7.4 The C-Minor Type System

The C-Minor type system is safe, static, and explicit. As a result, it is fairly compact to describe, and straightforward to implement, and will eliminate a large number of programming errors. However, it may be more strict than some languages (like C), so there are going to be a large number of errors that we must detect.

Here are the rules that we are going to enforce:

• A value may only be assigned to a variable of the same type.

• A function parameter may only accept a value of the same type.

• The type of a return statement must match the function return type.

• All binary operators must have the same type on the left and right hand sides.

• The equality operators !=, == may be applied to any type except void, array, or function and always return boolean.

• The comparison operators < <= >= > may only be applied to integer values and always return boolean.

• The boolean operators ! && || may only be applied to boolean values and always return boolean.

• The arithmetic operators + - * / % ^ ++ -- may only be applied to integer values and always return integer.
7.5 Implementing Type Checking

Before checking expressions, we need some helper functions for checking and manipulating type structures. Here is pseudo-code for checking equality, copying, and deleting types:

```c
int type_equals( struct type *a, struct type *b )
{
    if( a->kind == b->kind ) {
        if( a and b are atomic types ){
            Return true;
        } else if ( both are array ) {
            Return true if subtype is recursively equal
        } else if ( both are function ) {
            Return true if both subtype and params
            are recursively equal
        } else {
            return false;
        }
    }
    return false;
}
```

```c
struct type * type_copy( struct type *t )
{
    Return a duplicate copy of t, making sure
    to duplicate subtype and params recursively.
}
```

```c
void type_delete( struct type *t )
{
    Free all the elements of t recursively.
}
```

Next, we construct a function `expr_typecheck` that will compute the proper type of an expression, and return it. To simplify our code, we assert that `expr_typecheck`, if called on a non-null `expr`, will always return a newly-allocated `type` structure. If the expression contains an invalid combination of types, then `expr_typecheck` will print out an error, but return a valid type, so that the compiler can continue on and find as many errors as possible.

The general approach is to perform a recursive, post-order traversal of the expression tree. At the leaves of the tree, the type of the node simply corresponds to the kind of the expression node: an integer literal has integer type, a string literal has string type, and so on. If we encounter a variable name, the type can be determined by following the `symbol` pointer.
to the symbol structure, which contains the type. This type is copied and returned to the parent node.

For interior nodes of the expression tree, we must compare the type of the left and right subtrees, and determine if they are compatible with the rules indicated in Section 7.4. If not, we emit an error message and increment a global error counter. Either way, we return the appropriate type for the operator. The types of the left and right branches are no longer needed and can be deleted before returning.

Here is the basic code structure:

```c
struct expr * expr_typecheck( struct expr *e )
{
    if(!e) return;

    struct type *lt = expr_typecheck(e->left);
    struct type *rt = expr_typecheck(e->right);

    struct type *result;

    switch(e->kind) {
        case EXPR_INTEGER_LITERAL:
            result = type_create(TYPE_INTEGER,0,0);
            break;
        case EXPR_STRING_LITERAL:
            result = type_create(TYPE_STRING,0,0);
            break;
        /* more cases here */
    }

    type_delete(lt);
    type_delete(rt);
}
```

DRAFT November 7, 2016
Let’s consider the cases for a few operators in detail. Arithmetic operators can only be applied to integers, and always return an integer type:

```c
case EXPR_ADD:
    if( lt->kind != TYPE_INTEGER || rt->kind!=TYPE_INTEGER ) {
        /* display an error */
    }
    result = type_create(TYPE_INTEGER,0,0);
    break;
```

Comparison operators can be applied to any type, as long as the types are equal on both sides. Comparisons always return boolean.

```c
case EXPR_LT:
    if(!type_equals(lt,rt)) {
        /* display an error */
    }
    result = type_create(TYPE_BOOLEAN,0,0);
    break;
```

An array dereference like `a[i]` requires that `a` be an array, `i` be an integer, and returns the subtype of the array:

```c
case EXPR_DEREF:
    if(lt->kind==TYPE_ARRAY) {
        if(rt->kind!=TYPE_INTEGER) {
            /* error: index not an integer */
        }
        result = type_copy(lt->subtype);
    } else {
        /* error: not an array */
        /* but we need to return a valid type */
        result = type_copy(lt);
    }
    break;
```

**Exercise 9.** Complete the implementation of `expr_typecheck` so that it checks and returns the type of all kinds of expressions.

Most of the hard work in typechecking is done in `expr_typecheck`, but we still need to implement typechecking on declarations, statements, and the other elements of the AST. `decl_typecheck`, `stmt_typecheck` and the other typechecking methods simply traverse the AST, compute the type of expressions, and then check them against declarations and other constraints as needed.

For example, `decl_typecheck` simply confirms that variable declarations match their initializers and otherwise typechecks the body of function declarations:
### 7.5. Implementing Type Checking

```c
void decl_typecheck( struct decl *d )
{
    if( d->value ) {
        t = expr_typecheck(d->value);
        if(!type_equals(t,d->symbol->type)) {
            /* display an error */
        }
    }

    if(d->code) {
        stmt_typecheck(d->code);
    }
}
```

Statements must be typechecked by evaluating each of their components, and then verifying that types match where needed. After the type is examined, it is no longer needed and may be deleted. For example, if-then statements require that the control expression have boolean type:

```c
void stmt_typecheck( struct stmt *s )
{
    struct type *t;

    switch(s->kind) {
        case STMT_EXPR:
            t = expr_typecheck(s->expr);
            type_delete(t);
            break;
        case STMT_IF_THEN:
            t = expr_typecheck(s->expr);
            if(t->kind!=TYPE_BOOLEAN) {
                /* display an error */
            }
            type_delete(t);
            stmt_typecheck(s->body);
            stmt_typecheck(s->else_body);
            break;
        /* more cases here */
    }
    /* more cases here */

    } /* end of switch */
}
```

**Exercise 10.** Complete the implementation of `stmt_typecheck` by enforcing the constraints particularly to each kind of statement.
7.6 Error Messages

Compilers in general are notorious for displaying terrible error messages. Fortunately, we have developed enough code structure that it is straightforward to display an informative error message that explains exactly what types were discovered, and what the problem is.

For example, this bit of C-Minor code has a mess of type problems:

```c
s: string = "hello";
b: boolean = false;
i: integer = s + (b<5);
```

Most compilers would emit an unhelpful message like this:

```plaintext
error: type compatibility in expression
```

But, your project compiler can very easily have much more detailed error messages like this:

```plaintext
error: cannot compare a boolean (b) to an integer (5)
error: cannot add a boolean (b<5) to a string (s)
```

It’s just a matter of taking some care in printing out each of the expressions and types involved when a problem is found:

```c
printf("error: cannot add a ");
type_print(lt);
printf(" (\"));
expr_print(e->left);
printf(") to a ");
type_print(rt);
printf(" (\"));
expr_print(e->right);
printf("\n");
```

**Exercise 11.** Write a function `myprintf` that display printf-style format strings, but supports symbols like `%T` for types, `%E` for expressions, and so forth. This will make it easier to emit error messages, like this:

```c
myprintf("error: cannot add a %T (%E) to a %T (%E)\n", 
   lt,e->left,rt,e->right);
```

Consult a standard C manual and learn about the functions in the `stdarg.h` header for creating variadic functions.