MicroC Project Report

Giuseppe Muntoni

Contents

1 Overview

In this report, I will explain the structure of the microC compiler, trying to formalize the various analyses made by the frontend, and the code generation.

2 Symbols and storage

First, I explain how symbols are made, and where they are stored.

2.1 Symbol data

A symbol can be a function, a local variable, or a global variable.

A **function** has a return type and a list of function parameters made of a type and an identifier.

A **local variable** is made of a type and declaration location. Saving the location along with the type is important for a local variable for a reason that I will show in a moment.

A **global variable** is made of a type and a boolean that indicates if the variable is annotated with the extern keyword or not. An extern variable is a variable that is declared and initialized in another compilation unit and is used in the current one.

2.1.1 Types

In the abstract syntax tree, the types are recursive, because we can represent arrays of arbitrary dimension and pointers of arbitrary indirection.

During the various analyses, to not recursively check the type, which is costly, I choose to save in the symbol a *linearized type*. So, a pointer type contains the indirection as an integer, and the same applies to arrays that save the number of dimensions, and the optional size of each dimension.

Although there is the possibility of representing multi-dimensional arrays, for now only unidimensional ones are supported.

2.2 Symbol builder

The module symbol builder contains functions to build a symbol, through AST's pieces of information, and insert it into the scope passed as a parameter.

2.3 Symbol table

The symbol table is used to insert and look up symbols of each scope of the program. Is made of a pair of the upper scope and a map from identifiers to symbols.

2.4 Repository for symbol tables

To give the possibility to make more passes of analysis, and to not construct each time the symbol tables from scratch, a storage for them is needed.

For this reason, I implemented a repository with simple *CRUD* operations to create/retrieve/update scopes associated with a block in a specific location.

- The **create** function is used to create the symbol tables of each scope in the program, given the abstract syntax tree. In this phase, also checks about the declarations are made (3).
- The **read** function is used to read the symbol table associated with the location of a block.
- The **update** function takes the location of the scope and an update function that takes a symbol table and returns the updated version. The function simply reads the symbol table at the location passed as a parameter, and calls the update function on this symbol table. Finally, saves the updated scope in the repository.

3 Declarations analysis

The *Declarations_analyzer* module contains a function to check that each array, declared as a local variable has an explicit size.

In addition, there is a function that checks that each access to a variable, or call to a function happens to a declared one.

These checks are made directly during the creation of the symbol tables via the create function of the repository.

4 Type checker

The module *Type_checker*, as the name suggests, checks if a program is well-typed.

Below, I will explain the grammar of types, and a formalization with typing judgments of the type system.

Then, the typing algorithm implemented in the module follows exactly this formalization.

4.1 Grammar of types

The types can be:

- **Primitive:** Here we have void, bool, char, int, and float.
- **Compound:** Here we have arrays and pointers.

4.2 Typing rules

Program

$$\frac{s_1 : void, \dots, s_n : void}{prog([s_1, \dots, s_n]) : void}$$

As we can see, we treat void as a unit. Indeed, a program is made of declarations of functions and global variables. Each function is made of statements that are supposed to have side effects. Also, the declaration of a global variable is a side effect, because causes the allocation and initialization of a piece of memory.

Top declaration

$$\frac{f\_decl.type : primitive(t), f\_decl.body : void}{f\_decl : void} \qquad \frac{}{vardecl(t, id) : void}$$

In microC, functions are not first class, so we haven't a specific type for them.

Of course, although a declaration has a void type, in the symbol table it is saved with the information about its type to make the checks when it is used.

In addition, a function has a return type that must be a primitive type.

Block

$$\frac{}{dec(t, id) : void} \qquad \frac{s : t}{stmt(s) : void}$$

A code block is made of statements or declarations of local variables.

Statement

$$\frac{guard : bool, then : void, else : void}{if(guard, then, else) : void}$$

$$\frac{guard : bool, body : void}{while(guard, body) : void}$$

$$\frac{e : t}{expr(e) : void}$$

$$\frac{f_{curr}.type : void}{return : void} \qquad \frac{e : t, f_{curr}.type : t}{return(e) : void}$$

$$\frac{s_1 : void, \dots, s_n : void}{block([s_1, \dots, s_n]) : void}$$

In the return statement, we must check that the type of the expression carried by the return matches the return type of the current function.

Expression

$$\frac{a : t}{access(a) : t} \qquad \frac{access : t, expr : t}{assign(access, expr) : void}$$

The assignment is a side-effect because modifies a piece of memory, so has a void type, but the variable and the value assigned must have the same type.

$$\frac{access : primitive(t)}{addr(access) : t*} \qquad \frac{access : t *_1 \cdots *_n}{addr(access) : t *_1 \cdots *_{n+1}}$$

The addressing is not possible for arrays, because arrays and pointers are not interchangeable, they are different types.

$$\frac{}{\mathit{sliteral}(s) : \mathit{char}[s.\mathit{length} + 1]}$$

The string literal is treated as an array of characters, in which the last element is the terminator character.

The other literals have the obvious type.

$$\frac{e : \mathit{int}}{\mathit{unaryop}(-, e) : \mathit{int}} \qquad \frac{e : \mathit{float}}{\mathit{unaryop}(-, e) : \mathit{float}}$$

$$\frac{e : \mathit{bool}}{\mathit{unaryop}(!, e) : \mathit{bool}}$$

$$\frac{e_1 : \mathit{bool}, e_2 : \mathit{bool}}{\mathit{binop}(\mathit{and}, e_1, e_2) : \mathit{bool}}$$

The same applies to the *or*.

$$\frac{e_1 : t_1, e_2 : t_2}{\mathit{binop}(+, e_1, e_2) : \mathit{max}(t_1, t_2)}$$

Here we are simply comparing the two types t_1 and t_2.

If we consider all types, but int and floats, the domain is discrete in the sense that a type precedes only itself. For int and float, instead, we can say that *int* precedes float. So, if at least one of t_1 and t_2 is a float, the type of the expression is a float. In this way, we prevent loss of information.

Of course, both t_1 and t_2 can be only integers or floats.

The same applies to other arithmetic operations.

$$\frac{e_1 : t_1, e_2 : t_2}{\mathit{binop}(=, e_1, e_2) : \mathit{bool}}$$

Also in this case t_1 and t_2 must be an integer or a float.

The same applies to other comparison operators.

$$\frac{\rho(f) : t_1, \ldots, t_n \to t, e_1 : t_1, \ldots, e_n : t_n}{\mathit{call}(f, [e_1, \ldots, e_n]) : t}$$

Here ρ is the environment, so in our implementation the symbol table of the current scope.

Access

$$\frac{}{\mathit{accvar}(\mathit{id}) : \rho(\mathit{id})}$$

In the implementation, since the symbol table is constructed in advance for each block, we don't know if the declaration in the nearest scope is before or after this access. So, using the location encapsulated in the local variable symbol, we must check if the declaration returned by ρ is before or after the access. If it is after, then we must lookup in the upper scope and so on, until we find the right declaration, associated with this access.

For global variables, this check is not made, and indeed the location is not present in the global variable symbol. This is because global declarations are visible everywhere, before and after they are declared.

$$\frac{e : t*}{accderef(e) : t} \qquad \frac{e : t *_1 \cdots *_n}{accderef(e) : t *_1 \cdots *_{n-1}}$$

$$\frac{access : t[], expr : int}{accindex(access, expr) : t}$$

5 Return analysis

The return analysis is used to check if each possible execution path of a function, has a return statement, that ends the execution of the function.

Functions that return void can omit the return statement.

Also, the main function, if returns int, can omit the return, and we consider it as if it returns 0.

5.1 Rule system

Program Consider the list of top declarations that make a program, and filter them maintaining only the function declarations.

$$\frac{f_1 \to true, \dots, f_n \to true}{prog(f_1, \dots, f_n) \to ok()}$$

In this case, we are happy because each function has a return.

$$\frac{f_1 \to true, \dots, f_i \to false}{prog(f_1, \dots, f_n) \to no\_return(f_i.name)}$$

When we encounter a function that hasn't a return, we return the name of the function for which the return is missing.

Top declaration

$$\frac{f\_decl.body \to has\_ret}{f\_decl \to has\_ret}$$

If the body is not present, we cannot make any assumption, so we return true.

Statement

$$\frac{}{expr(e) \to false} \qquad \frac{}{return(e) \to true}$$

$$\frac{then \to has\_then\_ret, else \to has\_else\_ret}{if(guard, then, else) \to has\_then\_ret \;\; \& \;\; has\_else\_ret}$$

$$\frac{}{while(guard, body) \to false}$$

$$\frac{s_1 \to false, \ldots, s_i \to true}{block([s_1, \ldots, s_n] \to true}$$

$$\frac{s_1 \to false, \ldots, s_n \to false}{block([s_1, \ldots, s_n] \to false}$$

6 Deadcode analysis

There are two phases of the deadcode analysis: the first one is used to detect unreachable statements, and the second one is used to detect unused local variables and unused function parameters.

6.1 Unreachable code

The unreachable code analysis gives us a list of locations that correspond to unreachable statements.

The rule system is simple and is only for programs, top declarations, and statements.

Program

$$\frac{s_1 \to [loc_1, \ldots, loc_n], \ldots, s_m \to [loc_1, \ldots, loc_n]}{prog(s_1, \ldots, s_n) \to [loc_{1,1}, \ldots, loc_{i,j}, \ldots, loc_{m,n}]}$$

As the rule explains, provided the unreachable statements locations of each top declaration, we concatenate them.

Top declaration

$$\frac{f\_decl.body \to (is\_term, [loc_1, \ldots, loc_n])}{f\_decl \to [loc_1, \ldots, loc_n]} \qquad \frac{}{vardecl(type, id) \to []}$$

Here we omitted the rule of the function declaration without a body: in this case, we have an axiom, and we return the empty list.

Here, *is_term* indicates the fact that a statement is a terminator. A statement is a terminator if all subsequent statements in the current block are unreachable. In other words, if this block is reached, the code after the terminator statement, cannot be reached.

Statement

$$\frac{}{expr(e) \rightarrow (false, [])} \qquad \frac{}{return(e) \rightarrow (true, [])}$$

$$\frac{then \rightarrow (is\_then\_term, then\_locs), else \rightarrow (is\_else\_term, else\_locs)}{if(guard, then, else) \rightarrow (is\_then\_term \ \& \ is\_else\_term, then\_locs + else\_locs)}$$

The *if* is a terminator if both the then and the else are terminators. For the locations, we concatenate them.

$$\frac{body \rightarrow (is\_term, dead\_locs)}{while(guard, body) \rightarrow (false, dead\_locs)}$$

The *while* is never a terminator, also if the body is a terminator because we cannot assume that the guard is satisfied at least one time. Indeed, if the body contains a return, we cannot assume that the code after the while statement is unreachable, because if the guard is never satisfied, we can reach this code.

$$\frac{s_1 \rightarrow (false, s_1\_dead), \ldots, s_i \rightarrow (true, s_i\_dead)}{block([s_1, \ldots, s_n] \rightarrow (true, s_1\_dead + \cdots + s_i\_dead + [s_{i+1}.loc, \ldots, s_n.loc])}$$

$$\frac{s_1 \rightarrow (false, s_1\_dead), \ldots, s_n \rightarrow (false, s_n\_dead)}{block([s_1, \ldots, s_n] \rightarrow (false, s_1\_dead + \cdots + s_n\_dead)}$$

6.2 Unused variables

In this analysis, the scope is to return a set of symbols that are never used, and the location of their declaration.

Between global variables, local variables, functions, and function parameters, we consider only local variables and function parameters. This is because we cannot make assumptions on global variables and functions due to the separate compilation.

Program

$$\frac{f_1 \rightarrow D_1, \ldots, f_n \rightarrow D_n}{prog([f_1, \ldots, f_n]) \rightarrow D_1 \cup \cdots \cup D_n}$$

As in the return analysis, in this case, too, we can filter out the global variables and maintain only the functions.

Top declaration

$$\frac{f\_decl.body \rightarrow (X_f, D_f)}{f\_decl \rightarrow D_f \cup (f\_decl.params \setminus (X_f \cap f\_decl.params))}$$

Here, X_f is the set of accessed variables, that are not local variables of the body of the function. So for the function considered, we are taking the local variables unused and the parameters unused, constructed by considering all the params minus the ones accessed inside the body.

Statement

$$\frac{e \to X_e}{expr(e) \to (X_e, \emptyset)} \qquad \frac{e \to X_e}{return(e) \to (X_e, \emptyset)}$$

For each statement, we return the variables that are accessed in the current or nested blocks but are declared inside this block or one of the upper blocks.

This is because, when we analyze a block, we can access variables that are local to the block or declared in some upper block, so the information of the variables accessed but not local to this block, must be passed bottom-up to let know to a block that some variables are accessed in one of the nested blocks.

In addition to that, each block must return its local variables that are never accessed in the block itself or one of its nested blocks.

$$\frac{guard \to X_g, then \to (X_t, D_t), else \to (X_e, D_e)}{if(guard, then, else) \to (X_g \cup X_t \cup X_e, D_t \cup D_e)}$$

$$\frac{guard \to X_g, body \to (X_b, D_b)}{while(guard, body) \to (X_g \cup X_b, D_b)}$$

$$\frac{s_1 \to (X_1, D_1), \ldots, s_n \to (X_n, D_n)}{block([s_1, \ldots, s_n]) \to (X_b, D_b)}$$

$$where$$

$$X_b = \bigcup_{i=1}^{n} X_i \setminus Vars_i$$

$$D_b = D_1 \cup \ldots \cup D_n \cup (Vars \setminus (\bigcup_{i=1}^{n} X_i \cap Vars_i))$$

Here $Vars$ denotes the set of variables declared in the current block, and $Vars_i$ denotes the set of variables declared until the i_{th} statement of the current block.

Expression

$$\frac{a \to X_a}{access(a) \to X_a} \qquad \frac{a \to X_a, e \to X_e}{assign(a, e) \to X_a \cup X_e}$$

$$\frac{}{literal \to \emptyset} \qquad \frac{a \to X_a}{addr(a) \to X_a}$$

$$\frac{e \to X_e}{unaryop(\_, e) \to X_e} \qquad \frac{e_1 \to X_{e_1}, e_2 \to X_{e_2}}{binaryop(\_, e_1, e_2) \to X_{e_1} \cup X_{e_2}}$$

$$\frac{e_1 \to X_{e_1}, \ldots, e_n \to X_{e_n}}{call(\_, [e_1, \ldots, e_n]) \to X_{e_1} \cup \cdots \cup X_{e_n}}$$

Access

$$\frac{}{accvar(id) \rightarrow \{id\}}$$

$$\frac{a \rightarrow X_a, e \rightarrow X_e}{accindex(a, e) \rightarrow X_a \cup X_e} \qquad \frac{e \rightarrow X_e}{accderef(e) \rightarrow X_e}$$

7 Code generation

Regarding the generation of the **LLVM IR**, I explain a rule system that shows how to convert the ast to **LLVM bitcode**. Then, the module for the code generation implements this rule system, using the Ocaml bindings for LLVM.

Program

$$\frac{s_1 \rightarrow cl_1, \ldots, s_n \rightarrow cl_n}{Prog([s_1, \ldots, s_n]) \rightarrow cl_1 + \cdots + cl_n}$$

A program is the concatenation of the commands generated by each top-declaration.

Top declaration

$$\frac{t \rightarrow t_{llvm}}{vardec(t, id, false) \rightarrow @id \ = \ global \ t_{llvm} \ init}$$

$$\frac{t \rightarrow t_{llvm}}{vardec(t, id, true) \rightarrow @id \ = \ external \ global \ t_{llvm}}$$

An internal global variable must be initialized, with 0 for primitive types and null for pointers.

For arrays, each element is initialized with the initializer of the element's type.

$$\frac{f\_decl.t \rightarrow t_{llvm}, f\_decl.formals \rightarrow [(t_{llvm_1}, x_1), \ldots, (t_{llvm_n}, x_n)], f\_decl.body \rightarrow cl_b}{f\_decl \rightarrow \begin{array}{l} define\ t_{llvm}\ @f(t_{llvm_1} \%x_1, \ldots, t_{llvm_n} \%x_n)\{ \\ entry: \\ \qquad \%x_1.addr = alloca\ t_{llvm_1}, align\ |t_{llvm_1}| \\ \qquad store\ t_{llvm_1}\ \%x_1, t_{llvm_1} *\ \%x_1.addr, align\ |t_{llvm_1}| \\ \qquad \vdots \\ \qquad \%ret = alloca\ t_{llvm}, align\ |t_{llvm}| \\ \qquad cl_b \\ \qquad br\ label\ \%return \\ return: \\ \qquad \%n = load\ t_{llvm}, t_{llvm} *\ \%ret, align\ |t_{llvm}| \\ \qquad ret\ t_{llvm}\ \%n \end{array}}$$

If the function returns void, instead of allocating a variable for the return value and loading it in the return basic block, we branch directly to the return basic block and here we generate a *ret void* instruction.

If the function does not have a block instead of a *define*, we have a *declare* without a body.

Statement or declaration in block

$$\frac{t \rightarrow t_{llvm}}{dec(t, id) \rightarrow \%id = alloca\ t_{llvm}, align\ |t_{llvm}|}$$

$$\frac{s \rightarrow cl_s}{stmt(s) \rightarrow cl_s}$$

Statement

$$\frac{guard \rightarrow (cl_g, \%guard), then \rightarrow cl_t, else \rightarrow cl_e}{if(guard, then, else) \rightarrow \begin{array}{rl} & cl_g \\ & br\ i1\ \%guard,\ label\ \%if.then,\ label\ \%if.else \\ if.then: & \\ & cl_t \\ & br\ label\ \%if.end \\ if.else: & \\ & cl_e \\ & br\ label\ \%if.end \\ if.end: & \ldots \end{array}}$$

In the real implementation, the branch at the end of the then and the else block is generated only if there is not a branch already.

$$\frac{guard \rightarrow (cl_g, \%guard), body \rightarrow cl_b}{while(guard, body) \rightarrow \begin{array}{rl} cond: & \\ & cl_g \\ & br\ i1\ \%guard,\ label\ \%body,\ label\ \%end \\ body: & \\ & cl_b \\ & br\ label\ \%cond \\ end: & \ldots \end{array}}$$

As in the *if*, the branch at the end of the body is generated only if there is not already a branch in cl_b.

$$\frac{e \rightarrow (cl_e, \%val)}{expr(e) \rightarrow cl_e} \qquad \frac{}{return \rightarrow br\ label\ \%return}$$

$$\frac{e \rightarrow (cl_e, \%val)}{\begin{array}{rl} & cl_e \\ return(e) \rightarrow & store\ ret\_t\ \%val,\ ret\_t*\ \%ret,\ align\ |ret\_t| \\ & br\ label\ \%return \end{array}}$$

The return basic block and the ret variable are the ones generated during the function declaration.

$$\frac{s_1 \rightarrow cl_1, \dots, s_n \rightarrow cl_n}{block([s_1, \dots, s_n] \rightarrow cl_1 + \cdots + cl_n)}$$

Expression Regarding the expressions, just for brevity, I omit the rules of the literal, unary operations, and binary operations except for the *and* operation to show the short-circuiting.

$$\frac{a \rightarrow (cl_a, \%addr, \%val)}{access(a) \rightarrow (cl_a, \%val)}$$

$$\frac{a \rightarrow (cl_a, \%addr, \%old), e \rightarrow (cl_e, \%new)}{assign(a, e) \rightarrow (\%new, cl_a + cl_e + [store\ t\ \%new,\ t*\ \%addr,\ align\ |t|])}$$

$$\frac{a \rightarrow (cl_a, \%addr, \%val)}{addr(a) \rightarrow (cl_a, \%addr)}$$

$$\frac{e_1 \rightarrow (cl_1, \%v_1), \dots, e_n \rightarrow (cl_n, \%v_n)}{call(f, [e_1, \dots, e_n]) \rightarrow (\%res, cl_1 + \cdots + cl_n + [\%res = call\ t\ @f(\%v_1, \dots, \%v_n)])}$$

The expressions always return the value register associated with the expression.

In the function call rule, if the function returns void we have not a result, and a *call void* instead of a *call t*.

$$\frac{e_1 \to (cl_1, \%val_1), e_2 \to (cl_2, \%val_2)}{binaryop(and, e_1, e_2) \to \begin{aligned} current\_bb :& \\ & \vdots \\ & cl_1 \\ & br\ i1\ \%val_1, label\ \%and.rhs,\ label\%and.end \\ and.rhs :& \\ & cl_2 \\ & br\ label\ \%and.end \\ and.end :& \\ & \%res = phi\ [(0, current\_bb), (\%val_2, and.rhs)] \end{aligned}}$$

As a value, the expression returns $\%res$.

The *or* operation has the same structure but evaluates the second argument, going to $or.rhs$, only if the first one is false.

Access

$$\frac{}{accvar(id) \to ([\%n = load\ t, t * \%id, align|t|], \%id, \%n)}$$

Here, in practice, if the id accessed is an array, in addition to the load, is necessary to generate a *getelementptr* instruction, to get the address of the first element of the array and return it instead of $\%n$.

$$\frac{e \to (cl_e, \%addr)}{accderef(e) \to (cl_e + [\%val = load\ t,\ t * \ \%addr,\ align|t|], \%addr, \%val)}$$

$$\frac{a \to (cl_a, \%addr, \%val), e \to (cl_e, \%i)}{accindex(a, e) \to \begin{aligned} & cl_a \\ & cl_e \\ & \%i = sext\ i32\ \%i\ to\ i64 \\ & \%addr\_i = getelementptr\ t,\ t * \ \%addr, 0, \%i \\ & \%val\_i = load\ t,\ t * \ \%addr\_i,\ align\ |t| \end{aligned}}$$

In addition to this list of commands, we return also $\%addr\_i$ and $\%val\_i$.

In the case in which we are trying to index a pointer, which happens only for function parameters arrays, in the *getelementptr* we take $\%val$ and as the index, we use only $\%i$.

Types The Llvm types are generated through adapter functions in the *Llvm_types_adapter* module. The types are the obvious ones for each type of the language. The only interesting thing is that arrays passed as a parameter to a function have a type pointer instead of an array because the size in this case is unknown.

8 Errors management

Each module of the various semantic analyses uses the result to encapsulate either the ok result or the error generated.

Errors are types that encapsulate the context of the error, defined in the module *Semantic_errors*.

Various errors are propagated using the monadic binding.

For each possible error, there is a corresponding message.

The *Semantic_analysis* module takes the result of the analysis and if there is at least one error it raises an exception with the locations and the messages.

The code generation does not use results to manage errors, but exceptions because are unexpected errors that cannot be generated after the semantic analysis.

9 Testing and execution

Each module has unit tests to check the correctness of the code.

In addition to that, there are two shell scripts to run a complete execution test, one for correct programs and one for failing programs.

The first one is called *test_all_correct.sh* and takes the path of the C implementation of the runtime support, as an argument.

The second one is called $test\_all\_failing.sh$ and takes the path of a file in which to write the errors generated by the compiler.

They are both in the $test/script$ directory.

Regarding the execution, you can use dune to install the compiler in a chosen path, and then execute it.

You can pass to it, with the $-c$ option, as many sources as you want. With the $-rts$ option you can pass the object file of the runtime support. All files are linked together, and a unique executable is generated in the output path of choice.