all variable names are unique.
Assumption 7 Using named equivalence to determine whether two struct types are equivalent.

Additionally, we also **modified (or revoked) assumption 5, 6 and 7**, that is:

Modified Assumption 5 Field names in struct definitions and function definitions are not unique, i.e., the names of struct fields, functions, and variables may overlap.
Modified Assumption 6 Variable names may not unique; variables defined in the outer scope will be shadowed by the inner variables with the same identifier.
Modified Assumption 7 Using structural equivalence to determine whether two struct types are equivalent; struct variables that share the same structure of fields will be seen as same type.

For further regularization, we also added other assumptions:

Assumption 8 In nested struct definition, all the inner struct definitions will be discarded, and do not conflict with outer struct definitions, but the fields in the inner struct definition will still be legal for struct variables. E.g., the following code in our extra test case 6:

```
struct A{struct B{int b;}a;}aa;
struct B{int c;}cc;
int main(){
    aa.a = cc;
    return 0;
}
```

is OK, and the semantic analyzer should not output any errors.
Assumption 9 Two array variables will be seen as structural equivalent if they share the same type and dimension, **even if their sizes are different**.

Semantic Errors

As described in the requirement, we have implemented the detection of all the following

errors:

Type 1 A variable is used without a definition.
Type 2 A function is invoked without a definition.
Type 3 A variable is redefined in the same scope.
Type 4 A function is redefined (in the global scope, since we don't have nested functions).
Type 5 Unmatching types appear at both sides of the assignment operator (=).
Type 6 Rvalue appears on the left-hand side of the assignment operator.
Type 7 Unmatching operands, such as adding an integer to a structure variable.
Type 8 A function's return value type mismatches the declared type.
Type 9 A function's arguments mismatch the declared parameters (either types or numbers, or both).
Type 10 Applying indexing operator ([…]) on non-array type variables.
Type 11 Applying function invocation operator (foo(…)) on non-function names.
Type 12 Array indexing with a non-integer type expression.
Type 13 Accessing members of a non-structure variable (i.e., misuse the dot operator).
Type 14 Accessing an undefined structure member.
Type 15 Redefine the same structure type.

Additionally, we also added another semantic error:

Type 16 Using an undefined structure type.

Furthermore, for regularization, we have some additional assumptions to these errors:

Assumption 10 The type of an undefined variable, undefined function, undefined structure member, expressions with unmatched variables, **or an indexed non-array type variable (e.g., `int a; a[1]...`)** will be seen as NULL. Assigning NULL to a variable or assining value to a Null-typed variable **will not** trigger error type 5, and letting NULL-type variables do operations **will not** trigger error type 7, but passing NULL-type variables to a function **will** trigger error type 9.
Assumption 11 A redefined variable or function will be descarded. E.g., if **a** is defined as **`int`**, then redefined as **`float`**, in the following statements, the type of **a** will still be **`int`**.
Assumption 12 Constant values (literals), expressions and function invocations are regarded as rvalue.
Assumption 13 Array indexing with a non-integer type expression will **not** result in NULL type. Which means, the following code:

```
int a[2];float b[2];
int foo(int v){return v;}
int main(){
    foo(a[b[0]]);
    return 0;
}
```

will trigger error type 7, and **not** trigger error type 9.

Implementation

Symbol Table

We use **treap** to support insert and lookup operations, use **stack** and **orthogonal list** to support scope checking, use **multi-level linked list** to store the members of a struct, and use

tree hash for structural equivalence checking.

Treap is a simple implementation of balanced binary search tree, which randomly assigns a key value to each node, and remains balanced by modifying the structure of the tree to maintain the heap structure of all the key values.

The use of stack and orthogonal list is already described in the requirement document, and our implementation is exactly the same as described in the requirement document.

Our definition of multi-level linked list is shown below:

```
struct Type{
    char* type_name;//the name of the type
    char isStruct;//s -> structure, v-> var, f->function
    unsigned long long hash;//type hash
    struct Var* contain;
};

struct Var{
    char* name;
    int dim; // dimension of the array, 0 if it is a variable
    struct Type* type;
    struct Var *next;
};
```

In **Type**, **type_name** is the name of the type, **isStruct** shows whether the type is for a struct, a variable or a function, **hash** is the hash value for structural equivalence checking, and **contain** is its members. In **Var**, **name** is the name of the variable, **dim** is the dimension of the array variable (**dim == 0** if the variable is not an array), **type** indicates the type of the variable, **next** is the next member if the **Var** is representing a field in a struct.

For using tree hash for structural equivalence checking, our process of calculating hash value for a type is:

1. Define $base_1 = 998244353, base_2 = 13331, base_3 = 19260817, modulus = 1610612741$, all are big prime numbers.
2. If the type is primitive, assign 2 for **int**, 3 for **char**, or 5 for **float**.
3. If the type is struct, calculate the hash value of each of its members.
    1) If the member is not an array, simply pass the original hash value of its type.
    2) If the member is an array, its hash value should be $(original\_hash\_value \times base_3^{dimension}) \bmod modulus$.
4. Store the hash values of the members in the array $children\_hash$, and sort it in ascending order.
5. Suppose there are $length$ members, the hash value of the struct type is $(\sum_{i=0}^{length-1} children_{hash_i} \times base_1^{length-1-i}) \times base_2 \bmod modulus$.

We intentionally ignored the names and order of the members, because we think that judging two struct types as structural inequivalent simply because the order of names of members are different is **unnatural**.

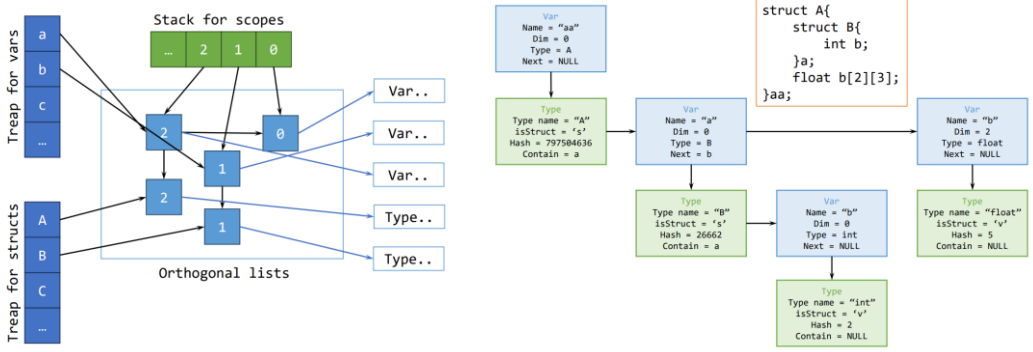
Our code for calculating hash value is shown below:

```
unsigned long long get_hash(Type* typeptr){
    unsigned long long base1 = 998244353, base2 = 13331, base3 = 19260817, mod = 1610612741,
hashval = 0;
    if(strcmp(typeptr->type_name, "int") == 0){
```

```
        hashval = 2;
    }
    else if(strcmp(typeptr->type_name, "char") == 0){
        hashval = 3;
    }
    else if(strcmp(typeptr->type_name, "float") == 0){
        hashval = 5;
    }
    else{
        Var* tmp = typeptr->contain;
        int length = 0;
        while(tmp != NULL){
            length ++;
            tmp = tmp->next;
        }
        unsigned long long* children_hash = (unsigned long long*)malloc(sizeof(unsigned long
long) * length);
        tmp = typeptr->contain;

        for(int i = 0; i < length; i ++){
            children_hash[i] = tmp->type->hash;
            if(children_hash[i] == 0){
                children_hash[i] = tmp->type->hash = get_hash(tmp->type);
            }
            int dim = tmp->dim;
            while(dim){
                unsigned long long tmpbase = base3;
                if(dim & 1){
                    children_hash[i] = children_hash[i] * tmpbase % mod;
                }
                tmpbase *= tmpbase;
                dim >>= 1;
            }
            tmp = tmp->next;
        }
        qsort(children_hash, length, sizeof(unsigned long long), cmp);
        for(int i = 0; i < length; i ++){
            hashval = (hashval * base1 + children_hash[i]) % mod;
        }
        free(children_hash);
        return hashval * base2 % mod;
    }
    return hashval;
}
```

In summary, our implementation can be illustrated by the following diagrams (They illustrate two unrelated examples):



Extra Test Cases

In out extra test cases, **test case 1** corresponds to **modified assumption 6**, **case 2** corresponds to **modified assumption 5**, **case 3** corresponds to **modified assumption 7** and **assumption 9**, **case 4** violates **error type 3** and corresponds to **assumption 12**, **case 5** violates **error type 3** and corresponds to **assumption 11**, **case 6** corresponds to **assumption 8**, and **case 7** violates **error type 16** and corresponds to **assumption 8**.