Languages, Compilers and Interpreters
Università di Pisa, Department of Computer Science

MicroC Project Report
2021-02-05

Andrea Laretto (619624)

1 Introduction

This project consists in developing a working compiler for a subset of the C language, called MicroC, using the OCaml programming language, the ocamllex lexer and the Menhir parser generator, while leveraging the LLVM compiler infrastructure as code generation backend. The reference page of the lab can be found at https://github.com/lillo/compiler-course-unipi. In this project report we will present and discuss the main design and implementation choices adopted in the final project.

1.1 Language extensions

Along with completing the basic syntactic and semantic elements of the language, it was required to complete at least four additional language constructs. In this project we have implemented all the eight available language extensions, listed as follows:

- `do-while` loops;
- pre/post increment/decrement operators, i.e., `++` and `--`;
- abbreviation for assignment operators, i.e., `+=`, `-=`, `*=`, `/=` and `%=`;
- variable declaration with initialization, e.g., `int i = 0`;
- multi-dimensional arrays;
- floating point arithmetic;
- strings as in C, i.e. null-terminated arrays of characters;
- structs.

1.2 File structure

We briefly present here a list of the modules defined for the project, highlighting the additional files added along those already present in the starting project skeleton:

`scanner.mll`	Contains the ocamllex specification of the Scanner module.
`parser.mly`	Contains the Menhir specification of the Parser module, and is exposed through the `parser_engine.ml` interface.
`parser_engine.ml`	Contains the exposed interface of the Parser, leveraging the Menhir Incremental API.
`ast.ml`	Contains the definition of the polymorphically annotated AST and its pretty-printing procedures.
`position.ml`	Incapsulates the definition of a located AST and of positions, shared among all modules. This is separated from the generic AST module `ast.ml`, of which this file represents a concrete AST instance with positions as annotations.
`symbol_table.mli`	Contains the the `symbol_table.mli` interface, specifying the procedures implemented by the `Symbol_table` abstract data type.
`symbol_table.ml`	Contains the concrete implementation of the `symbol_table.mli` interface, using linked lists of hash tables.
`semant.ml`	Contains the Semant module, implementing the semantic checker and the semantic AST definition.
`codegen.ml`	Contains the Codegen module, interfacing the compiler with the LLVM compiler framework and transforming semantically-checked ASTs into the LLVM IR.
`builtins.ml`	Incapsulates the name and type information of the MicroC builtin functions, so that they are logically independent from both the Semant and the Codegen module and can be separately extended with more procedures.
`util.ml`	Contains various utilities and error managing procedures.
`microcc.ml`	Contains the main compiler command-line interface logic.
`opt_pass.ml`	Includes many optimization flags that can be optionally provided to LLVM to optimize the resulting bitcode.
`rt-support.c`	Contains the runtime support for MicroC programs, with primitives such as `print`, `getint`, `printchar`, etc.

1.3 General remarks

The code has been thoroughly commented and documented, with a special attention in describing the main procedures and the tricky design-dependent choices inside the code itself. We will now introduce the four macro-sections of the compiler (Scanner, Parser, Semant, and Codegen), and elaborate on the main design and implementation paths taken in the project.

2 Scanner

The Scanner is implemented as a standard token-based lexer, using ocamllex. The final output of the Scanner is a stream of tokens, which is later processed by the Parser. There are no major implementation details of note inside the Scanner, but we can list some relevant details regarding the language extensions and the lexing section as follows:

- We use a hash table to simplify the lexer logic and to keep track of the association between language keywords and their tokens.
- In order to provide precise indication of error locations, it was necessary to signal to the ocamllex lexer the fact that newlines have been read from the input; this can be accomplished by manually calling the `Lexing.new_line lexbuf;` procedure. This is such that the lexer can increase the newline counter, and properly keep track of the position that will be used to tag the tokens.
- Numeric literals have been implemented without considering them as possibly having a leading prefix minus; this is later implemented inside the Parser, by using the more general `MINUS` token which can negate entire expressions.
- The value of integers and floats has been obtained and is provided by the corresponding OCaml parsing functions, allowing for both standard (e.g.: `0.5`, `.5`, `2.`) and scientific notation (e.g.: `0.5e3`, `.5e-2`).
- To accommodate the additional language requirements, we needed to extend the lexer with strings, characters and floating point constants, along with the extra language keywords added in the hash map.
- String literals have been implemented by reusing the definition of character literal, thus allowing for escaped literals (e.g.: `'\n'`, `'\0'`) to appear both inside of strings and inside individual characters.

3 AST

Before presenting the Parser and Semant portions of the project, we introduce the main design choices regarding the structuring of the AST and its use throughout the project. The AST structure is introduced and contained in the `ast.ml` file.

3.1 Pretty-printing

The string representation mechanism of the AST has been rewritten from scratch, without using the Haskell-inspired `[@@deriving show]` library. The pretty-printing functions have been hand-crafted in order to customize and improve the readability of the produced AST expression, while at the same time easing the debugging of the MicroC front-end when dumping the AST and checking for misparsed expressions.

3.2 Generic AST

The main design choice adopted for the AST structure is the fact that the AST is *annotated using a generic type* which propagates the annotation type throughout the AST nodes, starting from the `Ast.program` node and going downwards. This has several advantages: we can exploit and use different annotations for different purposes, both to include additional information into the AST as well as "marking" the AST at compile-time with the fact that a certain transformation has been performed on it. The main structure of annotations is introduced as follows:

```
(* Define an annotated element as a node
   type 'n and a generic annotation 'a *)
type ('n, 'a) annotated = { node : 'n; ann : 'a }

(* Helper function to create annotated nodes *)
let annotate a v  = { ann = a; node = v }

(* Getter for annotations *)
let annotation e = e.ann

(* Getter for nodes *)
let node e = e.node

(* Helper function to preserve annotations *)
let lift_annotation f = fun e -> annotate e.ann (f e)
```

As an example, we can show how the type information is concretely used with the `expr` AST node, where the propagation of the type annotation can be clearly seen throughout the AST:

```
type 't expr = ('t expr_node, 't) annotated
and 't expr_node =
  | Null
  | Access    of 't access
  | Assign    of 't access * 't expr
  | AssignOp  of 't access * binop * 't expr
  | Addr      of 't access
  ...
  | UnaryOp   of uop * 't expr
  | BinaryOp  of binop * 't expr * 't expr
  | Call      of identifier * ('t expr) list
  | Increment of 't access * prepost * incrdecr
...
and 't access = ('t access_node, 't) annotated
and 't access_node =
  | AccVar    of identifier
...
and 't stmt =  ('t stmt_node, 't) annotated
and 't stmt_node =
  | If      of 't expr * 't stmt * 't stmt
...
type 't program = Prog of 't topdecl list
```

The two main uses of the generic AST are exemplified as the output of the Parser and the Semant modules, described in their own sections. Using the generic AST also ensures a logical separation between the information given to annotate the AST and the AST structure itself, which is defined independently of its possible uses and the internal format of the given annotation (e.g.: the node positions with the format given by ocamllex).

4 Parser

The grammar accepted by the parser closely follows the guidelines provided in the project specification, available at https://github.com/lillo/compiler-course-unipi/tree/main/microc/microc-parsing, while also taking care that the Menhir generated parser does not incur in any shift-reduce conflict due to an ambiguous grammar.

4.1 Menhir API

The Parser uses the incremental API of Menhir in order to provide a simple but precise error signalling within the parser, indicating the lexeme where the syntactic error occurs. A further extension of this project could be to provide better parsing error by capturing and manually managing the parser state transitions. This detail is in any case transparent to the main Parser logic, with the actual Menhir API mechanism used being kept separate and managed in the `parser_engine.ml` file.

4.2 Located AST

The output of the Parser module is an instance of the generic AST which uses as annotations the position of the various nodes within the file, as they are given internally by the Scanner. The annotations are introduced as follows, in a separate module `position.ml` that only defines the annotation for located AST nodes:

```
(* Annotation for Position-annotated AST *)

(* Position information given from the lexical nodes up to the AST nodes *)
type position = Lexing.position * Lexing.position

(* Dummy position used for desugaring *)
let dummy_pos = (Lexing.dummy_pos, Lexing.dummy_pos)
```

The concrete use of the annotated AST can be witnessed in the definition for the Parser starting symbol `program` inside `parser.mly`, where the generic AST is instantiated with a concrete `position` annotation for all nodes of the AST:

```
/* Starting symbol: the parser returns a
   position-annotated Ast.program. */
%start <position program> program
```

4.3 Annotating nodes

The parser uses, whenever convenient, the parsing utilities available in the standard library, e.g.: `separated_list`, `option`, while also trying to make the rules follow the grammar as closely as possible. Token aliases have also been used to give readable names to tokens and simplify the parsing rules.
The parser also mirrors the fact that AST nodes can be wrapped with annotations, as it is done in the MicroC AST specification inside `ast.ml`. The convention of using an underscore as "wrapper" to locate syntactic nodes (e.g.: `expr` and `expr_`) has been also applied in the parser, using the customly-defined `annotate` Menhir procedure whenever possible.

```
/* A helper Menhir function to directly annotate
   expressions with their position */
%inline annotate(X):
   | x=X { annotate $loc x }
/* Other useful Menhir functions to simplify parsing */
%inline parens(X):
   | x=delimited("(", X, ")") { x }
%inline parentify(X):
   | x=parens(X) { x }
   | x=X         { x }
```

4.4 Precedences

The precedences expressed within the parser specification closely follow the grammar specified, while also checking the standard C reference when introducing new previously undefined operators, such as `++`, `--`, `+=`, and the `DOT` for struct member access. In particular, the dangling else problem has been solved by specifying the following predecences in the Menhir parser:

```
%nonassoc without_else
%nonassoc ELSE
```

Since the `without_else` rule comes before the predecence for the encountered token `ELSE`, branches without else have a higher-priority than the ones with the `else` keyword. This crucial section of the parser is then implemented as follows:

```
  | "if" "(" b=expr ")" e1=stmt "else" e2=stmt
    { If(b, e1, e2) }
```

```
| "if" "(" b=expr ")" e=stmt %prec without_else
  /* Apply the if-without-else precedence and use
     an empty block as missing else statement. */
  { If(b, e, annotate dummy_pos @@ Block([])) }
```

An empty `else` branch is essentially equivalent to an empty statement, which has been canonically represented with an empty block statement.

4.5 Desugaring

The Parser has the job of producing a position-located AST that can be passed to the Semant checker and later to the Codegen module, using a suitable representation. In order to simplify the inner workings of both backend transformations, it is useful and often common to "desugar" more complex structures (called "syntactic sugar") into combinations of simpler structures, and defining the syntactic sugar in terms of these already-available constructs. We present here some language constructs that can prima-facie identified in terms of other parts, and discuss why some have been desugared and why others have not.

4.5.1 Desugaring `for`

The semantics of the `for` construct in C are completely equivalent to its counterpart implementation using the `while` statement, almost by definition. For this reason, we chose to desugar it in the parser without introducing additional useless complexity in the backend. The transformation is described as follows:

```
for ( x ; y ; z )
    b
```

```
{
   x ;          // or: empty statement
   while ( y ) // or: "true"
       {
         b ;
         z ;    // or: empty statement
       }
}
```

This syntactic transformation corresponds to the following OCaml code:

```
| "for" "(" x=option(expr) ";" y=option(expr) ";" z=option(expr) ")" b=stmt
    { let expr_to_stmt  e = annotate e.ann @@ Expr(e) in
      let stmt_in_block e = annotate e.ann @@ Stmt(e) in
```

```
let expr_in_block e = stmt_in_block (expr_to_stmt e) in
let init = Option.to_list @@ Option.map expr_in_block x in
let default_condition = annotate $loc @@ BLiteral(true) in
let cond = Option.value y ~default:default_condition in
let incr = Option.to_list @@ Option.map expr_in_block z in
  Block(init
      @ [stmt_in_block  @@
         annotate $loc  @@ While(cond,
         annotate b.ann @@ Block(stmt_in_block b :: incr))]) }
```

Note how a great part of the code complexity is due to maintaining the position annotations of the various nodes in the correct location, along with having to wrap nodes inside the correct AST constructs. We have chosen to keep the three elements of the `for` as `exprs` in order to closely adhere to the grammar specification given; using `stmtordec` for the initialization part could have also possibly been a choice.

4.5.2 Parsing `do-while`

The `do-while` construct, on the other hand, is quite hard to desugar into an equivalent `while`-based structure, due to having to manage namespace and variable declaration issues. A new AST node was added to accommodate this structure, which is later treated in the Codegen section with a similar approach used for the `while`.

4.5.3 Assignment abbreviations

A quite tempting desugaring possibility concerns the abbreviation assignments, like `+=`, `-=`, etc. The obvious desugaring consists in, for example, translating `a[i] += 3` into `a[i] = a[i] + 3`. However, this has the subtle effect of duplicating the lvalue on the left, which might contain variable increment operations that would end up being performed twice. (e.g.: `a[i++] = a[i++] + 3`). For this reason, it is kept as a new AST node and translated at Codegen time. This also saves in terms of efficiency, since obtaining the variable address can be performed only once.

4.5.4 Increments

```
...
| "++" a=lexpr          { Increment(a, Pre,  Incr) }
| "--" a=lexpr          { Increment(a, Pre,  Decr) }
| a=lexpr "++"          { Increment(a, Post, Incr) }
| a=lexpr "--"          { Increment(a, Post, Decr) }
```

For a similar reason, increments and decrements are not desugared, and are simply packaged in a suitably-defined AST nodes that includes the four possibilities. Desugaring them would be technically feasible in the pre-increment version, but the post-increment one would require generating (hygienic) temporary variables. Thus, they are simply implemented at Codegen.

4.5.5 Variable declaration with initialization

A quite tempting desugaring operation would be decomposing initialized variable declarations with the declaration and the assignment, respectively. This is for the most part a valid solution. However, the non-trivial initialization logic of C becomes clear when treating arrays (and strings), showing that such a desugaring would not entirely preserve the semantics. As a direct concrete example, consider the following desugaring case:

```
char s[6] = "hello";
```

```
char s[6];
s = "hello";
```

Clearly, the desugared case cannot be compiled in our restrained MicroC setting, where we hypothesize that arrays cannot be (re)-assigned. And in fact, this also brings up an error in the C language itself, further showing that it is not simply a desugaring but that some complex initialization semantics are here at play.
Global variable initialization obviously cannot be desugared, and it requires an optional initialization element in the AST. This is because there is simply no "executable" section to which the assignment can be declared. Further semantic checks are then required to make sure that the value assigned to the global variable is effectively a compile-time constant.

5 Symbol table

A useful structure that is used throughout both the Semant and Codegen modules is the concept of **symbol table**. Symbol tables are useful in essentially every compiler to keep associations from variables to other related structures, and capture in their logic the notion of lexical scope. In this project we implemented the symbol table abstract data type using one of the most popular and standard implementation techniques for symbol tables, a (linked) list of hash tables. The concrete implementation is given as follows:

```
(* Implement a symbol table as a chained list of hash tables,
   with each table representing a scope and with lookup searching
   and traversing up through the chain. The head of the list
   represent the current innermost scope, going outwards. *)
type 'a t = ((string, 'a) Hashtbl.t) list
```

In terms of values associated to each variable, the symbol table here given is fully polymorphic, and can be reused to associate variables with different kinds of structures in different contexts.

Logically, each hash table keeps the associations relative to a scope. The list essentially acts as a stack, where each hash table "points" to another hash table that represents the enclosing parent scope. The stack is then traversed during variable lookup, achieving a $O(n)$ time complexity. During lookup, we start from the head table, which logically corresponds with the innermost scope; if we find the variable, we stop searching. This effectively implements a form of "variable shadowing", where variables declared in inner scopes can shadow previously declared variables (while obviously not allowing two identical variables in the same scope, which causes a `DuplicateEntry` exeception to be thrown). Furthermore, using lists allows us to partially share parent hash tables (e.g.: two inner scopes sharing a common scope). Variables are always declared and added to the innermost scope using the `add_entry` function, which alters the hash table.

6 Semantic checking

Semantic checking is implemented within the Semant module: it takes as input an AST annotated with locations (where locations become useful to report semantic errors at the correct locations), while returning a semantically-checked AST. In particular, the semantically-checked AST leverages again annotations to both provide valuable type information for Codegen, and act as proof that semantic checking has been performed on a given AST. Furthermore, it allows us to encapsulate the typing logic inside the Semant module, without particular type checks inside the Codegen module. In a sense, producing and applying transformations on an explicitly-typed AST is similar to the technique of using an explicitly-typed intermediate representation employed in LLVM as well as in the Haskell Core language used in GHC.[1]

[1] See for example: Sulzmann, Martin, et al. "System F with Type Equality Coercions" (2007)

The annotation chosen for semantically-checked ASTs is a MicroC type, indicated with the `Ast.typ` node. AST nodes where no meaningful notion of type can be defined (e.g.: statements) they are simply set to `TypV` and ignored. Type annotations will be later taken advantage of in the Codegen section of the module, while also being inductively used in the Semant module itself to recursively check the semantics of the nodes being considered. The semantic information is simply expressed as follows:

```
(* The type of the semantic annotation output during semantic checking.
   At the end of semantic checking, we will return a fully explicitly
   typed AST that serves as "evidence"/proof that semantic checking
   has been performed. *)
type semantic = typ
```

6.1 Semantic symbol table

During semantic checking, however, we need to keep track of different namespaces and scopes: one for variables, one for function names and one for struct names and their variables. Even though a scoped symbol table is not strictly required for functions and structs, as they cannot be nested, they are used for both conceptual (they are namespaces after all) and convenience (we can reuse the logic so far implemented with symbol tables, such as checking for duplicates, etc).

```
(* Information for functions contained in the symbol table.
   Each function can be either builtin or declared with a fundecl. *)
type fun_info =
  | Builtin of typ * typ list
  | Fundef of position fundecl

(* Information for variables contained in the symbol table.
   Each variable is identified with its type and where it
   has been declared. *)
type var_info = position * typ

(* Information for structs contained in the symbol table.
   Each struct is a namespace of field variables. *)
type struct_info = var_info Symbol_table.t

(* The main symbol table carried around during semantic checking,
   contains the namespaces for functions, variables and struct types. *)
```

```
type sym = { fun_sym    : fun_info    Symbol_table.t
           ; var_sym    : var_info    Symbol_table.t
           ; struct_sym : struct_info Symbol_table.t
           }
```

In our specific case, the concrete "symbol table" type `sym` used in Semant keeps the information about the three mentioned namespaces. Functions can be both builtin or concretely defined, variables are simply associated with their type and positions, and structs are implemented as a namespace of field declarations. Concretely, this allows to reuse the `DuplicateException` exception defined for symbol tables for the similar case of structs.

6.2 Checking

The actual semantic checking proceeds with a somewhat standard traversal of the AST, checking types, variable declarations and other constraints. An interesting function that has proven itself to be quite useful in managing types throughout this phase is the well-known concept of type unification, implemented with the `unify_type` and `unify` procedures:

```
let rec unify_type expected provided =
  match (expected, provided) with
  (* Unify NULL with any pointer type
     and return the most specific. *)
  | (TypP t, TypP TypV) -> Some(TypP t)
  (* If an array with a certain size is required, only
     that exact size can be provided. *)
  | (TypA(t, Some(s)), TypA(t', Some(s'))) when s = s' ->
     Option.map (fun t -> TypA(t, Some(s))) (unify_type t t')
  (* However, if the required size is not specified,
     an array of any size can be provided. *)
  | (TypA(t, None), TypA(t', _)) ->
     Option.map (fun t -> TypA(t, None)) (unify_type t t')
  (* Check recursive unification for the pointer type. *)
  | (TypP a, TypP b) -> Option.map (fun t -> TypP(t)) (unify_type a b)
  (* Else, check for structural type equality. *)
  | (a, b) when a = b -> Some(a)
  (* Unification failed. *)
  | _ -> None
```

```
(* Unify a provided AST expression with the expected type,
   while setting the type annotation of the provided expression
   to the unification result. *)
let unify expected provided =
  Option.map (fun t -> {provided with ann = t}) @@
    unify_type expected provided.ann
```

These functions encapsulate the casting and polymorphism logic, especially for the case of sized/unsized arrays and the `NULL` pointer type: one of the main uses of unification is to type-check a provided AST expression against an expected type, and then set its annotation with the result of the unification. This is especially useful with `NULL` expressions so that we can statically know their expected type later in the Codegen phase thanks to the explicitly-typed LLVM IR. Another crucial use is in the conversion between sized arrays and unsized arrays, where sized arrays can always "decay" and be accepted where unsized arrays are expected, thus showing a form of polymorphism. Further extensions of the language can leverage this function to simplify treating new cases and introducing new polymorphic mechanisms.

6.3 Further validation

Aside from the semantic checks described in the project specification, further checks are required by the language extensions implemented in the project. As an example, global variables can only be initialized with compile-time constants, without being able to refer to dynamic properties of values, e.g.: obtaining variable addresses, calling functions, etc.
Furthermore, it is required both in structures and in (multi-sized) arrays for elements to possess a compile-time known size, in order to be able to calculate their size and their pointer displacement in the actual machine code. This concept is incapsulated in the following function, which checks types using the C notion of a *complete type*:

```
let rec is_complete_type t = match t with
  (* Recursively check that the array element is complete *)
  | TypA(t, Some(_)) -> is_complete_type t
  | TypA(t, None)    -> false (* Unbounded arrays have unknown size *)
  | _                -> true  (* Basic types and pointers have fixed size *)
```

As a non-trivial example, we can imagine how this interplays with the semantics of multi-dimensional arrays, where we could have for example an unsized array composed of other fixed-size arrays, but not viceversa: this induces further semantic

checks, for example `int[][3][4]` is a valid type, but `int[3][][4]` and `int[3][4][]` are not. Similarly, structs can only have complete types as fields.[2] Therefore, this underlying notion of compile-time known size is crucial in the concrete implementation and checking of multi-dimensional arrays, and it can pave the way for other more complex constructs.

7 Codegen

After receiving the explicitly-typed AST produced by the Semant module, we can finally leverage the LLVM infrastructure to generate the LLVM IR and let the framework eventually optimize it and produce machine code.

7.1 Codegen symbol table

During the Codegen phase, we need another kind of information inside symbol tables: llvalues representing functions and variables, and information on how to translate struct fields into code. For this reason, the kind of symbol table that is generated and passed around in this phase can be defined as follows:

```
(* Information kept in the codegen symbol table for structures.
   We keep a list in order to lookup the element index,
   later used in gep accesses. *)
type struct_info = L.lltype * (identifier * L.lltype) list

(* The main scoped symbol table carried around during codegen *)
type sym = { fun_sym    : L.llvalue   Symbol_table.t
           ; var_sym    : L.llvalue   Symbol_table.t
           ; struct_sym : struct_info Symbol_table.t
           }
```

Function names are mapped to their corresponding global llvalues, and variables are mapped to an llvalue representing a pointer to their allocated space. Struct names are mapped to their corresponding lltype, along with an association map that associates struct field names with their field type. The use of an association list is crucial, because it allows each field to be mapped and serialized as "index" inside the map: this index is then used as argument to the `Llvm.build_struct_gep` function

[2] For simplicity, we here ignore the C99 extension for flexible array members, i.e.: unsized array declarations at the end of structs.

and retrieve field addresses. This turns out to be the core implementation detail required for the codegen implementation of structs inside the language.

7.2 Type implementation

One of the most crucial and tricky definitions in C is the concept of unsized arrays (e.g.: `int[]`) along with how to concretely implement it. In this project we decided to effectively consider it as equivalent to a pointer, a choice that seems to be supported by other LLVM-based compiler implementations following this path, namely, the Clang compiler.[3] Inside the semantics of the C language, this mechanism also takes the term *array decay* (into pointers).
Taken from the C99 standard specification,[4] section 6.3.2.1/3:

> *Except when it is the operand of the sizeof operator or the unary* `&` *operator, or is a string literal used to initialize an array, an expression that has type "array of type" is converted to an expression with type "pointer to type" that points to the initial element of the array object and is not an lvalue.*

Our implementation choice requires us to effectively cast sized arrays to unsized ones/pointers using the LLVM GEP instruction. This functionality is implemented by the following helper function, which is connected to the fact that `Semant.unify` function annotates AST values with their *expected* type. This allows us to compare the two types of the `Expr` and `Access` nodes respectively, and cast as follows:

```
(**
  Cast the given address into a pointer, if required.
  For example, this is necessary in the case where a sized
  array is given to a function that expects an unsized
  array (or, in the future, a pointer casted from the array.)
  Else, simply obtain the value given by the address.
  @param et the expected type of the access expression
  @param at the actual   type of the access expression
  @param var the address given by the access
 *)
let unify_access et at var builder =
  (match (et, at) with
```

[3] Tested with: `clang version 7.0.0-3 ubuntu0.18.04.1`, no compilation flags
[4] http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf

```
  (* Unsized arrays are already pointers; simply return their memory *)
  | (TypA(_, None), TypA(_, None)) -> var
  (* A sized array is expected, simply return the memory address as-is *)
  | (TypA(_, Some(_)), TypA(_, Some(_))) -> var
  (* An unsized array is expected, so simply obtain the pointer to it *)
  | (TypA(_, None), TypA(_, Some(_))) ->
      L.build_gep var [| llvm_zero; llvm_zero |] "" builder
  (* Standard case: load the value from the access address *)
  | _ -> L.build_load var "" builder)
```

A similar implementation choice is applied in the specific case of passing sized arrays to functions: once again, this choice is similarly followed by Clang, and is due to the fact that functions accepting sized arrays effectively receive just a pointer to the array, as it is essentially described in the C standard. This transformation of explicitly considering pass-by-reference semantics of arrays is implemented in the Semant module with the `Semant.check_function_argument_type` function, which simply converts sized arrays into unsized ones in function arguments. As an example, the following code correctly compiles in standard C compilers despite the failing size checks:

```
void prints(char s[32]) {
    int i;
    for(i = 0; s[i] != '\0'; i++)
        putchar(s[i]);
}
int main() {
    int i;
    char s[] = "hello world!\n";
    prints(s);
    return 0;
}
```

Note that the concept described by `unify_access` is still required; for example, when passing the sized arrays of a matrix to another function expecting unsized arrays. This entire infrastructure also allows us to support multi-dimensional arrays without particular breaking changes or special cases dealt with in the code.
Finally, in the LLVM documentation[5] there seems to be a notion of "variable sized array" which appears to be related instead to runtime dynamically sized arrays (with length), and it does not seem to be useful in this simpler unsized arrays setting.

[5] https://llvm.org/docs/LangRef.html#array-type

7.3 Strings

Directly related to arrays, we mention here the type implementation of strings: after adding them as primitive lexical elements and literals in the parser, we essentially treat strings as sized arrays of `char` having length $|S| + 1$ to accommodate the null-terminating character of the literal. Strings can then be passed to functions and assigned to variables, because they undergo the same similar conversion from arrays to pointers described by the `Codegen.unify_access` procedure.

7.4 Main functions

While in the Semant modules all functions essentially just manipulate AST annotations without particular transformations, the Codegen phase treats different AST elements in fundamentally different ways. We can identify the following main Codegen procedures:

- `codegen_access`
  Each access is fundamentally related to obtaining an address, which can then be used for either writing to it or simply reading its value. This function only treats `Ast.Access` nodes, and returns an llvalue corresponding to the address that they refer to. This goes to show how a well-designed AST, divided into logically useful and distinct parts, naturally becomes simple and elegant to translate, and is ultimately one of the most crucial aspects of the project. Of note here is the final `Llvm.build_gep` operation used in the `AccIndex` constructor:

```
let rec codegen_access block_maker sym builder a =
  match a.node with
    ...
    | AccIndex(a, i) ->
      let var = codegen_access block_maker sym builder a in
      let index = codegen_expr block_maker sym builder i in
      (* Depending on the type of the reference,
         access it in different ways: *)
      let indexing =
        (match a.ann with
        (* Fixed-size array need to be dereferenced with
            an initial zero-index to get through the pointer; *)
        | TypA(_, Some(_)) -> [| llvm_zero; index |]
```

```
        (* Arrays without fixed size on the other hand are
           implemented as equivalent to pointers;
           simply dereference it with the index *)
        | TypA(_, None) -> [| index |]
        | _ -> Util.raise_codegen_error "Invalid type...") in
      L.build_gep var indexing "" builder
```

- `codegen_expr`
  Expressions simply return the llvalue associated to the result that they compute. It is precisely in this section that we might need to convert the addresses given by `codegen_access` into values by using the `unify_access` function. Here strings literals (i.e.: sized arrays of characters) receive a similar treatment, in case they are given to functions. This procedure is essentially one of the core elements of Codegen, transforming literals, assignments, binary expressions, and function calls into their LLVM intermediate representation.

- `build_binary_operator`
  Standard binary operator are here sequentially translated with their corresponding LLVM opcode. This is accomplished by using the helper functions `unary_operator` and `binary_operator`, which incapsulate all the possible opcode-operand types combinations and abstract the list of possible LLVM instructions which can be easily extended. In the special case of short-circuiting boolean operators, however, the following `build_boolean_operator` procedure is called.

- `build_boolean_operator`
  Here short-circuiting boolean operators are implemented using basic blocks, conditional jumps, and the φ-function to return a final boolean value. The requirements given by this function imply three crucial details that relate and influence the previous procedures:

  1. This function requires creating new basic blocks inside the current function to implement short-circuiting. For this reason, all the previous procedures require an additional closure as parameter to generate new basic blocks in the current function.
  2. Since we might require the binary arguments to be generated in separate blocks with separate builders, we require `build_binary_operator` to operate on closures to generate binary parameters on-demand. Standard binary operators, on the other hand, simply call directly the closures and sequence them as usual.

3. Finally, this procedure relies on and modifies the code builder given to `codegen_expr`, since here we need to create a new basic block to join the program flow back into a single flow of instructions, due to short-circuiting. For this reason, we might have to impurely modify the builder given to `codegen_expr` to make it point to another block. This is kept as transparent as possible to `codegen_expr` and `codegen_stmt`.

- `codegen_stmt`
  Each statement is translated in the usual way, and here `if`, `while`, and `do-while` are each handled by generating new basic blocks and chaining their execution. After generating the code relative to the statement AST node given, this function returns a boolean value indicating whether the code generation should continue further; this is necessary in order to avoid generating code after an early `return` statement. Similarly, a helper `add_terminator` function has been used throughout the code to ensure that no instruction can be inserted to already terminated blocks. Using a similar technique as the one used in the `build_boolean_operator`, we move the builder provided to the function using the `Llvm.position_at_end` function when creating new blocks. The `while` and `do-while` constructs are implemented in completely similar ways, just differing in picking as entry point, respectively, either the condition or the body basic block.

8 Testing

In order to test the compiler and the language extensions implemented, an additional suite of custom unit tests has been developed, along with a `test.sh` script to quickly verify after and during development that previous tests were not invalidated. The tests can be inspected to verify and explore the functionality implemented in the compiler.

9 Further work

Obviously there are many improvements and corrections that could be applied to the compiler in future extensions. For example, the Parser could be refined to provide more detailed error messages exploiting the Menhir API, while also showing the user glimpses of the code at the source location when presenting the error. Furthermore, warnings could be added to the language to notify the user; these would obviously

have to be added to the Semant module, since warnings are essentially related to checking and ensuring static properties of the program. Some examples of useful warnings, in decreasing order of importance, could be:

- Indicating that certain code might be unreachable[6] (i.e.: if it is positioned after a return statement);
- Checking that the code flow of non-void functions can always possibly reach a return statement;
- Inferring the size of strings in global declarations, as it is done in C;
- Signalling that certain variables or functions are declared but never used in the program;

As a further syntactic extension, array literals could be added to ease array initialization. Such a feature would not be particularly difficult to add since, in a way, strings can be already considered as a way to declare a restricted kind of array literal; array literals could leverage a similar initialization logic, while also requiring adding further checks for global compile-time initialization (e.g.: the array elements must be compile-time constants). Finally, using the previously described architectural choices and design, pointer arithmetic and proper casting between arrays and pointers could be implemented fairly easily.

10 Conclusion

Overall, the project was quite involved and articulated, but in the end it provided a great deal of fun and satisfaction. After settling down the main compiler design and its procedures, considering the additional language extensions revealed itself to be not particularly hard. For some extensions however, such as multi-dimensional arrays and strings, it did provide a chance to consider in greater detail some crucial aspects of the compiler, first and foremost on how to provide a clean implementation for array indexing, pointers, and initialization semantics. Finally, developing this project further pushed me to deepen and analyse the sometimes tricky semantics and details of the language being implemented, along with reflecting on some of the choices adopted by other language implementations and the language itself.

[6] Interestingly, the `-Wunreachable-code` flag seems to have been removed from gcc 4.4 onwards. https://stackoverflow.com/questions/17249934/why-does-gcc-not-warn-for-unreachable-code Clang, on the other hand, does implement this as of `clang version 7.0.0-3 ubuntu0.18.04.1`, using the `-Wunreachable-code` flag.