CSCI 355, Compiler Construction (Semantic Analyzer for Final Project)

Hans de Nivelle

06.11.2020

1 Introduction

The final project of this course consists of implementing a compiler for the language below, which we call $C\&$ (pronounce 'C flat') for the time being. In order to make it digestible, we cut the final project into 3 parts:

1. Implement a parser, using **jflex** and **CUP**.
2. Implement a semantic analyzer.
3. Implement a translation function.

We will give you an implementation of the register/stack machine that was presented in the slides, so that you can test your translation function. We have now arrived at the second part.

2 Checking Sanity of Types

For every type occurring somewhere in the program, the following needs to be checked:

- In $\textbf{array}(n, T)$, it must be the case that $n \geq 0$.
- Every identifier that is used in `type.Struct` must have a definition as struct.

In order to make these checks, complete the recursive function

```
checkwellformed( type.StructStore structdefs,
                 java.lang.String context, type.Type tp )
throws Error
```

The only purpose of variable `context` is to be able to report errors to the user. It should be called with a description of the position where `tp` occurs.

3 Checking Struct Definitions

For every struct definition, the following needs to be checked:

- Every type that is used for a field, must be well-formed. Use the function of the previous section 2 to check this.
- A struct definition cannot contain repeated fields.
- A field cannot have type **void**.

  Actually, this is a matter of taste. The **void** is type containing a single constant **trivial**, which carries no information and can be stored in zero space. We follow the usual convention that **void** should not be used, even when it is wrong from theoretical view point.

In order to perform these checks, complete the function

```
public static void
checkwellformed( type.StructStore structdefs,
                 java.lang.String structname, type.Field[] struct )
throws Error
```

Next we need to check that struct definitions are non-circular. Unfortunately this is bit harder: For example, the following combination is not well-formed:

```
structdef wronglist = ( first : double, rest : wronglist2 );
structdef wronglist2 = ( second : double, rest : wronglist );
```

If either of the `rest` fields is replaced by `pointer( ... )`, everything will be fine. In order to check non circularity, complete the functions

```
public static void
checknotcircular( type.StructStore structdefs,
                  java.util.Set< java.lang.String > visitedset,
                  java.util.Deque< java.lang.String > visitedstack,
                  java.lang.String structname, type.Field[] fields )
throws Error


public static void
checknotcircular( type.StructStore structdefs,
                  java.util.Set< java.lang.String > visitedset,
                  java.util.Deque< java.lang.String > visitedstack,
                  type.Type tp )
throws Error
```

The first function must check that `structname` does not occur in `visitedset`. After that, it adds `structname` to `visitedset` and `visitedstack`, and calls the second version of `checknotcircular` for each type occurring in `fields`. The second version recurses through type `tp` and calls the first version when it encounters a `type.Struct`.

4 Checking Constants

As far as I see, constant definitions do not need any checks, because their types are determined by the tokenizer. If the user tries to redefine a constant, this is rejected by the parser, so there is no need to check this any more.

5 Checking Function Headers

For every declared function, one needs to check the following:

- The return type must be well-formed. (This time **void** is allowed.)
- The types of the parameters must be well-formed, not **void**, and the names of the parameters must be different.

It is similar to checking **struct** definitions. Most of the work can be done by calling `checkwellformed` that you wrote already in Section 2. Complete the function

```
public static void
checkFunctionHeader( type.StructStore structdefs,
                     java.lang.String funcname,
                     type.Field[] parameters,
                     type.Type returntype )
throws Error
```

6 Checking Statements

Statements are checked by recursively walking through them and checking all expressions on the way. In addition, some other checks have to be made. For example, the types of declared variables must be well formed, and conditions of ifs and whiles must be convertible to **boolean**. During the check, expressions are destructively replaced by their checked versions. It was a lot of work to implement it, and luckily you don't have to write it. You should still look at the code and understand it. It done by the function `checkStatement`. Whenever a compound block is entered, the size of `localvars` is remembered, and it is restored when the block is exited.

7 Checking Expressions

Expressions are checked by the function `checkExpr( )` in class `FunctionChecker`. Let's start with a general remark: We defined two functions `checkStatement` and `checkExpr`. The fact that we include the type in the function name is always caused by failure of static typing. Instead we could have chosen to define different types of ast. This is not easy, because the recursive relations are complicated. When we designed the `ast.Tree` class, we didn't understand them yet. In principle, one could go back and redesign the `ast.Type` class. Function `checkExpr( )` must never modify the `ast.Tree` it is checking. Instead it always construct a new, checked copy. In order to keep the task manageable, you don't have to write the complete `checkExpr` function. You have to complete the following functions:

```
ast.Tree checkUnary( ast.Tree appl, java.lang.String unary,
                     ast.Tree sub ) throws Error
```

```
ast.Tree checkBinary( ast.Tree appl, java.lang.String binary,
                      ast.Tree sub1, ast.Tree sub2 ) throws Error
```

```
ast.Tree checkTernary( ast.Tree appl, java.lang.String ternary,
                       ast.Tree sub1, ast.Tree sub2, ast.Tree sub3 )
throws Error
```

When the functions are called, the subtrees are already checked, so you don't have to do that any more.

8 Dealing with Variables

You do not have to implement anything that deals with variables, but you still have to understand how it works, because it is fundamental, and it is easy to do it wrong. Local variables are introduced and removed in stack like fashion. This does not only apply to run time, but also to the compiler:

```
i: integer;
j: integer;
begin
   i : double;
   print i  // the double.
end
begin
   i : complex;
   i.re = 0.0; i.im = 0.0
```

```
end
print i;  // the integer.
```

The data structure that stores variable must remember for each declared variable its type, its size (follows from the type), and the current positition on the stack. The current position is the sum of the sizes of the variables that were declared after it, combined with possible alignment requirements. Stack positions are not important for semantic analysis, but they are important for intermediate code generation.

The data structure for variables needs two containers, a hashmap for quick lookup and a stack for scope management. The hashmap maps identifiers to stacks of triples consisting of the type, the relative offset, and the size.

In our implementation, variable management is done by `semantics.VarStore`. In order to process the code example above, it must be called as follows:

```
local. push( "i", structdefs, new Integer());
local. push( "j", structdefs, new Integer());

int n1 = local. nrvars();
local. push( "i", structdefs, new Double());
local. gettype( "i" );   // Will be Double().
local. restore( n1 );

int n2 = local. nrvars();
local. push( "i", structdefs, new Struct( "complex" ));
local. gettype( "i" );   // Will be Struct( "complex" );
local. restore( n2 );

local. gettype( "i" );   // Will be Integer().
```

Method `push` uses the structdefs to compute the size of the variable being added.

9 Differences with Slides

There are some differences between the implementation and the slides:

- In the slides, binary operations **op**(t_1, t_2) are handled as follows: First determine the types T_1, T_2 of t_1, t_2, and then go through a list of possible types U such that **penalty**(T_1, U) + **penalty**(T_2, U) is minimal. In all cases that we encountered, this is more complicated than necessary.

  It turns out sufficient to compare **penalty**(T_1, T_2) with **penalty**(T_2, T_1) to find U. We created a function

  ```
  type.Type bestcommontype( Type.type tp1, Type.type tp2 )
  ```

that does this. It returns `Void` when no conversion in either direction is possible.

This works for most cases, except for adding an integer to a pointer, and subtracting two integers, which you will have to implement separately in `checkBinary`.

- The implicit conversion from **array/lval** to **pointer/rval** is not correct in the slides, because it is applied only on variables. In reality, it also has to be applied on some subexpressions, for example in

  ```
  p : array(10,array(20,integer))
  p[1][2]
  ```

  The expression `p[1]` has type `array(20,integer)/lval` which must be converted to `pointer(integer)/rval`.

  In order to deal with this, I created a function `ast.Tree array2pointer( ast.Tree t )`, which will apply the transformation when needed. You should include this function everywhere, where possibly an **lvalue array** can be produced. As far as I see, these are variables, field selection, and the `*` operator. Don't worry about calling it too often, because the function changes `t` only when necessary.

- The slides were (intentionally) vague about this topic, but we will have to allow some conversions between pointer types, since otherwise you cannot write expressions of form `p != null`. Converting **pointer**(**void**) to any other pointer type has a penalty of 1. Other pointer conversions are not possible.

  In *C*, the null pointer `NULL` was defined by a macro, hence the compiler sees it as 0. Therefore, *C* has to allow conversion from **integer** to pointer types.

10 Warnings and Remarks

- Due to a dumb rule in Java, I had to move `Program` to a new package `cflat`. The class is now called `cflat.Program`. You will need to adopt `Main.java` and `grammar.cup` accordingly.
- Due to the same dumb rule, I also moved the class `SemanticError` to package `cflat`. It is now called `cflat.Error`.
- *C* does not operate on **char**. This means that **char** is converted to **int** implicitly. We are ignoring this.
- It turned out impossible to return from a **void** function, due to absence of the grammar rule Stat $\rightarrow$ **return**, so you will have to add it. (I like the idea of returning **trivial** more, but let's stick with conventions.)

11 Examples

```
structdef list = ( d: double, next: pointer( list ));
structdef complex = ( re: double, im: double );
constant pi = 3.141592653589793;


function length( p : pointer( list ) ) : integer
begin
   len : integer;
   len = 1;
      /* We don’t have syntax for immediate initialization */
      /* of form integer len = 0;                          */

   while p != null do
   begin
      ++ len;
      p = p -> next
   end;
   return len
end

function fact( n : integer ) : double
begin
   res : double;
   res = 1;
   while n != 0 do
   begin
      res = res * n;
      -- n
   end;
   return res
end

function complex( re : double, im : double ) : complex
begin
   res : complex;
   res. re = re;
   res. im = im;
   return res
end

function complexsum( c1 : complex, c2 : complex ) : complex
   return complex( c1. re + c2. re, c1. im + c2. im )

function strcpy( p : pointer( char ), q : pointer( char )) : void
```

```
begin
   while *p != '\0' do
      *q ++ = *p ++
end

function sinegrad( x : double ) : double
begin
   x = x * pi / 180;
   n : integer; n = 1;
   sum = 0;
   while sum + term != sum do
   begin
      sum = sum + term;
      term = term * x * x / ( n + 1 ) / ( n + 2 );
      n = n + 2
   end;
   return sum
end
```