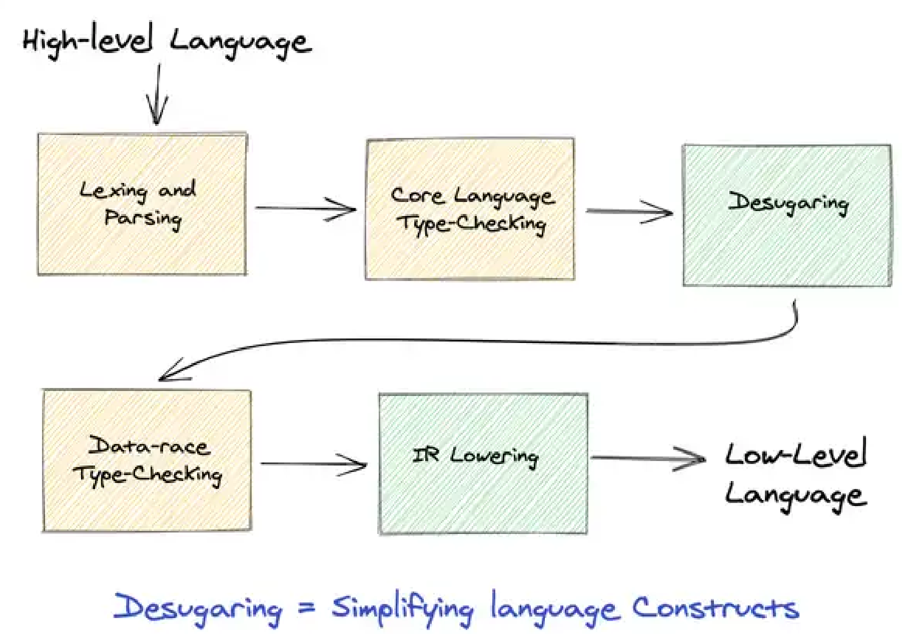
TABLE OF CONTENTS

CREATING THE BOLT COMPILER: PART 6

Desugaring - taking our high-level language and simplifying it!

JULY 01, 2020 6 MIN READ



SERIES: CREATING THE BOLT COMPILER

- Part 1: How I wrote my own "proper" programming language
- Part 2: So how do you structure a compiler project?
- Part 3: Writing a Lexer and Parser using OCamllex and Menhir
- Part 4: An accessible introduction to type theory and implementing a type-checker
- Part 5: A tutorial on liveness and alias dataflow analysis
- **Part 6: Desugaring - taking our high-level language and simplifying it!**

Just give me the code!

All the illustrative code snippets in this blog post link to the respective file in the Bolt repository. There's more code in there than could be covered without making this post extremely long!

The first half of this post will be looking at the desugaring/ folder and the second half is covering the ir_gen/ folder.

What is desugaring?

Programming languages are a series of abstractions. No one writes programs by typing in 0s and 1s - it's just not human readable. The closest we get to the hardware operations is with **assembly code** e.g. series of `add` `mov` and `jmp` instructions.

Assembly code is still not really a pleasant programming experience. Even languages we deem as *low-level* like C / C++ / Rust offer a host of abstractions over assembly code - things you take for granted like `if` statements and `while` loops.

We call these abstractions **syntactic sugar** - named because they make it *sweeter* for programmers to program in that language.

When we're writing a compiler though, we're going the other way - we're *desugaring* the source code - stripping away higher-level constructs. We also refer to this as *lowering* the high-level language constructs.

In this post we'll start by looking at desugaring a for loop. We'll then look at the "Desugaring" and "IR Lowering" stages in the Bolt compiler frontend. This will wrap up our compiler frontend and set us up to switch to C++ for the compiler backend.

Desugaring For Loops

The first case we desugar is actually between the parsing and type-checking phases - desugaring a `for` loop into a while loop:

desugar_for_loop.bolt

Copy

```
for (let i = 0; i < n; i:=i+1) {
  doSomething
}

// desugared

let i = 0;
while (i < n) {
  doSomething;
  i:=i+1
}
```

Note we handle this as a special case when type-checking the expression, however you might imagine if there was more sugar (like `++i` instead of `i:=i+1`) that we might add a full desugaring stage between the parsed AST and typed AST:

type_expr.ml

Copy

```
let rec type_expr class_defns function_defns (expr : Parsed_ast
...
| Parsed_ast.For
      (loc, start_expr, cond_expr, step_expr, Parsed_ast.Block
      (* desugar into a while loop *)
      type_block_with_defns
        (Parsed_ast.Block
           ( loc
           , [ start_expr
             ; Parsed_ast.While
                 (loc, cond_expr,
                 Parsed_ast.Block (block_loc, loop_expr @ [step
             ] ))
        env
```

Desugaring between type-checking stages

Desugaring gets its own stage in between the two stages of type-checking. The data-race type-checking is much more complex than the traditional type-checking (`int` , `bool` etc), so we simplify the language to *avoid having to consider as many cases*.

Removing variable shadowing

For example, consider variable shadowing, where we can declare the same variable name `x` in nested scopes: Consider the following:

variable_shadowing.bolt

Copy

```
let x = 0;
if (x >= 0) {
  let x = 1;
  let y = x + 1 // we now refer to the value x=1
}
else { // we refer to the value x=0
 x := 1
}
```

Variable shadowing is syntactic sugar - we don't require the programmer to use unique variable names in nested scopes. It makes the alias liveness analysis previously discussed much harder. How do we know which value of x is being aliased? We could track which scope we're in *orrrr* we could avoid it. It's much easier to deal with once we give variables unique names:

unique_variable_names.bolt

Copy

```
let _x0 = 0;
if (_x0 >= 0) {
  let _x1 = 1;
  let y = _x1+ 1
else {
 _x0 := 1
}
```

We first create a mapping from old to new variable names. We count the number of times the variable has been declared so far in outer scopes and stick that count on the end of the variable name. And to specify that these are compiler-generated names we prepend them with an `_`, since in Bolt programmers can't define a variable starting with an `_`.

remove_variable_shadowing.ml

Copy

```
type var_name_map = (Var_name.t * Var_name.t) list

let set_unique_name var_name var_name_map =
  let num_times_var_declared =
    List.length (List.filter ~f:(fun (name, _) -> name = var_na
  Var_name.of_string
    (Fmt.str "_%s%d" (Var_name.to_string var_name) num_times_va
```

Desugaring Function / Method Overloading

Function overloading is where we define multiple functions with the *same* name but *different* parameter types. This is useful if you want to call a different `print` method based on the type of the arguments passed in:

function_overloading.bolt

Copy

```
function void print(Foo x){
  ...
}
function void print(Bar x){
  ...
}
function void print(int x){
  ...
}
```

Again, this is a nice-to-have construct, but we've got an issue - which function do we call? We can't tell from the source code, but we can use the information about the argument types from the previous type-checking stage.

> By desugaring more complex language constructs to simpler language constructs, we make subsequent stages of the compiler simpler - they **do not need to know** about anything that has been desugared.

We encode the type of the parameters in the function application expression when type-checking it:

type_expr.ml

Copy

```
| Parsed_ast.FunctionApp (loc, func_name, args_exprs) ->
      type_args type_with_defns args_exprs env
      >>= fun (typed_args_exprs, args_types) ->
      get_matching_function_type class_defns func_name args_typ
      >>| fun (param_types, return_type) ->
      ( Typed_ast.FunctionApp (loc, return_type, param_types, f
      , return_type )
```

Name mangling functions

Now since each overloaded function has differing parameter types, we can map the parameter types to a unique string, which we append onto our function name. We call this process of generating a unique function name **name mangling**.

We're going to take the approach used in C++.

For each of the primitive types, we can map them to a unique single character, whilst for classes we map them to the class name prepended with its length. We then concatenate all param types together.

Why prepend the length? Consider param types `(Foo x, Bar y)` and `(FooBar x)` - both would map to `FooBar` if we concatenated their parameter names. Only when we prepend the lengths can they be distinguished - `3Foo3Bar` vs `6FooBar` .

desugar_overloading.ml

Copy

```
let name_mangle_param_types param_types =
  String.concat
    (List.map
```

```
    ~f:(function
      | TEVoid                 -> "v"
      | TEInt                  -> "i"
      | TEBool                 -> "b"
      | TEClass (class_name, _) ->
          let class_name_str = Class_name.to_string class_na
          Fmt.str "%d%s" (String.length class_name_str) clas
    param_types)
```

And then to name mangle a method or function, we have the following code:

[desugar_overloading.ml]

Copy

```
let name_mangle_overloaded_method meth_name param_types =
  Method_name.of_string
    (Fmt.str "_%s%s"
       (Method_name.to_string meth_name)
       (name_mangle_param_types param_types))
```

For example, with this name mangling scheme, `testFun(Foo x, Bar y)` maps to `_testFun3Foo3Bar(Foo x, Bar y)` .

> If you look at the master branch of the Bolt repo, you'll notice the desugaring stage also desugars generics. That's a topic that deserves its own post later in the series!

Lowering to IR

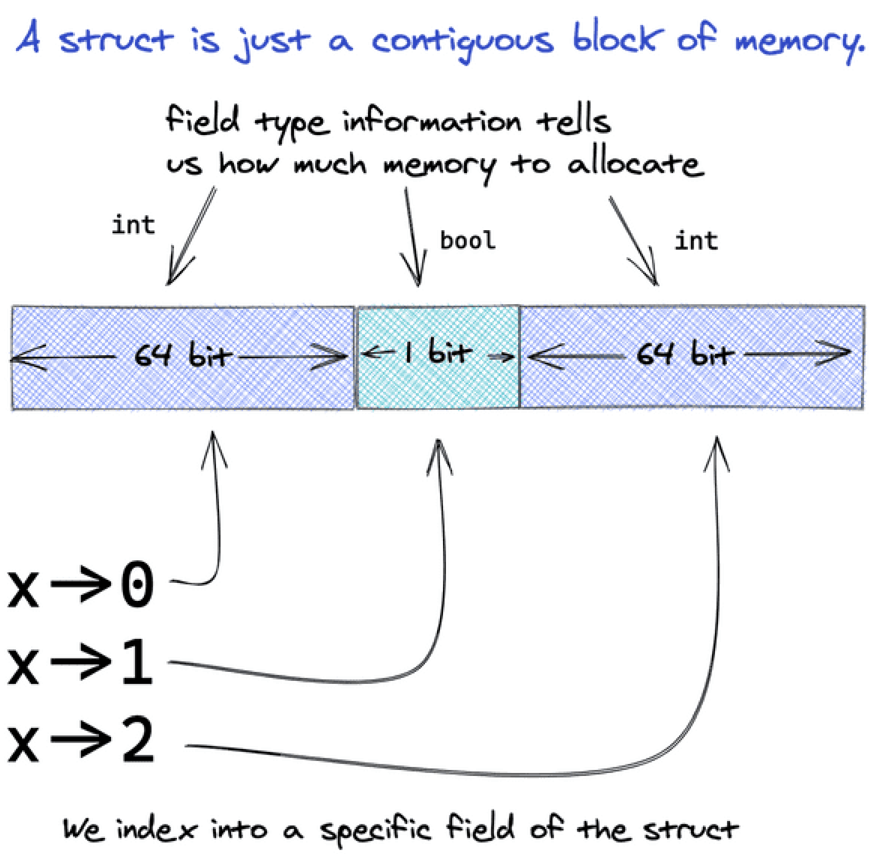
Recapping so far, we first looked at desugaring for loops - this occurs between the parsing and first stage of type-checking. We then looked at the desugaring stage which sits between the two stages of type-checking. We now look at IR lowering stage that occurs after type-checking.

IR stands for *intermediate representation* - it is simpler than the source code, but not quite lowered all the way down to assembly code.

Our goal with this IR is to **get close to the LLVM representation**, to make working with the LLVM API as simple as possible. We'll also strip away any unnecessary information that we won't need when running the program.

Lowering Objects to Structs

Classes are an **abstraction** that group together fields and methods. LLVM IR doesn't contain classes and objects, only **structs**, which are just a group of fields.



So how do we map from our Bolt class definition to a struct? We strip away information from our class:

- We dropped the `var` / `const` in the field definitions
- We dropped the capability annotations
- We drop type information in the AST except for field types
- We drop `loc` (line-position information that we used for our type-checker error messages).
- We drop field names
- We no longer associate methods with a class (more on that in a second!)

Recall, the goal of annotating types and capabilities to our AST is to check the program is correct. If we can assign a type to an expression, then it is *well-typed* so it satisfies our notion of correctness. Likewise, if we can assign capabilities then we know our program doesn't have data races. And `const` is just a compiler check to prevent us reassigning a field.

Once we've checked all that, we can drop that information, since we don't need it later in the compiler. In fact, our class definition is now quite barebones - just the class name (a `string`), and a list of field types, which LLVM will use to decide how much memory to allocate to an object:

[frontend_ir.mli]

Copy

```
type class_defn = TClass of string * type_expr list
```

Field names are useful to us as programmers, but for the computer we don't need to name our fields, we can number them instead as an **index** into the struct. Intuitively this is just like array indices.

[ir_gen_env.ml]

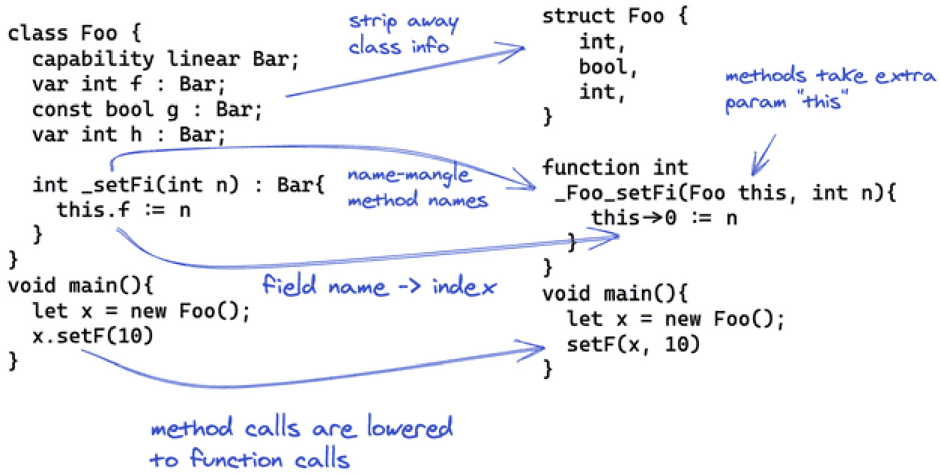
Copy

```
let ir_gen_field_index field_name class_name class_defns =
  get_class_fields class_name class_defns
  |> fun field_defns ->
  List.find_mapi_exn
    ~f:(fun index (TField (_, _, name, _)) ->
      if name = field_name then Some index else None)
    field_defns
```

> Note this `List.find_mapi_exn` function name might seem complex, but the goal is to `find` the field that matches the given field name by going through (`map`) each element of the list, along with that field's index (hence `mapi` not `map`), and raising an exception (`exn`) if it is not found. In practice, this function will never raise an exception because we already have checked in an earlier type-checking stage that the field exists.

Methods are just ordinary functions that implicitly take in an additional parameter: `this` , which refers to the object that called the method. In Python, this additional parameter (referred to as `self`) is explicitly declared in method declarations.

Note we need to name-mangle our methods again, by prepending the class name. Right now, we have unique method names *within* a class, when we separate them as normal functions, they need to be *globally* uniquely named.



Automatically Inserting Locks

Bolt has a `locked` capability, which is similar to the `synchronised` keyword in Java - this wraps locks around any access. Since we're dropping this locked capability we need to specify lock/unlock instructions in our IR.

[frontend_ir.mli]

Copy

```
type lock_type = Reader  | Writer

type expr =
  | Integer     of int
  | Boolean     of bool
  | Identifier  of identifier * lock_type option
  (* maybe acquire a lock when accessing an identifier *)
  ...
  | Lock        of string * lock_type
  | Unlock      of string * lock_type
```

and our to insert locks, we lock the object (`this`) and then compute the return value, release the lock and return the value:

Copy

```
... {
  methodBody
}
```

```
// adding locks
...{
  lock(this);
  let retVal = methodBody;
  unlock(this);
  retVal
}
```

The corresponding generation code is:

[ir_gen_class_and_function_defns.ml]

Copy

```
let ir_gen_class_method_defn class_defns class_name
    (Desugared_ast.TMethod
       ( method_name
       , _
       (* drop info about whether returning borrowed ref *)
       , return_type
       , params
       , capabilities_used
       , body_expr )) =
    ...
   |> fun ir_body_expr ->
  (* check if we use locked capability *)
  ( match
      List.find
        ~f:(fun (Ast_types.TCapability (mode, _))
        -> mode = Ast_types.Locked)
        capabilities_used
    with
  | Some _lockedCap ->
      [ Frontend_ir.Lock ("this", Frontend_ir.Writer)
      ; Frontend_ir.Let ("retVal", Frontend_ir.Block ir_body_ex
      ; Frontend_ir.Unlock ("this", Frontend_ir.Writer)
      ; Frontend_ir.Identifier (Frontend_ir.Variable "retVal",
  | None (* no locks used *) -> ir_body_expr )
  |> fun maybe_locked_ir_body_expr ->
  Frontend_ir.TFunction
    (ir_method_name, ir_return_type, ir_params, maybe_locked_ir
```

And for the identifiers, if we're meant to lock them, we acquire a Reader/Writer Lock depending on whether we're reading from them or assigning a value to them:

ir_gen_expr.ml

Copy

```
let rec ir_gen_expr class_defns expr =
...
  | Desugared_ast.Identifier (_, id) ->
      ir_gen_identifier class_defns id
      |> fun (ir_id, should_lock) ->
      let lock_held = if should_lock then Some Frontend_ir.Read
      Frontend_ir.Identifier (ir_id, lock_held)
...
  | Desugared_ast.Assign (_, _, id, assigned_expr) ->
      ir_gen_identifier class_defns id
      |> fun (ir_id, should_lock) ->
      ir_gen_expr class_defns assigned_expr
      |> fun ir_assigned_expr ->
      let lock_held = if should_lock then Some Frontend_ir.Writ
      Frontend_ir.Assign (ir_id, ir_assigned_expr, lock_held)
```

Wrapping Up Our Compiler Frontend

As mentioned in the previous parts, the Bolt repository also contains code for other language features (inheritance and generics, coming in a later post). So don't worry about "vtables" mentioned in the `ir_gen/` folder. To see a simpler version of the repository before these features were added, run `git checkout simple-compiler-tutorial` .

We've now wrapped up our discussion of the compiler frontend!

Next we're going to be switching from OCaml to C++ for the LLVM IR Code generation. To do this we'll be using Protobuf, a cross-language binary serialisation format.

Once we've done that, in a couple of posts we can talk about LLVM's C++ API!

Share This On Twitter

If you liked this post, please consider sharing it with your network. If you have any questions, tweet away and I'll answer :) I also tweet when new posts drop!

PS: I also share helpful tips and links as I'm learning - so you get them **well before** they make their way into a post!

Post Follow @mukulrathi_

SERIES: CREATING THE BOLT COMPILER

← A tutorial on liveness and alias dataflow analysis

A Protobuf tutorial for OCaml and C++ →