HYU-ELE4029, Compilers 3. Semantics

컴퓨터소프트웨어학부

2018008013 김영중

1. Compilation method and evnrionment

   - Env: Windows 10 WSL – Ubuntu 18.04
   - GNU Make 4.1. gcc 7.4.0
   - GNU Bison 3.0.4, flex 2.6.4
   - Make 명령어를 통해 cminus 실행파일 생성

2. Symbol table 변경점

   - Scope

C-Minus에 들어오면서 curly-brace로 이뤄진 compound statement에 variable scope 문법이 추가되었다. 이에 따라 global, function, nested compound statement에 대해 각각의 symbol table을 구성하여, 대상의 scope부터 상위 scope를 따라 가며 symbol table에서 ID search가 이루어져야 한다. 이에 따라 독립된 symbol buckets를 가지고, sibling과 child의 tree구조를 띄는 ScopeList 객체를 설계하였다.

```
typedef struct ScopeListRec
   { char * name;
     BucketList bucket[HASHSIZE];
     struct ScopeListRec * parent;  // parent node
     struct ScopeListRec * child;   // first child
     struct ScopeListRec * next;    // linked list, siblings.
   } * ScopeList;
```

[Code 1.] ScopeList declaration

ScopeList는 우선 상위 scope에 대한 접근이 가능해야 하기 때문에 parent -> child, child-> parent의 double linked-list의 구조를 취한다. 이후 sibling의 검색은 next로, 최초 자식으로의 이동은 child로 이뤄진다. 이 때 child의 sibling은 모두 같은 parent를 가진다.

이후 global scope를 initialize 하는 `global_init`, scope의 이름을 가지고 ScopeList 객체를 검색하는 `scope_find`, scope 내의 symbol buckets에서 주어진 symbol name을 검색하는 `scope_search`, 특정한 scope에 하위 scope를 추가하는 `scope_insert` 함수를 구현하였다.

   - Symbol

이 중 `scope_find` 함수는 ScopeList 객체를 반환하며, symbol의 검색, 추가, 삽입은 모두 이

ScopeList를 인자로 받아 활용한다. 이 두 과정을 수행하는 매크로 함수를 두어 편의성을 높였고, `st_lookup`은 scope name과 symbol name을 받아 ScopeList와 BucketList 객체를 반환, `st_lookup_excluding_parent`는 상위 scope에 대한 검색 없이 주어진 scope에서만 symbol을 검색한다.

`st_insert`와 `st_appendline`, `st_appendfn`은 Bucket을 조작하는 함수로, `st_insert`는 symbol table에 symbol을 추가한 후 BucketList를 반환, 나머지 두 함수는 이 BucketList를 기반으로 line number를 추가하거나 함수 정보를 추가하는 역할을 한다.

- Return scopelist, bucketlist

이렇게 Bucket과 Scope의 pointer를 header에 공개하고, 정보 은닉 없이 직접 함수에서 반환하여 조작하는 이유는 문자열 기반의 parameter로 인해 매 함수 호출마다 scope와 bucket을 검색하는 중복 과정을 제거하기 위함이 크다.

기존의 구조는 scope가 없었기 때문에 고정된 global symbol table에서 모든 symbol을 검색하고, 이 과정을 hashtable로 두었기 때문에 검색에 대한 overhead가 크지 않았지만, scope가 추가된 이후로는 tree를 모두 순회하며 scope를 검색하고, 해당 scope부터 global scope까지 parent를 따라 이동하며 모든 hashtable에 대해 symbol name을 질의해야 한다.

이 모든 과정을 symbol의 검색, 추가, 수정 과정에 수행해야 한다면 이에 대한 overhead는 단순 hashtable 검색 시간의 linear growth가 아닌 tree search 시간에 대한 추가적인 고려가 필요하다. 이는 사실 정보 은닉과 안전성을 조금 덜 추구하는 대신, bucket과 scope의 검색 결과를 직접 사용자에게 넘김으로 재사용성을 높여 해결할 수 있다.

- Function Infos

기존의 bucketlist에 ExpType member를 추가하여 analyze에서 type checking이 가능하게 두었다. 하지만 함수의 경우에는 return type과 여러 개의 parameter type 정보를 저장하고 있다가, call operation에서 부가적인 type checking을 요구한다. 이를 해결하기 위해 BucketList에 FunctionInfo 객체를 추가하였고, ExpType에 `Function` enum을 추가하여 call operation에서 `Function` enum을 확인한 경우 FunctionInfo를 통해 정보를 확인할 수 있게 구성하였다.

```
28  typedef struct
29    { ExpType retn;
30      int numparam;
31      struct {
32        ExpType type;
33        char * name;
34      } params[MAXPARAM];
35    } FunctionInfo;
36
```

[Code 2.] Struct FunctionInfo

FunctionInfo의 경우 단순히 return type을 담는 `retn`과 parameter가 총 몇 개인지 저장하는 `numparam`, parameter 정보를 저장하는 `params`로 구성된다. 이는 `st_appendfn`에서 주어진 BucketList에 TreeNode를 기반으로 함수 정보를 수집하여 적절히 구조체를 채워넣게 된다.

- Print procedures

다음은 출력 함수의 예시이다.

```
(base) revsic@DESKTOP-HAO1THT:~/2020_ELE4029_2018008013/3_Semantic$ ./cminus samples/test.cm
C-MINUS COMPILATION: samples/test.cm

Building Symbol Table...

< Symbol table >
Variable Name Variable Type  Scope Name  Location  Line Numbers
------------- -------------  ----------  --------  ------------
main          function       global      3          13
input         function       global      0           0   14   14
output        function       global      1           0   15
gcd           function       global      2           4    7   15
u             int            gcd         0           4    6    7    7
v             int            gcd         1           4    6    7    7    7
x             int            main        0          13   14   15
y             int            main        1          13   14   15

< Function Table >
Function Name  Scope Name  Return Type  Parameter Name  Parameter Type
-------------  ----------  -----------  --------------  --------------
main           global      void                         void
input          global      int                          void
output         global      void                         int
gcd            global      int                          int
                                        u               int
                                        v               int

< Function and Global Variables >
  ID Name     ID Type    Data Type
-----------  ---------  -----------
main         function   void
input        function   int
output       function   void
gcd          function   int

< Function Parameters and Local Variables >
Scope Name  Nested Level  ID Name  Data Type
----------  ------------  -------  ---------
gcd         1             u        int
gcd         1             v        int
main        1             x        int
main        1             y        int
```

[Img 1.] Table visualization

3. Analyze 변경점

- Compund statement and anonymous scope

Compound statement에 각각 scope가 적용되기 위해서는 anonymous scope에 대한 naming이 필요하다. 이 naming은 symbol table generation 이후에 type check에서도 tree node 순회 과정에 동일한 이름이 생성될 수 있어야 하므로 randomness를 포함하지 않은 normalize된 형태여야 한다.

analyze에서는 `annon_scope_name`에서 각 line number와 동일 line 내의 몇번째 compound statement인지를 판단하여 `annon_{lineno}_{num}` 형태로 이름을 구성하였다. 이렇게 각각의 독립된 scope를 구성하면 variable의 생애주기를 compound statement를 통해 통제할 수 있게 된다.

```
2   /* 16. compound statement and annonymous scope */
3   void compound(void) {
4       int a;
5       {
6           int b;
7           output(a + b);
8       }
9       /* { int c; output(b + c); } -> undeclared variable */
10      {{int c;}{int d; output(a + d);}}
11  }
```

```
< Symbol table >
Variable Name Variable Type  Scope Name  Location  Line Numbers
------------- -------------  ----------  --------  ------------
input         function       global      0          0
output        function       global      1          0    7   10
compound      function       global      2         10
a             int            compound    0          4    7   10
b             int            annon_8_0   0          6    7
c             int            annon_10_1  0         10
d             int            annon_10_2  0         10   10
```

[Img 2.] Compound statement example

- Scope stack

Tree traverse 과정에서 적절히 scope를 tracing하기 위해서는 scope name과 location을 담은 stack을 두고, preorder에서는 함수(DeclK, FnK) 혹은 Compound statement (StmtK, CompK)가 발견되었을 때 stack에 새로운 scope를 push하고, symbol table에 scope를 추가, postorder에서는 Compound statement가 발견되었을 때 stack에서 scope를 pop하는 방식으로 작동해야 한다.

기존의 코드에서는 symbol table generation 과정에 postorder에 대한 처리 routine이 존재하지 않아, `postInsert` 함수를 추가하여 Compound statement에 대한 scope postprocessing을 지원하였다.

- Symbol table generation

Symbol Table generation 과정에서는 단순 symbol table을 생성하는 것 외에 몇 가지 type checking을 진행한다. 먼저 redeclared variable과 undeclared variable을 확인한다. Redeclared variable의 경우에는 symbol table이 생성된 이후에는 이미 해당 variable이 table 내에 있기 때문에 declaration node가 오더라도 bucketlist에 flag를 추가하지 않는다면 확인할 방법이 없다. 따라서 구현하기 쉬운 것은 symbol table generation 과정에 확인하는 것이고, undeclared variable의 경우에도 비슷한 맥락에서 동일 과정에 사전 검사를 하게 두었다.

이 외에는 array size가 0 이하인지 검사하는 루틴이 하나 있다. 일전 Parser 부분에서 array를 parsing 할 때 variable을 wrapping하여 하나의 새로운 token을 둔 것이 아닌, 단순 child[0]에 const를 기록해두는 식으로 간편화 했기 때문에 type에 int[]를 따로 두지 않았다. 따라서 BucketList에서 해당 변수가 array type인지 확인하기 위한 부가적인 변수가 필요했고, 이에 BucketList에는 size member가 존재한다.

이는 Optional 혹은 Maybe 같은 template이 없는 C에서 BucketList의 size가 array인지 single 인지 검사하기 위해 -1 이라는 constant로 single 상황을 가정했고, 따라서 size로 음수를 허용하는 경우에는 validness를 검사할 수 없게 된다. 이 상황을 대비하기 위해서는 bucket list에 declaration을 등록하기 이전에 size를 검사해야 했고, 이에 따라 symbol table generation에서 array size가 0 이하인지 검사하는 루틴을 추가하였다.

이후 symbol table에 bucket과 scope를 추가, function info나 line number를 추가하는 routine을 각각의 token에 맞게 적절히 배치하였다.

- Type check

기존의 type check는 const와 variable부터 expression에 type을 부여하기 위해 child에서 parent로의 postorder traverse를 기반으로 작동하였다. 하지만 이 경우에는 scope stacking이 불가능하기 때문에 `scopeSetting` 함수를 추가로 두어 preorder에서 scope stack의 push와 pop을 진행하도록 하였다.

`checkNode` 함수에서는 각 token의 kind에 따라 적절한 type checking을 진행한다. Param과 Var에서는 void variable에 대한 검사, Assign Expr에서는 ID <- expr의 대입만을 허용하였고, ID는 함수와 배열을 제외한 일반 single int-type variable에 대해서만 가능하게 하였다. While과 If는 condition으로 int만을 받을 수 있고, Operation expression은 (int, int) -> int의 연산만 가능하게 두었다. void 함수와 int 함수의 return type을 검사하고, 함수의 호출에는 parameter의 수와 type을 검사한다.

4. Testcase

Testcase는 기존의 sample로 사용했던 [test.cm, test2.cm, test3.cm], 가능한 모든 error case를 포함한 testcase.cm, pojrect material에 있던 testexam.cm이 존재한다. 해당 sample은 "3_Semantic/samples" directory에서 확인 가능하다.