Semantic Analysis

Sungbin Jo (2021-13630), Kijun Shin (2021-15391)

CppSnuPL implementation notes

In Phase 3, we have many choices during implementation, such as separation of semantic check logic, parsing negative integers, etc. Unlike Phase 2, some logic implementation aren't quite strightforward. We are going to explain our choice and details of current implementation here.

SwiftSnuPL implementation notes

SwiftSnuPL also completes phase 3. The SwiftSnuPL Resolver handles both symbol resolving and type checking, covering a subset of phase 2 and phase 3. The Resolver saves type and symbol information on an side table separately from the parser AST. This not only allows the parser to continue and parse the code, and report resolver errors in a batch, but also allows centralizing resolving and type checking logic. The resolver finally produces a map of AST nodes to type information and resolved Symbols.

Notable differences from the provided code

First, these are some important decision of CppSnuPL: - prohibit implicit integer/longint conversion - usage of strict constant folding

And some decisions made during implement specific semantic rules: - Array subscription is only allowed with `integer` type. `longint` type isn't allowed. - String constant declaration is only way to initialize array type. Conversely, it means that we can evaluate though some char constant's initial value is defined by subscript on string. It will be compiled successfully. - When compiler meets overflow during integer/longint constant evaluation, it will keep calculate it and gives overflowed result to `CDataInitializer`.

Semantic Rule Checklist

There are many semantic rules which can't be described in EBNF form or Regex. So these have to be managed manually. List below is what we consider:

- Type Checking
  - compatible type at binary, unary operator

    - compatible type at assignment, declaration
    - boolean expression on if, while statement
    - integer/longint range check
    - Procedure/Function
        * compatible declaration, return type
        * compatible parameter, argument type(and same number)
    - Array
        * dimension/int type check on subscript
        * size must be provided in declaration
- Constant Evaluation
- Valid Identifier
    - duplicate declaration in same scope
    - invalid(undefined) symbol
    - Module/Subroutine: identifier match
- SnuPL/2 specific issues
    - prohibit use of composite type in return and assign
    - prohibit subarray argument
    - implicit type conversion of array parameter, argument

Type Checking

To implement type checking, 4 important methods `GetType()`, `TypeCheck()`, `Match(`, `Compare()` have to be completed. In many cases, type or type's validity depends on recursive function call result. According to method's purpose, `TypeCheck()` method return `false` immediately when they meet error on child, so they don't modify error message. `GetType()` return `NULL` immediately.

Operation of `Match()` and `Compare()` on scalar type is trivial. On array, `Match()` return false only when two array `_nelem` is finite and different. On pointer, `Match()` return result of `Match()` of dereferred value. Implementation of `Compare()` isn't that important since we use `Compare()` method only in `CTypeManager::GetPointer()`/`CTypeManager::GetArray()`. The 'real' comparison happened on `Match()`.

```
bool CArrayType::Match(const CType *t) const
{
  if ((t == NULL) || !t->IsArray()) return false;

  const CArrayType *at = dynamic_cast<const CArrayType *>(t);
  assert(at != NULL);

  if (GetNElem() == OPEN || at->GetNElem() == OPEN || GetNElem() == at->GetNElem()) {
    return GetInnerType()->Match(at->GetInnerType());
  }
  return false;
}
```

```
bool CPointerType::Match(const CType *t) const
{
  if ((t == NULL) || !t->IsPointer()) return false;

  const CPointerType *pt = dynamic_cast<const CPointerType *>(t);
  assert(pt != NULL);

  if (!GetBaseType() || GetBaseType()->Match(pt->GetBaseType())) return true;

  return false;
}
```

Implementation of `TypeCheck()` method of binary, unary operator is pretty straightforward. In case of `CAstSpecialOp::TypeCheck()`, when operation is `opCast`, `opWiden`, or `opNarrow`, it will be true logically. However we don't allow implicit type conversion so make `assert(false)` to it. According to it, some given method `CanWiden()` and `CAstBinaryOp::TypeCheck()` don't need to be modified. `opAddress` is always true, and `opDeref` checks only if given type is pointer type.

```
bool CAstSpecialOp::TypeCheck(CToken *t, string *msg)
{
  const CType *type = GetOperand()->GetType();
  EOperation oper = GetOperation();

  switch (oper) {
    case opAddress:
      return true;
    case opDeref:
      if (!type->IsPointer()) {
        if (t != NULL) *t = GetToken();
        if (msg != NULL) *msg = "Cannot dereference non-pointer type.";
        return false;
      }
      return dynamic_cast<const CPointerType *>(type)->GetBaseType() != nullptr;
    case opCast:
    case opWiden:
    case opNarrow:
      assert(false);
      return true;
    default:
      assert(false);
      return false;
  }
}
```

Handle Some Problems with Composite Type

- In `CAstStatAssign::TypeCheck()`, since composite type can't be assigned, it checked like this:

```
bool CAstStatAssign::TypeCheck(CToken *t, string *msg)
{
  if (GetLHS()->TypeCheck(t, msg) && GetRHS()->TypeCheck(t, msg)) {
    if (GetLHS()->GetType()->Match(GetRHS()->GetType())) {
      if (GetLHS()->GetType()->IsArray()) {
        if (t != NULL) *t = GetToken();
        if (msg != NULL) *msg = "Array assignments not supported.";
        return false;
      }
      return true;
    } else {
      if (t != NULL) *t = GetToken();
      if (msg != NULL) *msg = "Incompatible types in assignment.";
      return false;
    }
  }
  return false;
}
```

- Subarray can't be type of argument, so it has to be checked. In parser, every array argument is wrapped by `opAddress`, so type checking has to be dereferred first, then check whether subscription is once or more occurred. It is worth notable that by subscript as many as array's dimension, then it is scalar type, not subarray. It didn't have to be processed manually since we wrapped array by type checking at parser.

```
bool CAstFunctionCall::TypeCheck(CToken *t, string *msg)
{
  const CSymProc *sym = GetSymbol();
  // function definition matches with function call
  if (sym->GetNParams() == GetNArgs()) {
    for (size_t i=0; i<GetNArgs(); ++i) {
      if (!GetArg(i)->TypeCheck(t, msg)) return false;

      if (!sym->GetParam(i)->GetDataType()->Match(GetArg(i)->GetType())) {
        if (t != NULL) *t = GetToken();
        if (msg != NULL) *msg = "Argument " + to_string(i+1) + " type mismatch.";
        return false;
      }

      // check arg is sub-array
      const CAstSpecialOp *sop = dynamic_cast<const CAstSpecialOp *>(GetArg(i));
```

```
      if (sop && sop->GetOperation() == opAddress) {
        const CAstExpression *expr = sop->GetOperand();
        if (expr) {
          const CAstArrayDesignator *arrdes = dynamic_cast<const CAstArrayDesignator *>(expr
          if (arrdes && arrdes->GetNIndices() > 0) {
            if (t != NULL) *t = GetToken();
            if (msg != NULL) *msg = "Subarrays cannot be passed.";
            return false;
          }
        }
      }
    }
    return true;
  } else {
    if (t != NULL) *t = GetToken();
    if (msg != NULL) {
      if (sym->GetNParams() > GetNArgs()) {
        *msg = "More parameters required.";
      } else {
        *msg = "Too many parameters.";
      }
    }
    return false;
  }
}
```

Semantic Analysis in Parser

We checks some of semantic analysis in parsing phase. However, it is more natural that semantic analysis is operated on 'ast.cpp', we tried to do semantic analysis on 'ast.cpp'.

List below is what parsing phase checks:

- integer/longint range check: To easily parse decimal(or somewhat else) represented int type constant, `strtoll` function is recommended. So performing range check instantly after recognize constant value is convenient.

```
errno = 0;
long long v = strtoll(t.GetValue().c_str(), NULL, 10);
if (errno != 0) SetError(t, "Invalid number.");

if (t.GetValue().back() != 'L' && v > INT_MAX) {
  SetError(t, "Overflow in integer constant.");
}
```

- return type check: Checking whether return type of function is scalar type is determined by only one expression, which can easily caught by parser.

- Module/Subroutine identifier match: It is also similar to return type check, which is easier to check during parsing.
- duplicate identifier on declaration: Dividing symbol table managing logic and checking duplicate identifier logic is seemed unnecessary. We checks it on parser include parameter declaration. In general it can be done with find symbol before adding, but exception exists on checking function/procedure parameter duplication. It's because parameters are inserted to symbol table at once, not immediately after consuming token related to them.

```
set<string> param_set;
for (CSymParam *param : params) {
  if (param_set.find(param->GetName()) != param_set.end()) {
    SetError(t, "Duplicated parameter: " + param->GetName());
  }
  sym->AddParam(param);
  param_set.insert(param->GetName());
}
```

- undefined identifier usage: In `CParser::qualident()`, consuming identifier is used in constructor of `CAstDesignator`. So this semantic checks on parser is forced.
- array declaration: On variable declaration, array type cannot be open array, and their dimension size have to be evaluated in compile time. This logic is separated into `CParser::varDeclSequence()`, `CParser::type()`.

```
// open array check
const CType *vt = varDecl(s, &ts);
if (vt->IsArray()) {
  const CArrayType *at = dynamic_cast<const CArrayType *>(vt);
  if (at->GetDataSize() == 0) {
    SetError(t, "Open array is not allowed here");
  }
}

// array parsing
while (PeekType() == tLBrack) {
  Consume(tLBrack);
  if (PeekType() != tRBrack) {
    t = Peek();
    expr = simpleexpr(s);
    init = expr->Evaluate();
    if (!init) {
      SetError(t, "size of array is not determinable at compile time.");
    }
    if (!init->IsInt()) {
      SetError(t, "size of array is not an integer type.");
    }
```

```
    nelem = init->GetIntData();
  } else {
    nelem = CArrayType::OPEN;
  }
  Consume(tRBrack);
  nelems.push_back(nelem);
}
```

- implicit type conversion of array parameter, argument: To keep consistency, implicit conversion of array to pointer of parameter, argument is located in same domain. Conversion is just simply make pointer of array type, so wrap array by `opAddress` of `CAstSpecialop` will resolve it.

```
call = new CAstFunctionCall(&t, proc);
arg = expression(s);
argt = arg->GetType();
if (argt && argt->IsArray()) {
  arg = new CAstSpecialOp(t, opAddress, arg);
}
call->AddArg(arg);
```