# Toward A Provably Correct GoLite Compiler

JASON Z.S. HU, McGill University, Canada

In this report, we discuss our effort on implementing a compiler for the GoLite language, a subset of the Go language. We introduce our analysis and our approach to the problem, such that the compiler is cleanly implemented. We point out a number of properties to have in order to argue the correctness of the type checker and the code generator.

Additional Key Words and Phrases: compilers, GoLang, GoLite, OCaml, parsing, type systems, type checking, LLVM

**ACM Reference Format:**
Jason Z.S. Hu. 2020. Toward A Provably Correct GoLite Compiler. 1, 1 (May 2020), 21 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

Compilers are often understood as "magic boxes". Programmers feed a program in the box and the box will spit out an executable which somehow does what the programmers expect. This happens so often that programmers usually forget about the fact that compilers are nothing but a normal piece of software. The software is written by other programmers and they might introduce bugs.

Though the history of compilers is as long as the one of computer science and the software engineering community has been fighting bugs for about the same time, a sound compiler is only a recent achievement. CompCert [Leroy 2006], for example, is an optimizing (almost) C99 compiler formally verified in Coq [The Coq Development Team 2020]. Thus, it is possible to develop a correct compiler, and we can do better than just "being careful".

There are many categories of bugs in a compiler. Before writing a compiler, we ought to determine what the language consists of. The final decision forms a specification of a language. We often refer to a bug as certain behaviors of the compiler which disagree with the specification. There is another kind of bugs introduced by the mismatch between the specification and programmers' expectations. In this case, the specification needs to be improved but as an implementation, the compiler is not wrongly implemented. Another possibility is that the language does "less" than what the programmers think. This happens when programmers hit the limitation of the language which might not be easily resolvable.

In this report, we discuss our effort and mindset to implement GoLite in a provably correct way. By correctness, we only consider the first kind of bugs, namely those breaking the specification. We sometimes hit the second kind of bugs, which still need to be caught by a large number of tests. That said, handling the first kind of bugs has already given us advantages. We do not run proof assistants to prove the correctness formally. Instead, we employ the purely functional programming style and some methods to organize our code so that we can maintain certain logical invariant and claim that the implementation is "obviously" correct. In this report, we discuss our experience and how this is done in OCaml.

Figure 1 shows the organization of the compiler and the involved tools in each phase.

Author's address: Jason Z.S. Hu, zhong.s.hu@mail.mcgill.ca, School of Computer Science, McGill University, 3480 University St., Montréal, Quebec, H3A 0E9, Canada.

2020. XXXX-XXXX/2020/5-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

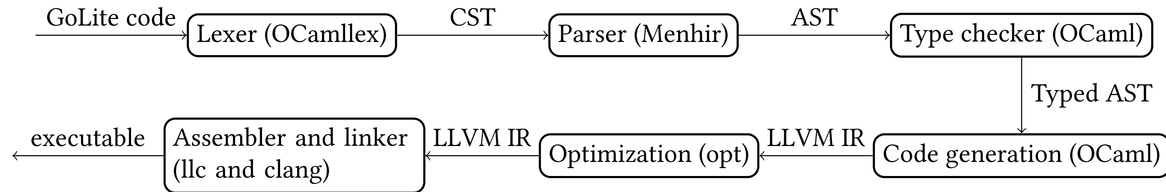


Fig. 1. Organization diagram

## 2 PARSING AND ABSTRACT SYNTAX

In the first two phases, we need to convert the input GoLite source code into an abstract syntax tree (AST) representing the essential portion of the program. We use the standard tool chain of OCaml: OCamllex and Menhir [Minsky et al. 2013, Chapter 16]. In this section, we discuss how the lexer and the parser are concretely organized.

### 2.1 Lexing and Parsing

OCamllex (the lexer) and Menhir (the parser) are tightly integrated. Unlike other tool chain, e.g. Flex and Bison [Levine 2009], the development process of OCamllex and Menhir is interleaving. We first need to determine the tokens in the parser file (`parser.mly`) and only after that we can start to write the lexer file (`lexer.mll`). Once the tokens are defined, the lexer returns a stream of tokens which is subsequently received by the parser and converted into an abstract syntax tree (AST).

OCamllex provides many very convenient functionalities. A profound one is to allow us to keep track of the locations of the characters. We later make use of the location information to generate very readable error messages. For example, in the type checker phase, we generate the following message:

```
Error: line 5 char 20-23: the expression foo has type [5]int, but the following type(s)
↪  is(are) expected: [5]float64
```

Internally, we use the following OCaml record to annotate data with such information:

```
type 'a meta = {
    start_l : int; start_c : int; end_l : int; end_c : int; data : 'a
  }
```

It is a polymorphic data type decorating a piece of data `data` with four additional integers. They record a consecutive region and correspond to the start line number of the region, the character position of the start line of the region, the last line number of the region, and the character position of the last line of the region, respectively. We also implement a number of combinators which make combining location information easy.

Another convenient functionality is that OCamllex allows lexing rules to be mutually dependent. This is very helpful when parsing the escape sequences in an interpreted string literal. The following is a piece of related OCamllex code fragment:

```
1 | '"' { let ... in ...;
2         let s = read_interpreted_string buf lexbuf in
3         ... }
4 and read_interpreted_string buf =
5   parse
6 | '"' { (* return the accumulated buffer *) }
7 | '\\' 'n'  { (* read a newline into the buffer *) }
```

```
8  | ... (* other escape sequences *)
9  | ('\\' _) as s { raise (UninterpretableEscape (coordinates lexbuf s)) }
10 | [^ '\\' '"' '\r' '\n' ] as c { (* read literal characters and ensure they are ASCII *) }
11 | ... (* more cases follow *)
```

Line 1 to line 3 is the main lexer code. It says when we encounter a double quote, we execute the code on the right and invoke `read_intepreted_string` which is define from line 4 to line 11. The interpreted string is constructed using a string buffer `buf`. In `read_intepreted_string`, when we encounter another double quote, we return the accumulated string in the buffer so far. When we encounter an escape sequence, e.g. `"\n "` on line 7, we insert a new line character into the buffer. On line 9, we throw an exception if an escape sequence is illegal. Otherwise, we just insert that character into the buffer as shown on line 10.

OCamllex also allows us to add additional parameters to the main lexer function. We use this feature to tackle the optional semicolon specified by the Go language. In the Go language, there are several occasions where a token of semicolon is automatically inserted when a new line is seen. We achieve this by passing the previous token, if any, as an additional parameter to the lexer so that it can generate different tokens when a new line is encountered.

The parser is also quite easy. It is similar to any typical parser. For each grammar production rule, there is an associated parser action which is executed once the production rule is matched. We proceed by transcribing the Go specification. Menhir detects ambiguities in the grammar and a human readable explanation can be accessed via the `--explain` flag. We resolve the ambiguities in the specification by either adding logic to the parser actions or altering the grammar rules.

## 2.2 Untyped ASTs

In our parser action, we need to generate an AST for the input program. We call this AST *untyped* in order to distinguish it from a different and *typed* AST generated by the type checker.

In a functional programming language like OCaml, it is a very common practice to represent ASTs using algebraic data types (ADTs) [Liskov and Zilles 1974] and we adopt this convention. ADTs make reasoning of the semantics of the data very clear and recursions of them are easy to reason about. This advantage subsequently contributes to the correctness arguments later in this report.

The actual design of the ASTs is quite standard. We simply strip off the unnecessary portion of the concrete syntax and the resulting ADTs are very close to the Go specification. Since we program in the purely functional style, we pay very close attention to our ADTs to ensure that their recursive functions are total. Maintaining totality of functions makes keeping track of logical invariants much easier. For untyped ASTs, we design the following ADTs:

```
type typ        (* represents types *)
type expr       (* represents expressions *)
type simp_stmt  (* represents simple statements *)
type decl       (* represents declarations (variables and types) *)
type stmt       (* represents statements *)
type stmts      (* represents sequences of statements *)
type top_decl   (* represents top level declarations *)
type program    (* represents whole programs *)
```

The concrete cases are omitted. Each **type** above remembers its own location `meta`. Notice that we separate `simp_stmt` from `stmt` in the interest of totality of functions explained above. Likewise, `decl` and `top_decl` have the same design.

| | |
|---|---|
| $x, y, z$ | **Names** |
| $\tau$ | **Types** |
| $\Gamma ::=$ | **Contexts** |
| $\cdot$ | Empty Context |
| $\Gamma, x \sim \tau$ | Context Concatenation |

| | |
|---|---|
| $x \sim \tau ::=$ | **Bindings** |
| $x : \tau$ | Variable Bindings |
| $x \div \tau$ | Constant Bindings |
| $x :: \tau$ | Type Bindings |

**Type Resolution**

A type reference is represented by a tuple $(x, \tau)$.

$$RT(\tau) \triangleq \begin{cases} \tau' & \text{if } \tau \text{ is a type reference } (x, \tau') \\ \tau & \tau \text{ is not a type reference} \end{cases}$$

**Binding Lookup**

$$\frac{\Gamma = \Gamma_1, x \sim \tau, \Gamma_2 \qquad x \notin dom(\Gamma_2)}{x \sim \tau \in \Gamma}$$

**(Partial) Typing Rules**

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{x \div \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash e : \tau \qquad RT(\Gamma, \tau) \in \{\texttt{int}, \texttt{float64}, \texttt{rune}\}}{\Gamma \vdash -e : \tau}$$

$$\frac{}{\Gamma \vdash \texttt{type}\ \ x\ \tau \Rightarrow \Gamma, x :: \tau} \qquad \frac{\Gamma \vdash e : \tau \qquad x \notin dom(\Gamma)}{\Gamma \vdash x := e \Rightarrow \Gamma, x : \tau} \qquad \frac{\Gamma \vdash \texttt{init} \Rightarrow \Gamma' \qquad \Gamma' \vdash e : \tau \qquad RT(\tau) = \texttt{bool} \qquad \Gamma' \vdash \texttt{stmt}* \Rightarrow \Gamma''}{\Gamma \vdash \texttt{if}\ \ \texttt{init}; e\ \{\texttt{stmt}*\} \Rightarrow \Gamma}$$

Fig. 2. Adjusted Syntax and Definitions

### 2.3 Weeding Processes

In Go and GoLite, there are certain grammatical requirements which are relatively complex to implement as context free grammar (CFG) rules. For example, **`continue`** cannot live outside of a loop body, and there is at most one **`default`** case in a switch statement. Thus we need more sanity checks other than the CFG itself. Luckily, we are already able to do many checks in the parser action itself, e.g. the check for **`default`** in a switch statement. The check for **`continue`** and **`break`**, on the other hand, is done after the parsing because they require understanding of the surrounding syntactical context.

When a weeding process fails, it throws a dedicated exception:

```
raise (AtMostOneDefault m)
```

which is caught in the main function to generate a proper error message (with location information, of course).

## 3 STATIC SEMANTICS

After the parsing phase, we have obtained an AST. In this section, we discuss the formal specification of the GoLite language and design its static semantics. We begin by considering the formal judgments relating the untyped expressions and statements to type information. This process helps us to understand what the functions we should implement and what are their inputs and outputs.

### 3.1 Formal Judgments

We proceed by defining a fragment of the formal syntax and judgments of GoLite in Figure 2. The fragment defines various concepts we need to use in the implementation and establish the correctness invariant of the compiler. During type checking, we need to maintain a *typing context* to understand the semantics of names.

A context is either empty or a given context concatenating another binding. In GoLite, we have three forms of bindings: variable bindings, constant bindings and type bindings. A variable binding binds a variable to a type. A constant binding is similar to a variable binding except that the binder cannot be mutated. We use constant bindings for constants like **true** and **false**. A type binding binds a name to a type and the binder behaves as a distinct type. Three kinds of bindings are syntactically distinguished as shown on the top right in Figure 2.

Since we have type bindings in our language, we sometimes need to find out the underlying types. This is done by the *type resolution* operation. Since a type binding is constructed when it is declared, the result of a type resolution should be independent of the context. Therefore, the type resolution function *RT* takes only a type as the parameter. This motivates the design in which a type reference should carry its resolved type. In our syntax, we represent a type reference as a tuple $(x, \tau)$, where $\tau$ is the resolved type.

In the formal language, we design two kinds of judgments:

$\Gamma \vdash e : \tau$ — the expression $e$ has type $\tau$ in context $\Gamma$

$\Gamma \vdash s \Rightarrow \Gamma'$ — the statement $s$ is well-formed in context $\Gamma$ returning an updated context $\Gamma'$

The judgments for expression are quite standard. The judgments for statements are more interesting. One might expect the judgments for statements to be

$$\Gamma \vdash s$$

In our judgments, $\Gamma'$ denotes the updated context. For example, in the type definition and variable definition rules in Figure 2, the updated contexts are the original contexts concatenated with a type binding and a variable binding, respectively. Consequently, when we type check an if statement, we have access to the updated context when type checking the expression $e$ and the body `stmt*`, as indicated by the premises $\Gamma' \vdash e : \tau$ and $\Gamma' \vdash \texttt{stmt*} \Rightarrow \Gamma''$. Without providing $\Gamma'$ in the judgment, $e$ must be type checked in the original $\Gamma$. This is clearly incorrect when `init` is a short declaration:

$$\frac{\Gamma \vdash \texttt{init} \qquad \Gamma \vdash e : \tau \qquad RT(\tau) = \texttt{bool} \qquad \Gamma \vdash \texttt{stmt*}}{\Gamma \vdash \texttt{if init;} e \; \{\texttt{stmt*}\}} \textsc{If-Wrong}$$

## 3.2 Typed ASTs

*3.2.1 Structure of Typed ASTs.* After analyzing the formal syntax and judgments of GoLite, we move on to design the ASTs representing typed programs. We have the following types in our typed ASTs:

```
type typ        (* represents types *)
type expr       (* represents (right) expressions *)
type ltyp       (* represents types for left expressions *)
type aexpr      (* represents addressable expressions (to be explained in more detail) *)
type lexpr      (* represents left expressions *)
type simp_stmt  (* represents simple statements *)
type 'a decl    (* represents declarations (variables and types) *)
type stmt       (* represents statements *)
type stmts      (* represents sequences of statements *)
type top_decl   (* represents top level declarations *)
type program    (* represents whole programs *)
```

We reuse the names but these types are distinct from the untyped ASTs shown in Section 2.2. Compared to our untyped ASTs, the typed ASTs, on one hand, exhibit a large similarity: we also have `typ`, `expr`, `stmt`, etc. to represent the corresponding concepts. On the other hand, we introduce some extra **type**s in the typed case: `ltyp`, `aexpr` and others. The typed counterpart of `decl` becomes polymorphic, as indicated by `'a`.

Conceptually, the typed ASTs are a semantic version of the untyped ASTs. The typed ones only represent those well-formed programs while the untyped ones represent all syntactically valid programs (regardless of being malformed or not). Indeed, in addition to location information, the typed ASTs are aware of their type information. The following functions witness this fact:

```
val get_typ  : expr  -> typ
val get_atyp : aexpr -> typ
val get_ltyp : lexpr -> ltyp
```

That is, all expressions know their own types.

THEOREM 3.1. *All typed expressions have well-formed types.*

We start to gradually build up logical arguments toward our final correctness claim.

*3.2.2 Semantic Types.* Among all concepts, the most important one is *types*. When redefining typed `typ` or *semantic* `typ`, we adjusted the structure so that the semantics are better reflected. In the untyped `typ`, we remember location information and a type reference contains no resolved type. Contrarily, in the semantic `typ`, we no longer store location information and a type reference carries its own resolved type as designed in the formal syntax in Section 3.1. Additionally, we define `void` and function types in the semantic `typ` in order to better unify the type representation. These treatments allow us to implement a function comparing the equality between semantic `typ`s via a structural recursion, which implies that we have succeeded in reflecting the semantics in the structure of semantic `typ`.

*3.2.3 Semantic Expressions.* In the untyped ASTs, we use `expr` to represent all expressions, while in the typed ASTs, we use three: `expr`, `aexpr` and `lexpr`. They correspond to three different kinds of expressions in a program. `expr` corresponds to ones used as right expressions, those producing values:

```
val x = foo[bar]
baz(x)
```

In this code fragment, `foo`, `bar`, `foo[bar]` and `x` in `baz(x)` are `expr` because they (are expected to) compute to values.

`aexpr` corresponds to addressable expressions, those with assignable locations in the memory. `aexpr` is a subset of expressions of `expr`, as proved by the following function:

```
val aexpr_to_expr : aexpr -> expr
```

THEOREM 3.2. *The set of expressions of* `aexpr` *is a subset of* `expr`.

Due to this theorem, `aexpr` can also produce a value. This fact is used in code generation. In the following code fragment, `a[x]` and `x` on the second line are represented by `aexpr` but `x` in `a[x]` is an `expr` because it evaluates to a value:

```
a[x] = 10
x = 20
```

We separate `aexpr` apart from `expr` in preparation for code generation. Otherwise, in code generation, we would have to duplicate the check that has occurred in the type checker. This separation helps us to write far less partiality in the code generation phase.

`lexpr` corresponds to left expressions. It is either an underscore on the left hand side or just an `aexpr`:

```
_ = foo(bar)
```

The _ is a `lexpr`. Notice that `lexpr` is not a subset of `expr` because _ does not produce a value (consider `(_ + 1)`). Thus, `lexpr` does not always have a `typ`, which motivates `ltyp`.

*3.2.4 Function Declaration and Name Representation.* Some consideration reveals that global variables and local variables are different. All global variables live precisely in one scope, while local variables have different scopes. This adds many details like how concretely a scope behaves in the language and these details are not easily expressible as structures of ADTs. Hence, it is of our interest to avoid making any assumption of scopes of GoLite and the target language in code generation, in order to simplify our program logic.

The idea we take is to use different data types to represent names of global and local variables. Specifically, we use string to represent global names while natural numbers to represent local names. We use the following example to demonstrate:

```
1 func foo(x, _ int) {
2     for x := true; false; {
3         if x {
4             var x string
5         } else {
6             var x float64
7         }
8     }
9 }
```

```
1 func foo(%0, _ int) { // [int; bool; string; float64]
2     for %1 := true; false; {
3         if %1 {
4             var %2 string
5         } else {
6             var %3 float64
7         }
8     }
9 }
```

On the left, we show a program in which names are intentionally heavily shadowed. We first receive an `int x` as a parameter on line 1. In the `for` loop on line 2, we declare another `x` in a new scope and use it on line 3. Finally we declare one more `x` as a `string` on line 4 and one as a `float64` on line 6. This program is not only confusing, but also adds difficulties to the implementation. One typical way to address shadowing is to rename all the variables so that they do not clash. For example, the `x`'s on both line 2 and line 3 can be renamed to `y`. Though it is a valid solution, it is not very convenient. There are two problems:

(1) We need to implement a whole set of substitution functions for all types in the ASTs.
(2) We need to be very careful to avoid clashing names which we are still not aware. For example, if line 4 is `var y string` instead, then `y` is really not a valid choice to substitute `x` for on line 2 and 3.

Instead, we take a much easier approach, shown on the right. We associate each function declaration a list of types. Each variable corresponds to a position in the list and its occurrences are replaced by the index of the position. In the program on the right, the `x` on line 1 is replaced by `%0`. It is 0 here because it is the first variable. We use `%` here to avoid making the program look like an assignment to an integer literal. Likewise, the `x`'s on line 2 and line 3 are replaced by `%1`. This approach makes it clear that the `x` on line 3 refers to the one on line 2. We replace `x` on line 4 with `%2` and the one on line 6 with `%3` even though they are not used anywhere. This approach tackles both aforementioned problems at once. We no longer need to implement substitution nor do we have to figure out a strategy to generate fresh names. This approach is not entirely innovative; it is very related to locally nameless representation [Charguéraud 2012] and the indices are similar to de Bruijn indices [de Bruijn 1972].

In this example, we have already silently made an optimization: we have omitted the blank parameter of `foo`. We could have made the associated list `[int; int; bool; string; float64]`, where the second `int` corresponds to the blank parameter of `foo`. However, we do not bother doing so because blank identifiers can never be referred to.

This approach also motivates us to distinguish two different kinds of declarations: global and local ones. In untyped ASTs, we use only one ADT `decl` to represent declarations. In typed ASTs, we use two and that is why we make `decl` polymorphic:

```
type 'a decl
type gdecl = string decl  (* global declarations *)
type ldecl = int decl     (* local declarations *)
```

$\Gamma \vdash^r e \Rightarrow e'$ type checking a right expression

$\Gamma \vdash^a e \Rightarrow e'$ type checking an addressable expression

$\Gamma \vdash^l e \Rightarrow e'$ type checking a left expression

$\Gamma \vdash \tau \Rightarrow \tau'$ type checking a type

$\Gamma \vdash^\tau s \Rightarrow (\Gamma', s'*)$ type checking a statement where $s'*$ is a sequence of statements

$$\frac{}{\Gamma \vdash^{\texttt{void}} \texttt{return} \Rightarrow (\Gamma, \texttt{return})} \qquad \frac{\Gamma \vdash^r e \Rightarrow e' \qquad e' \text{ has type } \tau}{\Gamma \vdash^\tau \texttt{return}\ e \Rightarrow (\Gamma, \texttt{return}\ e')}$$

Fig. 3. Type checking functions in judgmental form. Terms in red are untyped and Terms in blue are typed.

We specialize the generic `decl` with **`string`** for global declarations and with **`int`** for local ones.

### 3.3 Type Checking

Combining Sections 3.1 and 3.2, we can proceed to implement the type checker. The actual type checking functions are different from the judgments in Section 3.1. We made necessary adjustments so that the output includes typed ASTs.

We list the type checking functions extending formal judgments in judgmental form in Figure 3. We use colors to distinguish untyped and typed ones. Notice that $\Gamma$ is always blue because contexts keep track of semantic information which has to be typed. Terms to the left of $\Rightarrow$ are inputs and those to the right are outputs. For example, $\Gamma \vdash^r e \Rightarrow e'$ denotes a type checking function taking a context and an untyped `expr`, and returning a typed `expr`. These extensions are justified by Theorem 3.1.

Theorem 3.3. *If $\Gamma \vdash^r e \Rightarrow e'$, then $\Gamma \vdash e : \tau$ where $\tau$ is the type of $e'$.*

Similar theorems exist for other type checking functions for expressions.

In our actual implementation, $\Gamma$ needs to keep track of two contexts. One is often called *symbol table*, which maps from names to types. This is the context defined in Section 3.1. In our code, this context is implemented as a stack of maps from string to bindings. The other context is the list associated to each function discussed in Section 3.2.4. It is especially important to carefully accumulate all local variables in a function body so that the indices are correctly mapped. This also provides a reason why type checking for statements need to return a potentially updated $\Gamma'$.

The type checking function for statements $\Gamma \vdash^\tau s \Rightarrow (\Gamma', s'*)$ is worth an elaboration. First it takes a semantic `typ` $\tau$ as input. $\tau$ denotes the return type of the current function and is used during type checking **`return`**s to ensure that all expressions returned by the **`return`** statements are consistent with the declared function as denoted by the judgments. Moreover, $\Gamma \vdash^\tau s \Rightarrow (\Gamma', s'*)$ returns $s'*$ which indicates a list of statements. We translate one untyped statement to zero, one, or multiple typed statements for convenience. For example, **`type _ int`** introduces no new type binding, so when type checking this type declaration, we just ensure that **`int`** is well-formed (in this case, it is) and disregard it. As a result, it returns no statement. It is also possible to generate multiple statements:

```
var (
    x int
    y float64
)
```

This program generates two typed statements from a untyped one.

### 3.4 Return and Terminating Statements

By simply accumulating $\Gamma \vdash^{\tau} s \Rightarrow (\Gamma', s'*)$, we can convert an untyped AST for the program to a typed one. However, this typed AST is not necessarily well-formed. We must ensure:

(1) **`return`** conforms the return type of the enclosing function, and
(2) if a function has a return type, the function always ends with *terminating statements*.

The first item has been taken care of by the type checker because the return type is passed in as $\tau$. The second item is handled by recursively looking into the last statement and require it to be a terminating statement. A terminating statement in GoLite is either:

(1) a **`return`** statement,
(2) a block with the last statement terminating,
(3) a **`if`** statement with an **`else`** block and both blocks are terminating,
(4) a **`switch`** statement with a **`default`** block and all blocks are terminating and do not contain a **`break`** statement breaking the **`switch`**, or
(5) a **`for`** statement without a condition and the body does not contain a **`break`** statement breaking the loop.

This is literally transcribed in our code. After this check, we finally obtain a well-formed typed AST to be passed to the code generation phase.

## 4 CODE GENERATION AND DYNAMIC SEMANTICS

In this section, we discuss the code generation phase. In this phase, we transform a typed AST to LLVM [Lattner and Adve 2004] intermediate representation (LLVM IR). We show how we structure our program logic and maintain the logical invariant, so that the resulting translation to LLVM IR can obtain a correctness argument.

### 4.1 LLVM IR

LLVM is a common compiler backend which implements an intermediate representation (IR) and sophisticated optimizations based on this IR. LLVM is very easy to program and provides official binding APIs in OCaml. Moreover, LLVM has an architecture very similar to a physical computer: it has registers and memory which is divided into multiple segments. Nonetheless, LLVM provides many convenient abstractions so it is much easier to program LLVM than an actual machine.

*4.1.1 Types.* LLVM is strongly typed. That is, all data in LLVM are associated with types. The type system of LLVM is simple, as it contains only basic types like integers, floating points and pointers, as well as aggregate types like arrays and structs. In particular, it does not feature polymorphism. LLVM provides many instructions for converting values which can be used to implement conversions among integers and floating points, and many pointer castings.

*4.1.2 Static Single Assignment.* One of the most useful abstractions LLVM IR employs is the static single assignment form (SSA) [Rosen et al. 1988]. In LLVM IR, there is an infinite supply of registers and values are assigned to registers. Once a register is assigned a value, it can no longer be mutated, thus single assignment. This form makes keeping track of values much easier than mutable registers in actual machine architectures and enables various simple yet effective optimizations.

*4.1.3 Memory.* SSA is not a complete silver bullet because it makes mutation impossible. In order to complement the lack of mutation, LLVM adopts a memory model very similar to typical machines. For each function, there is allocated a stack frame, which is released after the function returns. Inside a function, we can use **`alloca`**

```
void arr_init()
{
        double a[20]; int i;
        for (i = 0; i < 20; i ++) a[i] = 0.0;
}
```

(a) A program initializing an array of **doubles** to 0

```
1  define void @arr_init() {
2  entry:
3    %a. = alloca [20 x double]
4    br label %valinit.pred
5
6  valinit.pred:                                      ; preds = %valinit.body, %entry
7    %valinit.idx = phi i32 [ 0, %entry ], [ %3, %valinit.body ]
8    %1 = icmp slt i32 %valinit.idx, 20
9    br i1 %1, label %valinit.body, label %valinit.end
10
11 valinit.body:                                      ; preds = %valinit.pred
12   %2 = getelementptr inbounds [20 x double], [20 x double]* %a., i32 0, i32 %valinit.idx
13   store double 0.000000e+00, double* %2
14   %3 = add nsw i32 %valinit.idx, 1
15   br label %valinit.pred
16
17 valinit.end:                                       ; preds = %valinit.pred
18   ret void
19 }
```

(b) The corresponding LLVM program

Fig. 4. An example of C program and its corresponding LLVM program

to allocate memory on the stack frame. We can also allocate heap memory by invoking `malloc` or other heap management functions. Given a pointer to the memory, we can use **store** to change its value. Therefore, when we implement mutation in the code generation phase, we need to somehow ensure that a portion of the memory is properly allocated.

*4.1.4 Basic Blocks.* Control flows in LLVM are divided by blocks of instructions called *basic blocks*. A function is composed of one or more basic blocks and has exactly one entry block. Other than the entry block, all basic blocks have any number of predecessor blocks and can jump to any number of successor blocks. Instructions in a block are executed sequentially. The last instruction of a basic block has to be a *terminator instruction*, which either returns the current function, or jumps to other basic blocks. All basic blocks of a function and their terminator instructions form a control flow diagram of the function.

*4.1.5 Correspondence between C.* A LLVM program sometimes finds a great similarity to its counterpart in C. Figure 4 shows a simple C program and its translation. The C program defines a function, in which an array of **doubles** are allocated on the stack and all its elements are set to be zero. The LLVM program is much more verbose, but if we look closer, it just add greater details to the C program. Essentially line 8 tests whether `i` is

a less than 20, and if so, the basic block from line 11 is execute. It uses **getelementptr** to compute the pointer position of the element in the array on line 12 and use **store** to store zero to that position on line 13.

In general, we can often see a very straight connection between C programs and LLVM programs, while LLVM programs are just more verbose. Even though we are generating LLVM programs, in the rest of the report, we choose to present C programs in most cases. We only present LLVM IR when certain very specific concepts are crucial in code generation.

## 4.2 Relating Types between GoLite and LLVM

Before starting to write the code generator, we should pause a second to consider a few important questions:

(1) How do types between GoLite and LLVM correspond?
(2) How do names between GoLite and LLVM correspond?
(3) How should functions be generated (like equality comparison between **struct**s and initialization of **struct**s)?

These are fundamental questions and are better determined up front. Let us consider the first question. The second question is answered in Section 4.6 and the third is answered in Section 4.7. Since LLVM has most types GoLite has (except slices, which will be discussed in Section 4.3), the first attempt is to do a direct translation from types in GoLite to ones in LLVM:

$$
\begin{aligned}
[\![\texttt{string}]\!] &= \texttt{i8*} \\
[\![\texttt{int}]\!] &= \texttt{i32} \\
[\![\texttt{float}]\!] &= \texttt{double} \\
[\![\texttt{rune}]\!] &= \texttt{i32} \\
[\![(\texttt{x},\ \tau)]\!] &= x && \text{(where } \tau \text{ is a } \texttt{struct}\text{. We also create a type declaration as a side effect.)} \\
[\![(\texttt{x},\ \tau)]\!] &= [\![\tau]\!] && \text{(where } \tau \text{ is not a } \texttt{struct}\text{)} \\
[\![\texttt{[n]}\tau]\!] &= \texttt{[n x } [\![\tau]\!]\texttt{]} \\
[\![\texttt{struct \{ } x_i\ \tau_i\texttt{; \}}]\!] &= \texttt{\{ } [\![\tau_i]\!]\texttt{, \}} && \text{(where } i \text{ ranges over the length of the } \texttt{struct}\text{)}
\end{aligned}
$$

We use $[\![]\!]$ to denote the translation from GoLite to LLVM. We overload the symbol for the translations of types, expressions and statements. The actual meaning will be clear from the context. We omit the case for slices for now and this case will be expanded shortly. Despite being naive, this translation has already contained some significant details.

(1) **string**s are represented by **i8***, a pointer to bytes, which is equivalent to **char** * in C.
(2) We have two case for type references. Recall that in Section 3.1, we concluded that a type reference is represented by a tuple of its name and its resolved type. In the first case for type references where the resolved type $\tau$ is a **struct**, we create a type definition in the final LLVM IR and then return the reference itself. A type definition in LLVM is similar to one in Go and creates a unique named type. LLVM only permits type definition for structs. Thus we do this check to make sure the type definition is well-formed.
(3) If the resolved type $\tau$ is not a **struct**, then we cannot refer to this type as x in LLVM because a type definition cannot be created, so we have to translate the resolved type instead.
(4) The cases for arrays and **struct**s are straightforward because LLVM supports corresponding aggregate types.

There have already been the following obvious problems in this translation:

(1) In the type reference case when $\tau$ is not a **struct**, our translation recurs down to the resolved type, which makes the translation no longer structural. As a result, this translation might not terminate and thus malformed. Consider the following goLite program:

```
type foo []foo
```

It is worth a pause to think about what this type should be translated to. Consider the following tentative translation:

$$[\![\texttt{(foo, []foo)}]\!] = \texttt{slice-in-LLVM}([\![\texttt{(foo, []foo)}]\!])$$

This translation is clearly malformed because the translation of `foo` depends on the translation of itself and this translation can never terminate. This means we have to be very careful about how slices are translated to in LLVM. We do have a solution to this and it will be discussed in Section 4.3.

(2) A less obvious problem has something to do with performance. Let us consider the following program:

```
var x struct { foo [100] struct { bar int; }; }
var y = x.foo[0].bar
```

This innocent program can be incredibly slow if we fail to be careful enough. Recall that LLVM employs SSA and values are stored in registers. Since registers are not part of the memory, assignments to registers copy the whole values, even when the values are *aggregate types*. That is, in a very unexpected case, this program is equivalent to the following in performance:

```
var x struct { foo [100] struct { bar int; }; }
var tmp0 = x.foo
var tmp1 = tmp0[0]
var y = tmp1.bar
```

In this program, `x.foo` copies the array of length 100 to `tmp0`, only to access the first element in `tmp1`, which also copies the **struct** containing `bar`. The majority of the copies are in vain, so we should attempt to avoid it. The solution is discussed in Section 4.4.

### 4.3 Slices

In this section, we tackle to representation problem of slices. It is important because if the solution is too complex, the implementation would be much more error-prone, the translation function might not even terminate in some cases, and thus our final correctness argument could be compromised.

Our solution to this problem is simple: all slices are represented as one single type in LLVM! In that case, `[]foo` in the previous example becomes a fixed type in LLVM which does not require any translation of the `foo` nested inside. An implementation of slices in C is shown in Figure 5. Again, we use C for conciseness. Memory management details are omitted in the code fragment.

Slices are represented as a struct with four fields, meaning the capacity, the length, the byte buffer for data and the size for each element in the buffer respectively. This struct directly translates to a fixed struct type in LLVM. Thus we add the case for slices to the translation:

$$[\![\texttt{[]}\tau]\!] = \texttt{\%slice*}$$

where `%slice` is the named struct in LLVM corresponding to `slice` in Figure 5. Notice that a slice is translated to a pointer to `%slice` in LLVM. We claim that the translation function terminates.

THEOREM 4.1. *The type translation function* $[\![]\!]$ *with the case added above terminates.*

This theorem follows from two facts: the translation of slices above goes to a fixed struct and in the second case for type references in Section 4.2, $\tau$ can only contain type references defined before `x`. The latter says that the translation forms a directed acyclic graph of type references and thus serves as a termination argument.

```c
typedef struct slice {
        // capacity
        int cap;
        // length
        int len;
        // byte buffer
        char *buf;
        // size of each element
        size_t elem;
} slice;

slice *do_new_slice(size_t elem)
{
        slice *s = malloc(sizeof(slice));
        s->cap = 0;
        s->len = 0;
        s->buf = NULL;
        s->elem = elem;
        return s;
}
```

```c
slice *slice_augment(slice *s)
{
        slice *ret = malloc(sizeof(slice));
        ret->cap = s->cap;
        ret->len = s->len;
        ret->elem = s->elem;
        if (s->len < s->cap) {
                ret->buf = s->buf;
        } else if (!s->NULL) {
                ret->buf = malloc(2 * s->elem);
        } else {
                ret->cap *= 2;
                ret->buf =
                    malloc(ret->cap * s->elem);
                memcpy(ret->buf, s->buf,
                           s->cap * s->elem);
        }
        ret->len += 1;
        return ret;
}
```

Fig. 5. Implementation of slices in C

In Figure 5, we also show two functions. `do_new_slice` allocates a new slice in the heap. It sets both the capacity and the length to zero and remembers the size of each element. `slice_augment` returns a new slice which can fit at least one more element than the original one. This function is called before we append an element to a slice. Before we access element from a slice, we need to cast the `buf` pointer to the right type. This is always possible because our typed expressions know their own types.

## 4.4 Involving Pointer Arithmetic

In Section 4.2, we gave an example which wastes nearly all computation power due to intermediate copies. This happens if we translate (right) expressions in GoLite one to one in LLVM. We formalize this claim as follows:

$$\text{If } \Gamma \vdash^r e : \tau \text{, then } \Vdash^r [\![e]\!] : [\![\tau]\!]^1$$

We use $[\![e]\!]$ to denote the translation of right expressions from GoLite to LLVM. The judgment $\Gamma \vdash^r e : \tau$ denotes that $e$ as a right value has type $\tau$. The judgment $\Vdash^r [\![e]\!] : [\![\tau]\!]$ denotes that the translated expression has the translated type in LLVM. This assertion reads: if $e$ has type $\tau$ in GoLite, then the translation of $e$ has the translated type $\tau$ in LLVM. This assertion is plausible; it at least does what we expect. If we implement $[\![e]\!]$, then we indeed obtain a correct compilation from GoLite to LLVM. However, the problem is the performance. Let us examine the example in Section 4.2 once more:

```go
var x struct { foo [100] struct { bar int; }; }
var y = x.foo[0].bar
```

Since the translation must always return values, it first looks at `x.foo` and translates it to a value in LLVM which are copied, essentially

[1]We very consistently use blue to denote typed terms.

```
var tmp0 = x.foo
```

Then it reaches `x.foo[0]`, looks up the first value in the array and copies it to a register:

```
var tmp1 = tmp0[0]
```

Finally another index is performed which returns the only data we want to copy in the entire program:

```
var y = tmp1.bar
```

Again, this compilation strategy generates very inefficient code and the ultimate cause is that we require $[\![e]\!]$ to always have type $[\![\tau]\!]$. Therefore, the solution is to not always require this property. Our translation for right expressions satisfies the following slightly more complicated property instead:

THEOREM 4.2. *If* $\Gamma \vdash^r e : \tau$ *and* $\tau$ *is not an aggregate type, then* $\Vdash^r [\![e]\!] : [\![\tau]\!]$.
*If* $\Gamma \vdash^r e : \tau$ *and* $\tau$ *is an aggregate type (*`struct`*s or arrays), then* $\Vdash^r [\![e]\!] : [\![\tau]\!]*$.

Let us reexamine the previous program to see that this theorem is the property we want. First the translation sees `x.foo`. Instead of returning the array as a value, it returns a pointer to that array. Then `x.foo[0]` computes the address of the first element in the array, which has the same value as the previous pointer. Finally, `x.foo[0].bar` correctly dereferences the pointer and obtain an integer value. In this translation, we copy only the target integer. Thus, this property is much more warranted and should be aimed at in the implementation.

This property might look convoluted because the type of $[\![e]\!]$ is determined by $\tau$. In fact, this invariant is very easy to maintain. In the actual implementation, we have the following small function:

```
let expr_invariance v t =
  if is_aggregate t then v else build_load v "" g.builder
```

This function receive an LLVM pointer `v` and a semantic type `t`. We test whether the type is aggregate. If so, we return the pointer, otherwise we create a **load** command to load the value from the memory via the pointer. This function is used to wrap the return values of the cases in which it is possible to return an aggregate type: variables, indexing, and selection. In equality comparison, since aggregate types can be compared, we need to insert `expr_invariant` appropriately. This is discussed in Section 4.7. Function applications also need special care, which is discussed in Section 4.6.

Now $[\![e]\!]$ does not necessarily return a value, so we have to consider how assignment is arranged. In an assignment statement in GoLite

```
x = e
```

when the expression `e` does not have an aggregate type, we simply assignment the value to the location of `x` in the memory via the **store** command. If it has an aggregate type, It is still possible to use **store** command. We can first use **load** command to load from the pointer and then **store** it to `x`. Our implementation is more direct. We use `memcpy` to copy the data from the pointer computed by `e` to `x`.

## 4.5 Addressable and Left Expressions

In this section, we briefly discuss addressable and left expressions. Addressable expressions in fact are much easier to handle than right expressions. Addressable expressions only occur to the left of an assignment, its translation must return an address to assign to.

THEOREM 4.3. *If* $\Gamma \vdash^a e : \tau$*, then* $\Vdash^a [\![e]\!]^a : [\![\tau]\!]*$.

The judgment $\Gamma \vdash^a e : \tau$ denotes that the addressable expression $e$ has type $\tau$. We use $[\![]\!]^a$ to denote the translation of addressable expressions and the judgment $\Vdash^a [\![e]\!]^a : [\![\tau]\!]*$ denotes the translation of $e$ has type $[\![\tau]\!]*$ in LLVM. For the following programs:

```
e++
e--
e += e'
```

We have obtained a pointer of $e$ via $[\![e]\!]^a$. By inspecting the typing rules for all three kinds of statements, we know that $[\![e]\!]^a$ must point to a non-aggregate type and thus we just need to use **load** to load the value and combine it with other operations. The correctness of this treatment is justified by Theorem 3.2.

For left expressions, we simply discriminate the case for blank identifiers and the case for addressable expressions is handled by $[\![]\!]^a$.

## 4.6 Variables and Functions

In the previous sections, we have discussed a complication due to a performance consideration involving unnecessary copying and have concluded that Theorem 4.2 is the right property we are looking for. However, in that conclusion we made an implicit assumption about variables. We assume that all variables have addresses. Recall that in LLVM, registers do not have addresses, so variables cannot be simply mapped to registers. In this section, we consider how variables are allocated in the memory. There are a few cases to consider.

*4.6.1 Global Variables.* All global variables in LLVM are allocated in memory, so this case is the easiest. We assign a global variable an LLVM type $[\![\tau]\!]$ where $\tau$ is the GoLite type of the variable. Then the pointer to it is automatically $[\![\tau]\!]*$. The memory locations for global variables are handled by LLVM.

*4.6.2 Local Variables Declared In the Body of a Function.* The cases for local variables are more complex. We first consider local variables in the body in contrast of function parameters.

```
func foo() {
    var x int
}
```

Let us consider how variables like x should be allocated. This case is similar to the case for global variables. We assign one local variable an LLVM type $[\![\tau]\!]$ where $\tau$ is its GoLite type. We allocate these variables on the stack by using the **alloca** command. thus their memory is automatically released once the function returns.

*4.6.3 Function Parameters.* So far the cases are quite straightforward. We can easily find a location in memory for the aforementioned variables. Now we move on to consider function parameters. Consider the following code fragment:

```
func foo(x int, y struct { bar int; }) {  }
```

After a quick thought, we arrive at the conclusion that x should be allocated the same as regular local variables, namely on the stack using the **alloca** command. We then use a **store** command to store the value of x on the stack.

What about y of an aggregate type?

Recall that GoLite implements some *call-by-value* semantics. That is, when foo is invoked, the value in y's position should be copied and the copy is passed in the function. If we alter the the bar field of y in foo's body, it should not alter the value given by the caller. One tentative answer is to also do the same for aggregate types as normal variables. A parameter of type $\tau$ in GoLite is translated to one of type $[\![\tau]\!]*$ in LLVM and the value is copied to the stack to obtain an address. This solution seems reasonable at its first glance until we study one optimization phase *SROA*.

SROA is short for *Scalar Replacement of Aggregates.* In this optimization phase, aggregate types are expanded to scalars. That is, y.bar will be assigned to a register regardless. This might look harmless but this phase also runs for arrays:

```
func foo2(x int, y struct { bar int; }, z [10000]int) {  }
```

SROA will unfold z 10000 times and assign each element to a register. In general, SROA has an exponential complexity if we regard parameters of aggregate types like y and z as values in LLVM. This gives us a reason to refrain from continuing this option. Instead, we assign aggregate types in function parameter $[\![\tau]\!]*$. Though this translation prevents SROA from applying, it alone would break the call-by-value semantics. In order to conform the call-by-value semantics, we add an LLVM attribute **byval** to the parameter. This attribute instructs LLVM to allocate enough memory for the values of the parameters in the caller. This is handled conveniently silently by LLVM. foo2 is translated to the following LLVM function by our code generator.

```
define void @foo2(i32 %x, { i32 }* byval %y, [10000 x i32]* byval %z) {
entry:
  %x. = alloca i32
  store i32 %x, i32* %x.
  ret void
}
```

At this point, we have determined all sources of memory locations for variables.

*4.6.4 Return Values of Functions.* In previous sections we discussed inputs to functions, now we consider the outputs. We see that there is an advantage of delegating aggregate types to pointer. In our implementation, we also do the same outputs.

```
func foo() struct { x int; } { /* code omitted */ }
```

Instead of returning the output as a value in LLVM, we require the caller to pass in a pointer to a struct as the last parameter and the function to return **void**. This might look convoluted at the first glance but this design is consistent with Section 4.4, in which we use memcpy to implement assignment of aggregate types. If we return by value, we would create exceptional cases here and there and complicate our program logic. foo then becomes:

```
define void @foo({ i32 }*) {
entry:
  ; body omitted
  ret void
}
```

At this point, we have arranged allocations of all data.

*4.6.5 Addresses of Local Variables.* In Section 3.2.4, we showed our strategy to map *all* local variables to an index in a list of types. In code generation, we use the same idea and keep track of a list of LLVM pointers. For each GoLite variable x of type $\tau$, the list remembers $[\![\tau]\!]*$ in the corresponding index. When we look for the address of a local variable, it is as easy as indexing this list. The stack allocation occurs right after the entry of the function, so all variables are guaranteed their own locations in memory and the locations do not overlap. How the memory addresses are determined has implicitly resolved the name clashing issue which typically needs to be handled by explicit renaming.

## 4.7 Equality Comparison and Initialization for Aggregates

In the past sections, we focused on reasoning about types and how their invariants should be maintained. We concluded that Theorem 4.2 is the desired property and expr_invariant is invoked to ensure this property. The conclusion has formed the spine of our code generator, and what is left is to fill in some implementation detail.

In GoLite, aggregate types like `struct`s and arrays can be compared by equality and all variables in GoLite should be given initialized values if one is not provided. For `struct`s, these two are implemented by generated functions.

Consider a situation in which we need to compare two arrays `x` and `y` of type `[n]`$\tau$. Due to Theorem 4.2, we know that they are given as two LLVM pointers of type `[n x` $[\![\tau]\!]$`]*`. It is clear that we just need to loop over all elements and compare each pair of elements of type $\tau$. This is done via doing the pointer arithmetic and obtaining the pointer of type $[\![\tau]\!]$`*`. There are two cases depending on $\tau$: if it is an aggregate type, then we keep them as pointers and recurse; otherwise, we need to apply the `load` command to load the values. This is precisely what `expr_invariant` does. We can then recurse on the values returned by `expr_invariant`.

`struct`s are very similar. We need to access each field in sequence and compare them recursively. Comparisons for `struct`s are delegated to generated functions. The names of the functions are mangled names of the `struct`s themselves so we can avoid generating duplicated functions for essentially the same comparison.

Initialization takes a similar strategy. For arrays, we loop over each entry and assign it a default value recursively. For `struct`s, we also generate functions with mangled names and assign a default value for each field.

## 4.8 Bound Checking

In GoLite, we do not permit out of bound access. Therefore, we need to do some bound checking when we index an array or a slice. This is done by calling a `bound_check` function prior to all indexing. In the `bound_check` function, if the index is larger then the length of the array or the slice, we invoke the `panic` function. This function simply prints a readable message to the standard error and exit the program immediate with non-zero value.

## 4.9 Translation of Statements

The translation for expressions is by far the most complex portion in the whole code generation phase. Due to the invariant, we have been able to follow a very clear thought in our implementation. Compared to expressions, the translation for statements is much simpler. Assignments and declarations are very easy because we have considered how these should be handled in previous sections. What remain are printing and control flows.

*4.9.1 Printing Statements.* Printing is handled by invoking `fprintf` from C standard. We only need to generate a format string and pass in the parameters.

*4.9.2 Invariant for Basic Blocks.* When handling control flow, we cannot avoid directly handling basic blocks. They are the building blocks for translation of control flows. In Section 4.1.4, we mentioned that each basic block must have one terminator instruction for the LLVM program to be well-formed. This well-formedness condition adds some complexity because some well-formed GoLite program does not correspond to a well-formed LLVM program directly. Consider the following program:

```
func foo1() {
    return
    return
}
```

This is a well-formed GoLite program. However, if we translate it directly to LLVM, we would generate the following code:

```
define void @foo1() {
entry:
  ret void
```

```
  ret void  ; one extra terminator instructor
}
```

This LLVM program is not well-formed because the `entry` basic block contains two terminator instructions.

A direct translation also might also miss a terminator instruction.

```
func foo2() { }
```

This program would be translated to

```
define void @foo2() {
entry: ; there is no terminator!
}
```

These two malformed cases need to be dealt with so that we can be sure our code generator always output valid LLVM programs. This is achieved by introducing another function $\langle\!\langle s \rangle\!\rangle$ which judge whether $s$ has generated a terminator instruction for a statement $s$. Only when $s$ is **return**, **continue** and **break**, $\langle\!\langle s \rangle\!\rangle$ returns true. We further augment $[\![\,]\!]$ and $\langle\!\langle\,\rangle\!\rangle$ to sequences of statements satisfying the following properties:

Theorem 4.4. *$[\![\,]\!]$ and $\langle\!\langle\,\rangle\!\rangle$ have the following relation when translating a sequence of statements:*

$$
\begin{aligned}
\langle\!\langle \cdot \rangle\!\rangle &= \mathit{false} & & \\
\langle\!\langle s; s* \rangle\!\rangle &= \langle\!\langle s* \rangle\!\rangle & & \text{If } \langle\!\langle s \rangle\!\rangle \text{ is not true} \\
\langle\!\langle s; s* \rangle\!\rangle &= \mathit{true} & & \text{If } \langle\!\langle s \rangle\!\rangle \text{ is true} \\
[\![ \cdot ]\!] &= \cdot & & \text{generates no instruction} \\
[\![ s; s* ]\!] &= [\![ s ]\!]; [\![ s* ]\!] & & \text{If } \langle\!\langle s \rangle\!\rangle \text{ is not true} \\
[\![ s; s* ]\!] &= [\![ s ]\!] & & \text{If } \langle\!\langle s \rangle\!\rangle \text{ is true}
\end{aligned}
$$

The equation for $[\![\,]\!]$ says that when the leading statement generates a terminator instruction, we do not generate code for subsequent statements $s*$. This effectively prevents the `foo1` function above from generating two **`ret void`**s.

The problem for not generating a terminator instruction at all is more subtle. In the `foo2` function, we intuitively know that a **`ret void`** should be inserted, but how much cases this intuition can cover? We shall reason more carefully.

When a function has no return value, GoLite allows to have no **return**. In this case, it is correct to insert a **`ret void`**.

However, even when a function has return value, it is still possible to generate a basic block with no return value. Let us consider the following function and its generated code:

```
func foo3(b bool) int {
    if b {
        return 1
    } else {
        return 2
    }
}
```

```
define i32 @foo3(i1 %b) {
entry:
  br i1 %b, label %if.true, label %if.false

if.true:                                    ; preds = %entry
  ret i32 1

if.false:                                   ; preds = %entry
  ret i32 2

if.end:                                     ; No Terminator!
}
```

Notice that the basic block `if.end` is empty and contains no terminator instruction. We could have been really careful and keep track of what instructions both branches are translated to, but there is a much simpler solution. We simply insert an **`unreachable`** instruction, indicating that the `if.end` block is not reachable. This has handled the terminator instructions for functions.

We do not always want to insert **`ret`** or **`unreachable`** when a terminator instruction is needed. Consider the following two functions:

```
func foo4(b bool) int {
    if b {
    }
    return 2
}
```

```
func foo5(b bool) int {
    if b {
        return 1
    }
    return 2
}
```

When we generating the **`if`** block in `foo4`, since it does not end with a terminator instruction, we need to insert an unconditional branch to the ending block, which executes **`return`** `2`. Meanwhile, in `foo5`, we *must not* generate a branching at the end of the **`if`** because it contains a **`return`**. The distinction can be made due to the ⟪⟫ function. We use ⟪⟫ to test whether the **`if`** block is translated to a terminator instruction or not. If not, we append an unconditional branching to the end of the **`if`**. The same treatment is applied to the **`else`** block too if it exists. Statements in **`switch`** and **`for`** are handled in a similar way.

*4.9.3 Switch Statements.* **`switch`**s are very complex because they test for equality and contain an indefinite number of code blocks. Conceptually, a **`switch`** statement can be recursively generated by the following translation:

```
switch e {
case e1, ... en:
  s*
// other cases
}
```

⇒

```
var v = e
if v == e1 || ... || v == en {
  s*
} else {
    switch v {
    // other cases
    }
}
```

We adopt a similar strategy. Special care is taken to handle the **`default`** case. Otherwise, it is very similar to an **`if`** statement.

*4.9.4 Continue and Break.* For **`switch`** and **`for`**, we need to keep track of the basic blocks so that when **`continue`** or **`break`** is encountered, we know which basic block to jump to. Concretely, we keep track of four states:

(1) Not inside of any **`switch`** or **`for`**: **`break`** and **`continue`** are illegal, which has been guarded by the weeding process of the parser.
(2) Inside of a **`switch`** that is not inside of a **`for`**: only **`break`** is legal. We need to keep track of the basic block immediately following the **`switch`** and **`break`**. **`break`** translates to a unconditioned branch to that block. **`continue`** is illegal in this case.
(3) Inside of a **`for`**: we keep track of the basic block for the loop condition and the basic block immediate following the **`for`**. **`continue`** jumps to the condition block and **`break`** jumps to the ending block.
(4) Inside of a **`switch`** that is inside of a **`for`**: we keep track of the basic block after **`switch`** and the condition block of the **`for`**. **`break`** jumps to the ending block of **`switch`** and **`continue`** jumps to the condition block of the **`for`**.

We need to carefully maintain these four states when we encounter **`switch`**s and **`for`**. The transition diagram is shown in Figure 6. Based on the state, we generate **`continue`** and **`break`** accordingly.

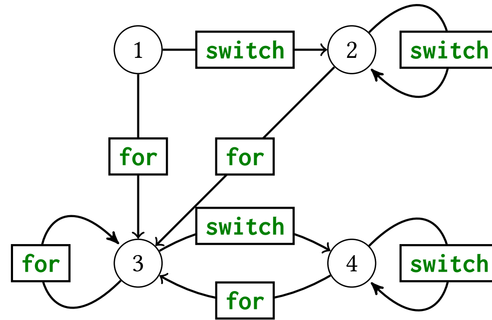


Fig. 6. State transition

### 4.10 Main Function and Program Initialization

We have set up all necessary components for code generation of expressions and statements. In this section we set up the entry point of the whole program. This is done via composing a `main` function. We first specify the initialization order:

(1) We initialize global variables in their order of appearances in the file.
(2) We invoke `init` functions in their order of appearances in the file.
(3) We invoke the `main` function defined in the file if it exists.

Thus, after generating all contents in source code, we generate the code in the `main` function according to this specification. Notice that some name mangling is need to avoid the using the `main` function in the source as the final entry point.

### 4.11 Well-formedness Verification of the Whole Module

After generating the `main` function, we have obtained "hopefully" a well-formed LLVM module. In the OCaml binding of LLVM, there provides a function `Llvm_analysis.verify_module` which allows us to verify the well-formedness of the generated module. This serves as the double check for our reasoning in the code generation phase.

## 5 LINKING AND EXECUTION

We finally have arrived at the end of the diagram in Figure 1. After code generation, we have obtained an LLVM file. We then need to convert it into an executable file. Before assembling it, we take the advantage of the optimizers in LLVM by invoking `opt`:

```
opt -S -O2 $INPUT -o $OUTPUT
```

`-S` flag means that the output is still an LLVM IR file and `-O2` flag triggers a level-2 optimization, which improves greatly the performance of our naively generated code.

We then pass the output file to `llc` which assembles the file to an object file. After that, we invoke `clang` to link the object file with `clib` and obtain an executable file. This gives us access to C standard functions like `malloc`, `fprintf` and others.

## 6 CORRECTNESS ARGUMENT

Up to this point, we have implemented a GoLite compiler to LLVM IR. We have conducted some formal analysis to guide our development and summarized a number of theorems. In this section, we connect these theorems together and argue that our compiler is *provably correct*.

(1) Theorem 3.3 asserts that the type checker converts an untyped expression to a typed one in a type preserving manner. This theorem justifies the correctness of the type checker.
(2) Theorem 4.1 asserts that the type translation from GoLite to LLVM is well-formed. This very important property provides a basis to measure our code generator.
(3) Theorem 4.2 and Theorem 4.3 provides guideline properties which we aim at when generating code for expressions and addressable expressions. It is very easy to see that our code generator satisfies these invariants due to the `expr_invariant` function.
(4) Theorem 4.4 asserts that the translation for statements always generate well-formed blocks.

These theorems combined together essentially ensure us the correctness of type checking, as well as translations of types, expressions and statements from GoLite to LLVM. Up to satisfaction of the invariants, the compiler correctly translates a GoLite program to LLVM.

## 7 CONCLUSION AND FUTURE WORK

In this report, we discussed our implementation of a GoLite compiler. We discussed how parsing, type checking and code generation are done in our implementation. Throughout the implementation, we carried a number of invariants in mind, in order to obtain a clean implementation and argue its correctness. We specified a number of theorems we aimed at during the development and explained why these theorems are the right properties to go after.

Employing LLVM as the backend opens the door to many interesting future directions. In our implementation, we use heap for data types like strings and slice and we never reclaim the heap memory in our generated code. A very interesting future work is to balance the heap. It is also possible to delegate memory management to garbage collection (GC).

## REFERENCES

Arthur Charguéraud. 2012. The Locally Nameless Representation. *J. Autom. Reasoning* 49, 3 (2012), 363–408. https://doi.org/10.1007/s10817-011-9225-2

Nicolaas Govert de Bruijn. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)* 75, 5 (1972), 381 – 392. https://doi.org/10.1016/1385-7258(72)90034-0

Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. IEEE Computer Society, 75–88. https://doi.org/10.1109/CGO.2004.1281665

Xavier Leroy. 2006. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, J. Gregory Morrisett and Simon L. Peyton Jones (Eds.). ACM, 42–54. https://doi.org/10.1145/1111037.1111042

John R. Levine. 2009. *flex and bison - Unix text processing tools.* O'Reilly. http://www.oreilly.de/catalog/9780596155971/index.html

Barbara Liskov and Stephen N. Zilles. 1974. Programming with Abstract Data Types. *SIGPLAN Notices* 9, 4 (1974), 50–59.

Yaron Minsky, Anil Madhavapeddy, and Jason Hickey. 2013. *Real World OCaml - Functional Programming for the Masses.* O'Reilly. http://shop.oreilly.com/product/0636920024743.do#tab_04_2

Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1988. Global Value Numbers and Redundant Computations. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*, Jeanne Ferrante and P. Mager (Eds.). ACM Press, 12–27. https://doi.org/10.1145/73560.73562

The Coq Development Team. 2020. *The Coq Proof Assistant, version 8.11.0.* https://doi.org/10.5281/zenodo.3744225