Creating the Bolt Compiler

Mukul Rathi

October 22, 2023

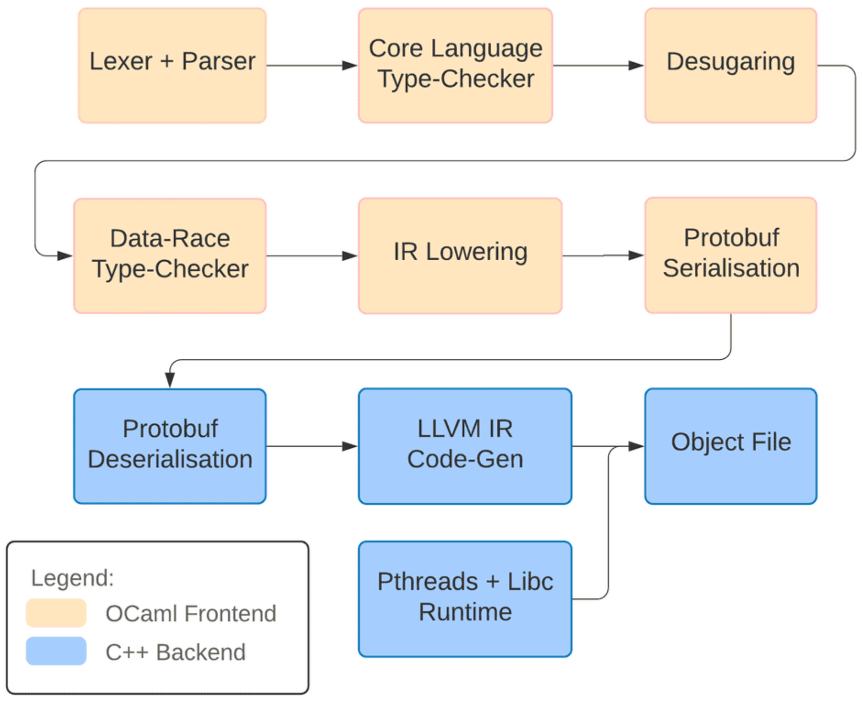
CONTENTS

Part I

INTRODUCTION

1 CREATING THE BOLT COMPILER: PART 1

May 10, 2020



The diagram above is the compiler for the language Bolt we'll be building. What do all the stages mean? I have to learn OCaml and C++? Wait I haven't even heard of OCaml…

Don't worry. When I started this project 6 months ago, I had never built a compiler, nor had I used OCaml or C++ in any serious project. I'll explain everything in due course.

In this series of posts we'll be building a *proper* programming language. One of the gripes I had when seeing programming language tutorials that created a toy language with only operations like addition and multiplication, was: okay, *but what about a real language like Java*?

So that's what this series aims to fix. The language Bolt I wrote as part of my third year disseration is a Java-style concurrent object-oriented language. Some of the highlights of this series:

- We implement **objects** and classes, with inheritance and method overriding
- **Concurrency** (as far as I could tell when writing this, no other programming language tutorial covered this)
- **Generics**: being able to write a class of type `LinkedList<T>` and then instantiating it with `LinkedList<int>`, `LinkedList<Person>` and so on.
- An introduction to how types are checked in a compiler
- Compiling to LLVM (this post was #2 on Hacker News!) – LLVM is used by C, C++, Swift, Rust amongst many other languages.

So I'd encourage you to follow the links in the "Series" overview to learn about these specific features. The remainder of this post will be to convince you why writing your own programming language is worthwhile, and the next post will outline the structure of a compiler.

1.1 WHY SHOULD YOU WRITE YOUR OWN PROGRAMMING LANGUAGE?

The question we should really be asking is *why design your own language*? Possible answers:

1. It's fun
2. It's cool to have your own programming language
3. It's a good side-project

1.2 MENTAL MODELS

Whilst all three of these (or none!) might be true, there's a bigger motivation: having the right **mental models**. See, when you learn your first programming language, you view programming through the lens of that language. Fast forward to your second language, and it seems hard, you have to relearn syntax and this new language does things differently. Using more programming languages, you realise that the languages share common themes. Java and Python have objects, Python and JavaScript don't require you to write types, the list goes on. Diving further into programming language theory, you read about the language constructs present - Java and Python are *object-oriented* programming languages and Python and JavaScript are *dynamically-typed*.

The programming languages you've been using actually build upon the ideas present in older languages that you may not have have heard of. Simula and Smalltalk introduced the concept of object-oriented programming languages. Lisp introduced the concept of dynamic typing. And there are newer research languages coming all the time that introduce new concepts. A more mainstream example: Rust builds **memory-safety** into a low-level systems programming language.

Building your own language (especially if you're adding new ideas) helps you think more **critically** about language design, so when you go learn a new language it's much easier. For example, I had never programmed in Hack before my internship at Facebook last summer, but knowing these programming language concepts made it much easier to pick up.

1.3 WHAT ARE COMPILERS?

So you've designed your fancy new language and it is going to revolutionise the world, but there's one problem. **How do you run it?** That's the role of a compiler. To explain how compilers work, let's first flash our minds back to the 19th Century, in the age of the telegraph. Here we have this fancy new telegraph but how do we send messages? **Same problem, different domain.** The telegraph operator needs to take in speech and convert it to Morse code, and tap out the code. The first thing the operator does is make sense of the speech - they split it into words (**lexing**) , and then understand how those words are used in a sentence (**parsing**) - are they part of a noun phrase, a subordinate clause etc. They check if it makes sense by classifying words into categories or **types** (adjective, noun, verb) and check the sentence makes grammatical sense (we can't use "runs" to describe a noun as it is a verb not a noun). Finally, they translate (**compile**) each word into dots and dashs (Morse code), which is then transmitted along the wire.

This seems like it labours the point, because so much of this is *automatic* for humans. Compilers work the same way, except we have to explicitly program computers to do this. The example above describes a simple compiler consisting of 4 stages: lex, parse, type-check and then translate into machine instructions. The operator also needs some additional tools to actually tap out the Morse code; for programming languages, this is the **runtime environment**.

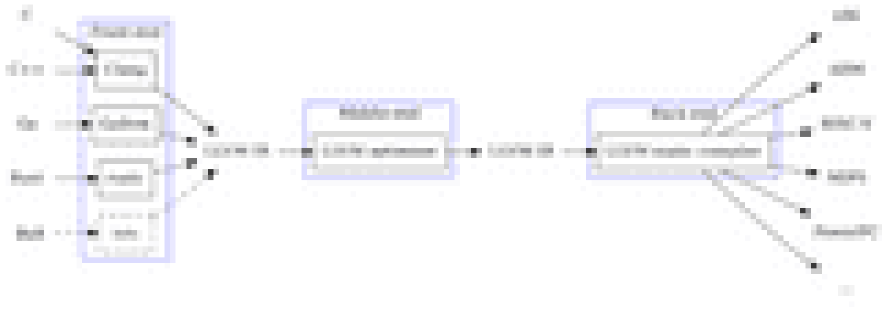
In practice, the operator likely constructs some shorthand notation that they know how to translate to Morse code. Now rather than converting speech into Morse code directly, they convert the speech into their shorthand, and then convert the shorthand into Morse code. In many practical languages, you can't just go directly from the source code to the machine code, you have *desugaring* or *lowering* stages, where you remove language constructs stage-by-stage (e.g. unrolling for loops) until we're left with a small set of instructions that can be executed. Desugaring makes later stages much easier, as they operate on a simpler representation. The compiler stages are grouped into frontend, middle-end and backend, where frontend does much of the parsing/type-checking, and middle-end and backend simplify and optimise the code.

1.3.1 *Compiler Design Choices*

We can actually frame a lot of language and compiler design in terms of the analogy above:

Does the operator translate words on-the-fly into Morse code as they transmit them, or do they convert the words into Morse code beforehand, and then transmit the Morse code? **Interpreted** languages like Python do the former, whilst ahead-of-time **compiled** languages like C (and Bolt) do the latter. Java actually lies somewhere in between - it uses a **just-in-time** compiler which does most of the work beforehand, translating programs to bytecode and then at runtime compiles bytecode to machine code.

Now consider a scenario where a new Lorse code came out that was an alternative to Morse code. If the operators are taught how to convert the shorthand to Lorse code, the person speaking doesn't need to know how that's done, they get it for free. Likewise, a person speaking a different language just needs to tell the operator how to translate it to the shorthand, and then they get translations into Lorse *and* Morse code! This is how **LLVM** works. **LLVM IR** (intermediate representation) acts as the stepping stone that lies between the program and the machine code. C, C++, Rust and a whole host of other languages (including Bolt) target LLVM IR, which then compiles code to a variety of machine architectures.



Static vs dynamic typing? In the first case, the operator either checks that the words make grammatical sense before they start tapping. Or, they don't and then midway through they're like "huh, this doesn't make sense" and stop. Dynamic typing can be seen as quicker to experiment in (like Python, JS) but when you send that message you don't know if the operator will stop midway through (crash).

I've explained it in terms of an imaginary telegraph operator, but any analogy works. Building up this intuition goes a long way in understanding which language features are right for your language: if you're going to be experimenting, then maybe dynamic typing is better as you can move faster. If you're using a larger codebase, it's harder to proof-read it all and you're more likely to make errors so you probably should shift towards static typing to avoid breaking things.

1.3.2 *Types*

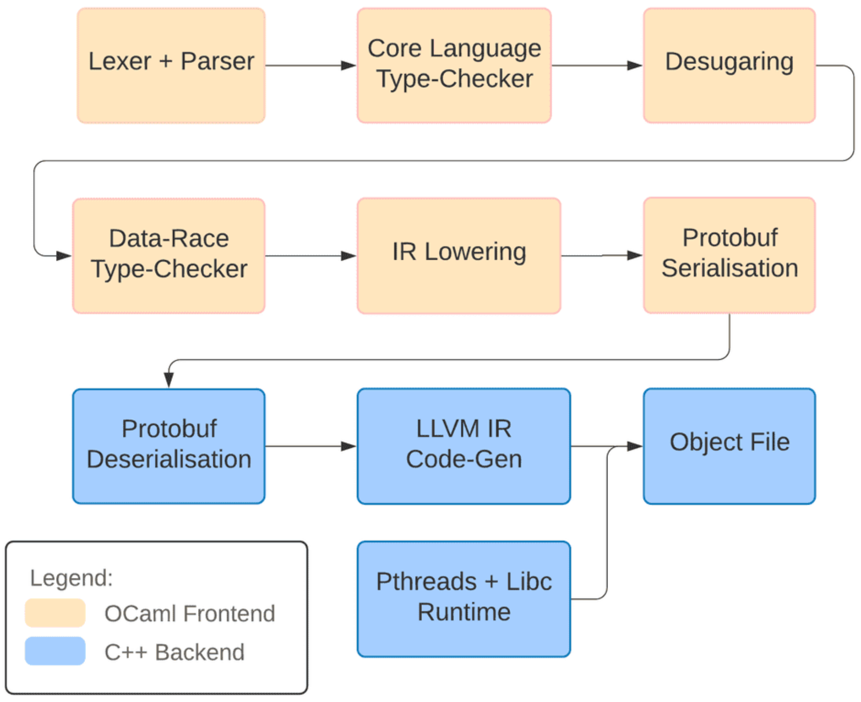
The most interesting part of the compiler (in my opinion) is the type-checker. In our analogy, the operator classified words as parts-of-speech (adjectives, nouns, verbs) then checked if they were used correctly. Types work the same way, we classify program values based on the behaviour we'd like them to have. E.g. `int` for numbers that can be multiplied together, `String` for streams of characters that can be concatenated together. The role of the type-checker is to prevent undesirable behaviour from happening - like concatenating `int`s or multiplying `String`s together - these operations make no sense so shouldn't be allowed. With type *checking*, the programmer annotates values with types, and the compiler checks if they're correct. With type *inference*, the compiler both infers and checks the types. We call the rules that check the types *typing judgements*, and a collection of these (along with the types themselves) forms a type system.

It turns out actually that there's a lot more you can do: type systems don't just check if `int`s or `String`s are used correctly. Richer type systems can prove stronger invariants about programs: that they will terminate, access memory safely, or that they do not contain data races. Rust's type system for example guarantees memory safety and data-race freedom, as well as checking traditional types `int`s and `String`s.

1.4 WHERE DOES BOLT FIT IN?

Programming languages still haven't cracked the problem of writing safe concurrent code. Bolt, like Rust, prevents data races (explained in this Rust doc), but takes a more fine-grained approach to concurrency. Before keyboard warriors come at me on Twitter, I think Rust has done a brilliant job in getting the conversation about this going - whilst Bolt will likely never go mainstream, it's demonstrating another approach.

If we look back at the pipeline now, you can see that Bolt contains the lexing, parsing, and desugaring/lowering phases. It also contains a couple of Protobuf serialisation and deserialisation phases: these are purely to convert between OCaml and C++. It targets LLVM IR, and then we link in a couple of runtime libraries (pthreads and libc) and finally we output our *object file*, a binary containing the machine code.



Unlike most compilers though, Bolt has not one but **two** type-checking phases! Bolt has both traditional types and **capabilities**, which are, informally, another set of types to type-check data races. I've written up a dissertation that explores this more formally, if you are interested in the theory, if not you can skip the data-race checking posts in this series. We type-check the traditional types first, simplify the language a bit in the desugaring stage, then do the data-race type-checking.

1.5 AND WHAT ABOUT THIS SERIES?

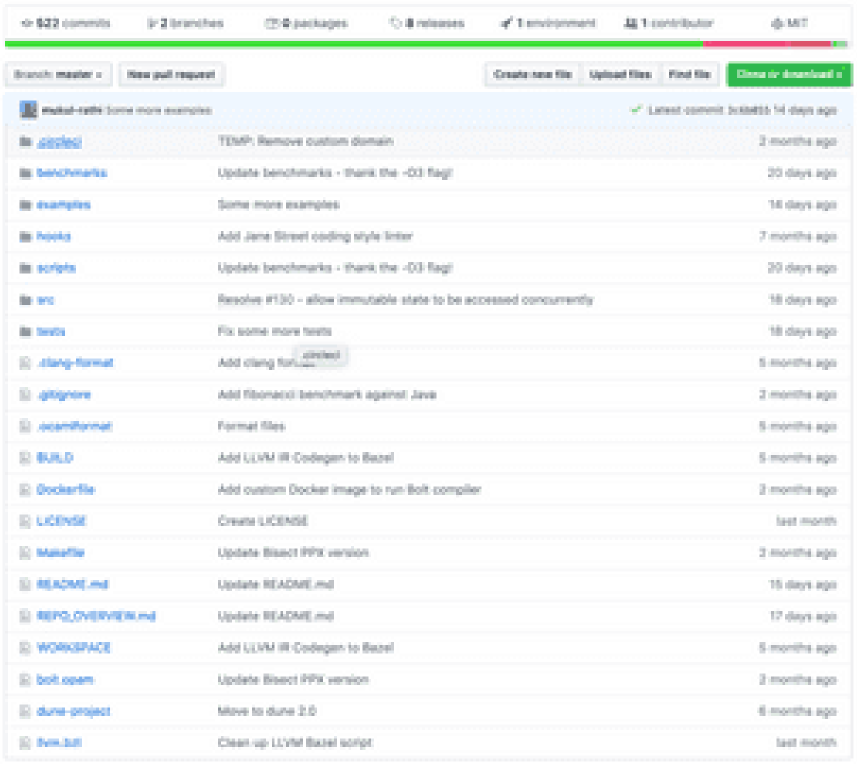
This series can be thought of from two perspectives: firstly, we will be discussing language design and comparing Bolt with Java, C++ and other languages along the way. Secondly, it is a practical step-by-step tutorial on building your own compiler. Unlike many build-your-own-compiler tutorials that tell you how to build a *toy* language, some of the topics this tutorial looks at form the basis of concurrent object-oriented languages like Java: how classes are implemented, how inheritance works, generic classes, and even how concurrency is implemented under the hood.

Bolt also doesn't output toy instructions but instead targets **LLVM IR**. Practically speaking, this means Bolt hooks into the amazing optimisations present in C/C++ compilers. The LLVM API is powerful, but it's also very hard to navigate the documentation. I spent many long nights reverse-engineering C++ programs - hopefully this series prevents at least one person from going through that pain!

In the next part, we'll look at the practical aspects of setting up a compiler project - I'll walk through the Bolt repository and explain *why* we're using OCaml of all languages for the frontend.

2 SO HOW DO YOU STRUCTURE A COMPILER PROJECT?

May 25, 2020



522 commits · 2 branches · 0 packages · 8 releases · 1 environment · 1 contributor · MIT

Branch: master · New pull request · Create new file · Upload files · Find file · Clone or download

mukul-rathi Some more examples · Latest commit 3c8d85 14 days ago

.circleci	TEMP: Remove custom domain	2 months ago
benchmarks	Update benchmarks - thank the -O3 flag!	20 days ago
examples	Some more examples	14 days ago
hooks	Add Jane Street coding style linter	7 months ago
scripts	Update benchmarks - thank the -O3 flag!	20 days ago
src	Resolve #130 - allow immutable state to be accessed concurrently	18 days ago
tests	Fix some more tests	18 days ago
.clang-format	Add clang for...	5 months ago
.gitignore	Add fibonacci benchmark against Java	2 months ago
.ocamlformat	Format files	5 months ago
BUILD	Add LLVM IR Codegen to Bazel	5 months ago
Dockerfile	Add custom Docker image to run Bolt compiler	2 months ago
LICENSE	Create LICENSE	last month
Makefile	Update Bisect PPX version	2 months ago
README.md	Update README.md	15 days ago
REPO_OVERVIEW.md	Update README.md	17 days ago
WORKSPACE	Add LLVM IR Codegen to Bazel	5 months ago
bolt.opam	Update Bisect PPX version	2 months ago
dune-project	Move to dune 2.0	6 months ago
llvm.bzl	Clean up LLVM Bazel script	last month

Writing a compiler is like any other software engineering project in that it involves a lot of key design decisions: what language do you use, how do you organise your files in the repo, which tools should you be using? Most compiler tutorials focus on a toy example and choose to ignore these practical concerns.

With *Bolt*, I'd like to highlight a larger compiler and the design decisions I've made. If you're reading this and you work on an industrial compiler for a more mature language, please reach out on Twitter! I'd love to hear about the design decisions you took!

2.1 USE THE RIGHT LANGUAGE FOR THE JOB, NOT JUST THE LANGUAGE YOU KNOW BEST

"Write a compiler using language Y" (insert your favourite language) tutorials are a dime a dozen. It might seem easier initially to write a compiler using a language you know, as it's one less thing to learn, but this is only a short-term gain. Choosing the correct language is like learning to touch-type: sure it will be slower to start with, but just think of how much faster you'll be once you've got to grips with it!

JavaScript is a great language for web apps and easy to pick up for beginners. But would I write a compiler in it? **Frankly, no.** I'm not hating on JavaScript (I use it in this very site), it just doesn't suit our goal.

What do we care about for compilers?

- **Coverage** - we need to consider all possible Bolt expressions and make sure we handle all cases - it's no good if our compiler crashes on Bolt programs we forgot to consider. Does our language help us keep track of this?
- **Data representation** - how do we represent and manipulate Bolt expressions in the compiler?
- **Tooling** - does our language have libraries we can use for our compiler? There's a balance between learning by doing and unnecessarily reinventing the wheel.
- **Speed** - there are two different aspects. Firstly, how fast is the compiled Bolt code? Secondly, how fast is the compiler (how long does it take to compile the Bolt code)? There's a tradeoff - to get faster compiled code, you need to include more optimisation steps in your compiler, making the compiler slower.

There's no silver bullet: each compiler design inherently has its tradeoffs. I chose to primarily write my compiler in OCaml.

2.2 WHY OCAML?

OCaml is a functional programming language with a powerful type system. You probably have two questions: why functional programming, and what do I mean by powerful type system?

In a large compiler, there's a lot of moving parts and keeping track of state makes our lives harder. Functional programming is easier to reason about: if you pass a function the same input it will always return the same output. With functional programming we don't have to worry about **side-effects** or state, and it lets us focus on the high-level design.

Another alternative is to write the compiler in Rust for performance reasons. Whilst you might have a faster compiler, I don't think the speed justifies the additional low-level details like managing memory that Rust requires you to track. Personally, since I'm not writing thousands of lines of Bolt code, I'm not too concerned with how long the Bolt compiler takes to compile programs.

2.2.1 *Types let you pair program with the compiler*

If you're coming from a dynamically typed language like JS or Python, OCaml's rich types can feel alien and may feel cumbersome. The way I think about types in OCaml is that they give the OCaml compiler more information about your program - the more you tell it, the more it can help you!

Coming back to our program of coverage, what we want to say is that a Bolt expression is *either* an integer, an if-else expression, a method call, a while loop etc. Normally to represent something like this, you would use an `enum` and a `switch` statement. In OCaml we bake this "enum" into our type system using *variant types*. We can encode the structure of each expression within the type! For example, to access a variable you only need to know its name `x`. To access an object's field you need to know both its name `x` and the field you're accessing `f`. We can then *pattern-match* based on each case: think of this like a `switch` statement on steroids!

Copy

```
type identifier =| Variable of Var_name.t| ObjField of
    ↪ Var_name.t * Field_name.t
let do_something id = match id with| Var(x) -> ...| ObjField
    ↪ (x,f) -> ...
```

I haven't even mentioned the **best part**. Because we've encoded our Bolt expression structures in a type, the OCaml compiler will check if we've covered all cases! That's one less thing for us to track!

So OCaml takes care of coverage, and we've decided that we'll encode Bolt expressions as variant types, so that's the data representation sorted. OCaml also has great tooling for the lexer and parser stages of the compiler (discussed in the next post) which ticks off another of our criteria.

2.3 TARGETING PERFORMANCE WITH LLVM

We touched upon the fact that we don't really care about the performance of the compiler itself. However, we do want our compiled Bolt code to be fast (*it's in the name!*). As touched on in the previous post, we don't have to reinvent the wheel. By targeting *LLVM IR*, we can hook into the C/C++ toolchain and then get our optimisations for free!

LLVM provide APIs for language authors to generate LLVM IR – the *native* API is in C++. LLVM also offers bindings in other languages – they are identical, just replace the C++ syntax with whichever language you're using. LLVM actually offers OCaml bindings.

2.3.1 *Why is the compiler backend written in C++?*

A natural question you might ask is: well why didn't you write everything in OCaml and use the LLVM OCaml bindings?

LLVM's OCaml bindings only map some of the C++ API. At the time of implementation there was no support for implementing memory fences (a machine instruction needed to implement locks correctly) so I was forced to write this part of the compiler in C++. I was also experimenting with some other fancier memory consistency instructions only present in the native C++ API.

I want to be frank with you, the OCaml LLVM bindings will likely be sufficient for your language and I **encourage you to use that instead**. See, the tradeoff with the approach I took is that now we have to pass data between the OCaml compiler *frontend* and the C++ compiler *backend* using Protobuf. Yes the C++ API has more power, but it results in a more complex compiler.

Like I said, the LLVM API is the same in OCaml and C++ (apart from syntax), so this tutorial still applies, just skip the Protobuf post and use the llvm OCaml package!

2.4 SOFTWARE ENGINEERING METHODOLOGY

The repository contains more information about how the compiler is structured in REPO_OVERVIEW.md. Most of these are general software engineering tips so I'll keep this brief. (If you want more detail, feel free to reach out!)

Firstly, the repository is structured to be *modular*. Each stage of the compiler has its own library, whose documentation you can view. Functions are grouped into *modules* (the `.ml` file provides the implementation for the module, and the `.mli` file provides the module interface). You can think of each module as performing a certain role within a stage e.g. `type_expr.ml` type-checks expressions, `pprint_parser_tokens.ml` pretty-prints parser tokens. It makes each module more focused, and avoids monolithic hundreds-of-lines-long files that are hard to read.

To build these files I use the Dune build system for OCaml (I have a blog post explaining it) and the Bazel build system for C++. For large repositories, manually compiling each file (e.g. by running `clang++ foo.cpp`) and linking files that depend on each other is nigh on impossible - build systems automate this for us (running one `make build` command will compile all the files in the repository). One of the main benefits of Bazel is that the dependencies are all self-contained so will work across machines.

For testing, the main library I use is the Jane Street's **Expect tests** library. This library is *really easy* to write tests for as it autogenerates the expected output. I have a blog post on testing in OCaml that covers this. The post also explains how I set up Continuous Integration (where the tests are run and documentation is generated on each commit to the repository).

I also automate common tasks using a `Makefile` and some scripts. One tool I would highly recommend is an autoformatter - I use `ocamlformat` for the OCaml code and `clang-format` for the C++ code - this formats your code for you so you have pretty looking code for free! You can automate it with the git pre-commit hook in the repo (which will lint and format your code every time you're about to commit) or via IDE format-on-save integration. One final tip: use VSCode's OCaml IDE extension or the equivalent for your IDE - whenever you hover over a function it will display its type signature and any documentation comments associated with it.

2.5 SUMMARY

In these first couple of posts, I've explained where Bolt sits in the spectrum of programming languages, and the compiler design and software engineering decisions made. In the next post, we'll actually get to building this. Before we do, I have a couple of action items for you:

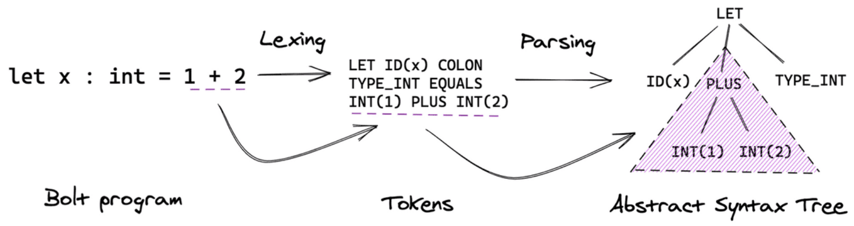
- Get up to speed with OCaml. Real World OCaml is a great free resource.
- Fork the repo. Have a quick high-level scan but don't worry about the details just yet! We'll break down each stage of the compiler in its own post, and dedicate entire posts to the more complex language features.

Part II

OCAML SEMANTIC FRONT END

3 WRITING A LEXER AND PARSER USING OCAMLLEX AND MENHIR

June 01, 2020



We can't directly reason about our Bolt program, as it is just an unstructured stream of characters. Lexing and parsing *pre-processes* them into a structured representation that we can perform type-checking on in later stages of the compiler.

3.1 LEXING TOKENS

The individual characters don't mean much, so first we need to split the stream into *tokens* (which are analogous to "words" in sentences). These tokens assign meaning: is this group of characters a specific keyword in our language (`if int class`) or is it an identifier (`banana`)?

Tokens also reduce our problem space substantially: they standardise the representation. We no longer have to worry about whitespace (`x == 0` and `x== 0` both become `IDENTIFIER(x) EQUAL EQUAL INT(0)`) and we can filter out comments from our source code.

How do we split our stream of characters into tokens? We *pattern-match*. Under the hood, you could think of this as a massive case analysis. However, since the case analysis is ambiguous (multiple tokens could correspond to the same set of characters) we have two additional rules we have to consider:

1. **Priority order:** We order our tokens by priority. E.g. we want `int` to be matched as a keyword, not a variable name.
2. **Longest pattern match:** we read `else` as a keyword rather than splitting it into two variable names `el` and `se`. Likewise, `intMax` is a variable name: not to be read as `int` and `Max`.

Here's a really simple lexer that recognises the keywords "IN", "INT" and identifiers (variable / function names), and tokens are separating by spaces. Notice we check for the IN case before the "default" variable case (priority order):

Copy

```
// this is pseudocode for a simplified lexerchars
SeenSoFar = "in"while(streamHasMoreCharacters){  nextChar = readCharFromStream()  if(nextChar == " "){
```

3.2 OCAMLLEX

Whilst you could hand-code the pattern-matching pseudocode, in practice it is quite finicky especially as our language gets bigger. Instead, we're going to use the lexer generator **OCamllex**. OCamllex is an OCaml library we can use in our compiler by adding it as a dependency to our Dune build file.

```
(ocamllex lexer)
```

The specification for the lexer is in the `lexer.mll` file (note the `.mll` file extension).

3.2.1 *OCaml Header*

To start with, we optionally provide a header containing OCaml helper code (enclosed in curly braces). We define a `SyntaxError` exception and a function `nextline`, which moves the pointer to the next line of the `lexbuf` buffer that the program is read into:

Listing 3.1: lexer.mll

```
{open Lexingopen Parser
exception SyntaxError of string
let next_line lexbuf =  let pos = lexbuf.lex_curr_p in
  ↪ lexbuf.lex_curr_p <-    { pos with pos_bol = lexbuf.
  ↪ lex_curr_pos;                pos_lnum = pos.pos_lnum +
  ↪  1    }}
```

3.2.2 *Helper Regexes*

Next, we need to specify the regular expressions we're using to match the tokens. For most of the tokens, this is a simple string e.g. `true` for token `TRUE`. However, other tokens have more complex regexes e.g. for integers and identifiers (below). OCamllex's regex syntax is like most regex libraries;

Listing 3.2: lexer.mll

```
(* Define helper regexes *)
let digit = ['0'-'9']let alpha = ['a'-'z' 'A'-'Z']
let int = '-'? digit+  (* regex for integers *)
let id = (alpha) (alpha|digit|'_')*
(* regex for identifier *)
let whitespace = [' ' '\t']+
let newline = '\r' | '\n' | "\r\n"
```

3.2.3 *Lexing Rules*

Next, we need to specify rules for OCamllex to scan the input. Each rule is specified in a pattern-matching format, and we specify the regexes in order of priority (highest priority first):

```
rule <rule_name> = parse | <regex> { TOKEN_NAME } (* output a token *)
                         | <regex>  { ... } (* or execute other code *)
and <another_rule> = parse  | ...
```

The rules are recursive: once it matches a token, it calls itself to start over and match the next token. Multiple rules are *mutually recursive*, that is we can recursively call each rule in the other's definition. Having multiple rules is useful if you want to treat the character stream differently in different cases.

For example, we want our main rule to read tokens. However, we want to treat comments differently, not emitting any tokens until we reach the end of the comment. Another case are strings in Bolt: we want to treat the characters we are reading as part of a string, not as matching a token. These cases can all be seen in the following Bolt code:

example.bolt

```
let x = 4 // here is a comment
/* This is a multi-line
comment*/
printf("x's value is %d", x)
```

Now we have our requirements for our lexer, we can define the rules in our OCamllex specification file. The key points are:

- We have 4 rules: `read_token`, `read_single_line_comment`, `read_multi_line_comment` and `read_string`.
- We need to handle eof explicitly (this signifies the end of file) and include a catch-all case _ to match all other regexes.
- We use `Lexing.lexeme lexbuf` to get the string matched by the regex.
- For `read_string`, we create another buffer to store the characters into: we don't use `Lexing.lexeme lexbuf` as we want to explicitly handle escaped characters. `Buffer.create 17` allocates a resizable buffer that initially has a size of 17 bytes.
- We use `raise SyntaxError` for error-handling (unexpected input characters).
- When reading tokens, we skip over whitespace by calling `read_token lexbuf` rather than emitting a token. Likewise, for a new line we call our helper function `next_line` to skip over the new line character.

Listing 3.3: lexer.mll

```
rule read_token =  parse
                 | "(" { LPAREN }  ... (* keywords and other
                     ↪ characters' regexes *)
                 | "printf" {PRINTF }
                 | whitespace { read_token lexbuf }
                 | "//" { single_line_comment lexbuf (* use
                     ↪ our comment rule for rest of line *)
                     ↪ }
                 | "/*" { multi_line_comment lexbuf }
                 | int { INT (int_of_string (Lexing.lexeme
                     ↪ lexbuf))}
                 | id { ID (Lexing.lexeme lexbuf) }
                 | '"'      { read_string (Buffer.create 17)
                     ↪ lexbuf }
                 | newline { next_line lexbuf; read_token
                     ↪ lexbuf }
                 | eof { EOF }
                 | _ {raise (SyntaxError ("Lexer - Illegal
                     ↪ character: " ^ Lexing.lexeme lexbuf))
                     ↪  }
```

```
and read_single_line_comment = parse
                             | newline { next_line lexbuf;
                                 ↪ read_token lexbuf }
                             | eof { EOF }
                             | _ { read_single_line_comment
                                 ↪ lexbuf }
and read_multi_line_comment = parse
                            | "*/" { read_token lexbuf }
                            | newline { next_line lexbuf;
                                ↪ read_multi_line_comment
                                ↪ lexbuf }
                            | eof { raise (SyntaxError ("
                                ↪ Lexer - Unexpected EOF -
                                ↪ please terminate your
                                ↪ comment.")) }
                            | _ { read_multi_line_comment
                                ↪ lexbuf }
and read_string buf = parse
                    | '"'       { STRING (Buffer.contents
                        ↪ buf) }
                    | '\\' 'n'  { Buffer.add_char buf '\n';
                        ↪ read_string buf lexbuf }
                    ... (* Other regexes to handle
                          ↪ escaping special characters *)
                    | [^ '"' '\\']+    { Buffer.add_string
                        ↪ buf (Lexing.lexeme lexbuf);
                                         read_string buf
                                             ↪ lexbuf    }
                    | _ { raise (SyntaxError ("Illegal
                        ↪ string character: " ^ Lexing.
                        ↪ lexeme lexbuf)) }
                    | eof { raise (SyntaxError ("String is
                        ↪ not terminated")) }
```

3.2.4 *Generated OCamllex output*

OCamllex generates a Lexer module from the lexer.mll specification, from which you can call Lexer.read_token or any of the other rules, as well as the helper functions defined in the header. If you're curious, after running make build you can see the generated module in lexer.ml in the _build folder – it's just a massive pattern-match statement:

Listing 3.4: _build/..../lexer.ml

```
let rec read_token lexbuf =   __ocaml_lex_read_token_rec
    ↪ lexbuf 0and __ocaml_lex_read_token_rec lexbuf
    ↪ __ocaml_lex_state =  match Lexing.engine
    ↪ __ocaml_lex_tables __ocaml_lex_state lexbuf  with| 0
    ↪ -># 39 "src/frontend/parsing/lexer.mll"          (
    ↪ LPAREN )# 2603 "src/frontend/parsing/lexer.ml"
  | 1 -># 40 "src/frontend/parsing/lexer.mll"          (
      ↪ RPAREN )# 2608 "src/frontend/parsing/lexer.ml"
  | 2 -># 41 "src/frontend/parsing/lexer.mll"          (
      ↪ LBRACE )# 2613 "src/frontend/parsing/lexer.ml"
  | 3 -># 42 "src/frontend/parsing/lexer.mll"          (
      ↪ RBRACE )# 2618 "src/frontend/parsing/lexer.ml"
    | 4 ->#  43 "src/frontend/parsing/lexer.mll"         ( COMMA
       ↪  )# 2623 "src/frontend/parsing/lexer.ml"
```

3.3 GRAMMAR

We use structure to reason about sentences – even if we don't think about it. The words on their own don't tell us much about the sentence – "use" is both a noun and verb – it's only because we have a subject-verb-object pattern in "we use structure" can we infer that "use" is a verb. The same is true for compilers: The individual tokens `x`, `+`, `y` don't mean much, but we can infer from `x+y` that `x` and `y` are numbers being added together.

We specify the structure of programs in Bolt using a *grammar*, which consists of a set of rules (*productions*) about how we can construct Bolt expressions.

For example, a program consists of a list of class definitions, following by some function definitions following by the main expression. This would look like this (using "X_defns" plural to informally refer to a list of "X_defn").

program ::=::= **class_defns function_defns main_expr**

This top level rule also has rules for each of the expressions on the right hand side. Expressions that can be expanded further with rules are called *non-terminals*, and those that can't i.e. the tokens are called *terminals*. Let's look at the **class_defn** rule and **main_expr** rules.

A class definition consists of a `CLASS` keyword token (tokens are in UPPER-CASE) followed by an `ID` token (identifier = the class name) and then the body is enclosed by braces. The body consists of capability definitions (this is Bolt-specific – used to prevent data races), field definitions and then method definitions. Here the `CLASS`, `ID`, `LBRACE` and `RBRACE` tokens are *terminals*, and the `capability_defn`, `field_defns` and `method_defns` expressions are *non-terminals* (they have their own rules expanding them).

class_defn ::=::= **CLASS ID LBRACE capability_defn field_defns method_defns RBRACE**

A class definition that satisfies the rules:

Listing 3.5: class_example.bolt

```
class Foo {
    // Foo is the identifier
    capability linear Bar; // capability definition (has its
        ↪  own rule)
    // field defns
    var int f : Bar;
    // (field has Bolt-specific capability annotatation)
    const bool g : Bar;
    //method defns
    int getF() : Bar {
       // method definition (again has its own rule)    this
           ↪  .f
    }
}
```

And our main expression has the following rule:

main_expr ::=::= **TYPE_VOID MAIN LPAREN RPAREN block_expr**

```
void main() {  ...}
```

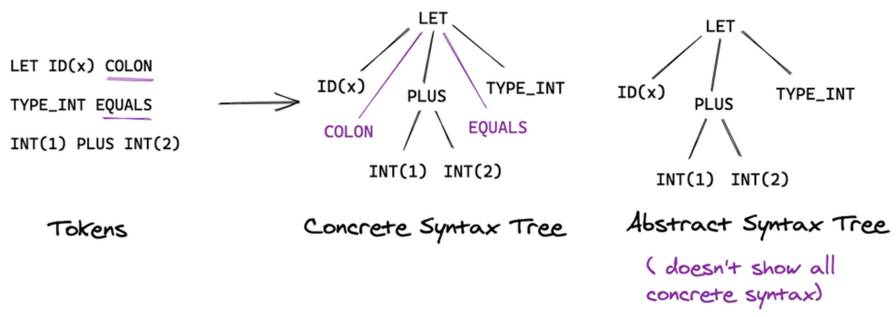
Expressions can have multiple forms:
expr ::=::=
| NEW ID LPAREN constructor_args RPAREN
| IF expr block_expr ELSE block_expr
| LET ID EQUAL expr
and so on.
Full details of Bolt's grammar are in the appendix of my dissertation.

3.4 ABSTRACT SYNTAX TREES

If we look at this grammar from another angle, this actually specifies a hierarchy on our program structure. At the top-level you have our **program** rule, and then we recursively expand out each of the terms on the right-hand-side of the rule (e.g. expand out the class definition using its rule, main expression using its rule etc.) until we end up with the tokens.

Which data structure represents hierarchies? *Trees*. So the grammar also specifies a *syntax tree* for our Bolt program, where tokens are the leaves.

We could display all tokens on our tree (this would be a *concrete syntax tree*) but in practice not all tokens are useful. For example, in the expression `let x : int = 1+2`, you care about the `ID` token (variable `x`) and the expression `1+2` being assigned to the variable. We don't need to represent the `=` token in our tree, as it doesn't convey additional meaning (we already know we're declaring a variable). By removing these unnecessary tokens, we end up with an *abstract syntax tree* (AST).



The goal of parsing is to construct an Abstract Syntax Tree from a Bolt program. The subsequent compiler stages then analyse and transform this parsed AST to form other intermediate ASTs.

Here we split the type definitions for our AST into two files:
`ast_types.mli` contains types common to *all* ASTs across the compiler.

Listing 3.6: ast_types.mli

```
(** Stores the line and position of the token *)
type loc = Lexing.position
type bin_op =  | BinOpPlus  | BinOpMinus  | BinOpNotEq
type modifier = MConst  (** Immutable *)
               | MVar  (** Mutable *)
(** An abstract type for identifiers *)
module type ID = sig  type t
  val of_string : string -> t
  val to_string : t -> string
  val ( = ) : t -> t -> boolend
module Var_name : ID
module Class_name : ID
(** Define types of expressions in Bolt programs*)
type type_expr =  | TEInt  | TEClass   of Class_name.t  |
    ↪ TEVoid  | TEBool
  ...
```

parsed_ast.mli contains the OCaml variant types for the class definitions, function defns and expressions, which are essentially a mapping of the grammar rules.

Listing 3.7: parsed_ast.mli

```
type expr =  | Integer     of loc * int  ...
             | Constructor of loc * Class_name.t * type_expr
                ↪  option * constructor_arg list  (*
                ↪ optional type-parameter *)
             | Let         of loc * type_expr option *
                ↪ Var_name.t * expr  (* binds variable to
                ↪ expression (optional type annotation) *)
             | Assign      of loc * identifier * expr
             | MethodApp   of loc * Var_name.t * Method_name
                ↪ .t * expr list (* read as x.m(args) *)
             | If          of loc * expr * block_expr *
                ↪ block_expr (* If ___ then ___ else ___
                ↪ *)
             | BinOp       of loc * bin_op * expr * expr
                ↪ ...and block_expr = Block of loc * expr
                ↪ list
type class_defn =  | TClass of Class_name.t * capability
    ↪ list  * field_defn list * method_defn list...
```

Note Class_name.t, Var_name.t and Method_name.t used rather than just a string as these types give us more information.

3.5 MENHIR

As with lexing, we *could* code this up by hand, but it would again get a lot more finicky as our language scales. We're going to use the parser generator **Menhir**. Again as with the lexer, we have a parser.mly (note .mly extension) specification file.

We need to add Menhir to the Dune build file.

```
(menhir (modules parser))
```

Unlike with OCamllex, we also need to update our main Dune project build file to use Menhir:

Listing 3.8: dune-project

```
(using menhir 2.0)
```

OCamllex and Menhir have a lot of parallels. We start with our optional OCaml header (here this just ignores the autogenerated parser file when calculating test coverage and opens the Ast_types and Parsed_ast modules):

Listing 3.9: parser.mly

```
%{
  [@@@coverage exclude_file]
  open Ast.Ast_types
  open Parsed_ast
%}
```

We then specify the tokens we'd defined in OCamllex (OCamllex and Menhir work seamlessly together):

Listing 3.10: parser.mly

```
%token  <int> INT
%token  <string> ID
%token  LPAREN
%token  RPAREN...
```

3.5.1 *Specifying Production Signatures*

Next, we specify the name of the top-level grammar production (here it is `program`):

Listing 3.11: parser.mly

```
%start program
```

We mentioned earlier that there was a mapping between OCaml variant types and the grammar productions, now we specify this, so Menhir knows what type to output when it generates the parser. This is done as a series of `%type <ast_type> production_name`:

Listing 3.12: parser.mly

```
%type <Parsed_ast.program> program
/* Class defn types */
%type <class_defn> class_defn...
%type <Capability_name.t list> class_capability_annotations
```

Note, since we opened the `Ast_types` and `Parsed_ast` modules in the header, we can write `Capability_name.t` rather than `Ast.Ast_types.Capability_name.t` and `class_defn` instead of `Parsed_ast.class_defn`. However, we need to specify absolute types for the top-level production (`Parsed_ast.program` not just `program`) since this production is exposed in the `parser.mli` interface:

Listing 3.13: parser.mli

```
(* this file is autogenerated by Menhir *)val program: (
    ↪ Lexing.lexbuf -> token) -> Lexing.lexbuf -> (
    ↪ Parsed_ast.program)
(* if we used program not Parsed_ast.program *)val program:
    ↪ (Lexing.lexbuf -> token) -> Lexing.lexbuf -> (program
    ↪ )(* now parser.mli doesn't know where to find program
    ↪ 's type definition *)
```

3.5.2 *Grammar Rules*

Finally, we can specify our grammar rules, with the returned OCaml variant expression in {} after the rule. We can optionally separate each of the non-terminal/terminals that make up the right-hand-side of the rule with semicolons for readability. To use the result of an expanded non-terminal (e.g. the class_defn) in the OCaml expression, we can refer to it with a variable class_defns (in general the form is var_name=nonterminal expression).

Listing 3.14: parser.mly

```
program:| class_defns=list(class_defn); function_defns=list(
    ↪ function_defn); main= main_expr;  EOF {Prog(
    ↪ class_defns, function_defns, main)}
```

Menhir provides a lot of small helper functions that make writing the parser easier. We've seen list(class_defn) - this returns the type Parsed_ast.class_defn list. There are other variants of this e.g. separated_list() and nonempty_list() and separated_nonempty_list():

Listing 3.15: parser.mly

```
params:| LPAREN; params=separated_list(COMMA,param); RPAREN
    ↪ {params}
param_capability_annotations:| LBRACE;  capability_names=
    ↪ separated_nonempty_list(COMMA,capability_name);
    ↪ RBRACE {capability_names}
```

Another useful function is option(). Here since let_type_annot returns an OCaml expression of type Parsed_ast.type_expr, option(let_type_annot) returns a value of type Parsed_ast.type_expr option - this is useful for cases where the user might optionally add type annotations. e.g. let x : int = 5 vs let x = 5.

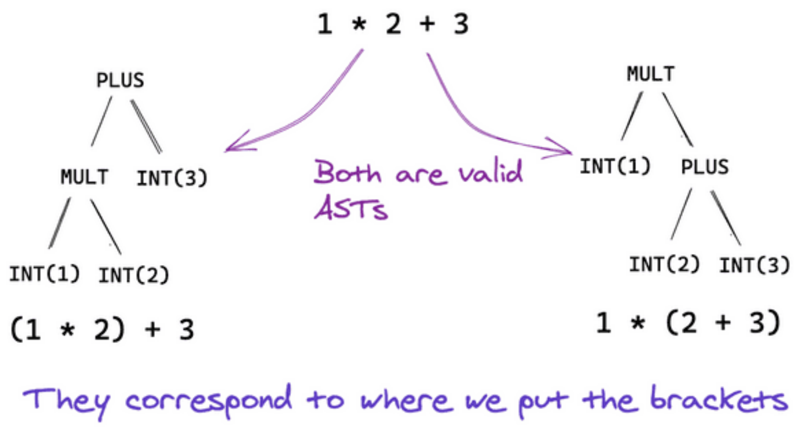
Finally, Menhir also lets us get the line number and position of the production in the program (we'd defined the type loc for this), which is useful for error messages! You can decide from where to get the position: I chose to get the position at the start of the production using $startpos.

```
expr:| i=INT {Integer($startpos, i)}
     | TRUE { Boolean($startpos, true)}
     | FALSE {  Boolean($startpos, false) }...+
     | e1=expr op=bin_op e2=expr {BinOp($startpos, op, e1,
         ↪ e2)}
     | LET; var_name=ID; type_annot=option(let_type_annot);
         ↪  EQUAL; bound_expr=expr  {Let($startpos,
         ↪ type_annot, Var_name.of_string var_name,
         ↪ bound_expr)}
     | id=identifier; COLONEQ; assigned_expr=expr {Assign(
         ↪ $startpos, id, assigned_expr)}
```

3.5.3 *Resolving ambiguous parses*

Sometimes our grammar can produce *multiple* abstract syntax trees, in which case we say it is *ambiguous*. The classic example is if we have the following grammar:

expr ::=::= | INT | expr PLUS expr | expr MULT expr



If we try to build this grammar with Menhir, then we get the following error message (the numbers aren't important, Bolt has many more operators than just + and –):

```
menhir src/frontend/parsing/parser.{ml,mli}
Warning: 17 states have shift/reduce conflicts.
Warning: 187 shift/reduce conflicts were arbitrarily resolved.
```

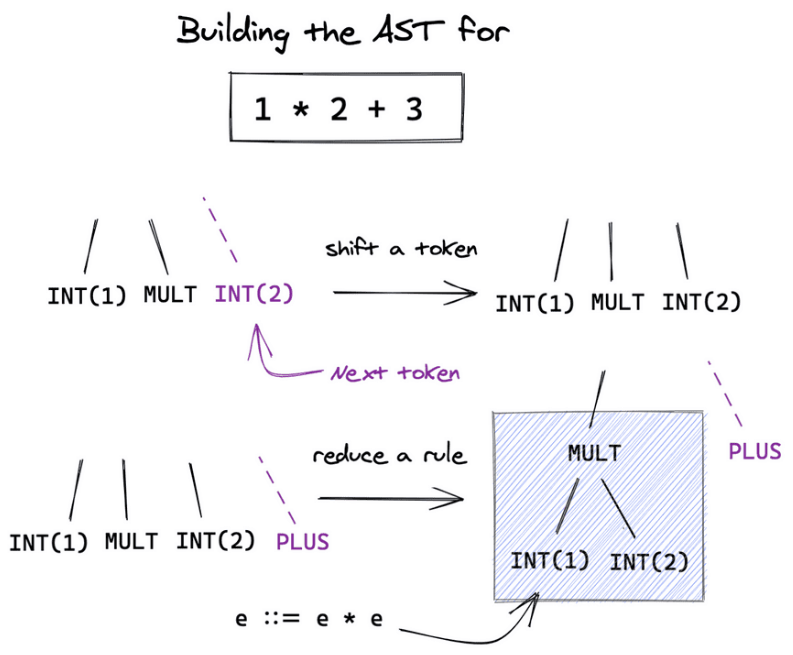
Arbitrarily resolved isn't good! This means it'll randomly pick one, even if it's not the one we want. To fix this we have two options

1. Rewrite the grammar to be unambiguous.
2. Keep the ambiguity but tell Menhir how to resolve it.

I had initially chosen option 1 for Bolt. It is possible to do this for small grammars but this becomes harder as your language gets bigger. The main disadvantage is that you have to put extra parentheses around Bolt expressions to disambiguate parses, which really isn't a good experience.

When writing this post, I looked at the OCaml compiler for inspiration (it too uses Menhir!) and the Menhir docs. Option 2 offers a better solution, but first we need to understand how Menhir works.

Menhir is a *shift-reduce* parser. It builds our AST bottom-up, deciding based on the next token whether to *reduce* the current stack of non-terminals and terminals (i.e. it matches the right-hand-side of a production rule, so reduce it to the left-hand-side) or to *shift* another token onto the stack.



A *shift-reduce conflict* occurs when it could do both and end up with a valid AST in either case. Menhir lets us manually specify which action to take in the form `<action> token` whether the action is:

1. `%left`: reduce
2. `%right`: shift
3. `%nonassoc`: raise a SyntaxError

If you're not sure which one to choose, you have a secret weapon!

Running `menhir src/frontend/parsing/parser.mly --explain` will generate an explanation in a `parser.conflicts` file, which gives an in-depth explanation as to where the conflict is!

This is only part of the story – we want to specify that multiplication takes precedence over addition. We can do this by specifying an *ordering* on the actions, from lowest priority to highest. Then when there are multiple rules, Menhir will choose the one with the highest priority – it will reduce the multiplication before the addition in our example, giving us the right AST:

```
%right  COLONEQ EQUAL
%left PLUS MINUS
%left MULT DIV REM
%nonassoc EXCLAMATION_MARK
```

So we're done right? Not quite. If we have multiple binary operators, we'd write this grammar instead as:

expr ::=::= **| INT | expr binop expr**

binop ::=::= **| PLUS | MINUS | MULT | DIV | REM | . . .**

After following the steps outlined, you might still get the following error:

```
Warning: the precedence level assigned to PLUS is never useful.
```

Or that you still have shift-reduce conflicts, even though you've resolved them all. What's gone wrong?

See I said this precedence works if there are *multiple rules*, not if there is one production. Here we just have the one (**expr binop expr**). What we want is one rule for each of the operators (**expr PLUS expr**) (**expr MULT expr**) etc. Menhir's got you covered - just add an `%inline` keyword. This expands all the uses of `bin_op` below to one rule per variant:

```
%inline bin_op:
| PLUS { BinOpPlus }
| MINUS { BinOpMinus }
| MULT { BinOpMult }...
```

Et voila, we're done! No more shift-reduce conflicts.

3.6 PUTTING THE LEXER AND PARSER TOGETHER

Right, let's wrap up the `src/parsing` folder of the Bolt repository by talking about how we'd put this all together. What we want is the following:

Listing 3.16: lex_and_parse.mli

```
(** Given a lex buffer to read a bolt program from, parse
    ↪ the
    program and return the AST if successful *)
val parse_program : Lexing.lexbuf -> Parsed_ast.program
    ↪ Or_error.t
```

To do that, we'll need to use the `Lexer` and `Parser` modules generated by OCamllex and Menhir from our `lexer.mll` and `parser.mly` files. Specifically we care about the two functions:

```
val Lexer.read_token: Lexing.lexbuf -> token
val Parser.program: (Lexing.lexbuf -> token) -> Lexing.
    ↪ lexbuf
                         -> (Parsed_ast.program)
```

These functions can throw exceptions. It is hard to track which functions throw exceptions, so we instead catch the exception and instead use a `Result` monad (don't be scared!) which has two possible options - `Ok something` or `Error e`. Then you can tell from the function signature `Or_error.t` that it could return an error.

Listing 3.17: lex_and_parse.ml

```
(* Prints the line number and character number where the
    ↪ error occurred.*)
let print_error_position lexbuf =
  let pos = lexbuf.lex_curr_p in
  Fmt.str "Line:%d Position:%d" pos.pos_lnum (pos.pos_cnum -
      ↪ pos.pos_bol + 1)

let parse_program lexbuf =
  try Ok (Parser.program Lexer.read_token lexbuf) with
  (* catch exception and turn into Error *)
  | SyntaxError msg ->
      let error_msg = Fmt.str "%s: %s@." (
         ↪ print_error_position lexbuf) msg in
      Error (Error.of_string error_msg)
  | Parser.Error ->
      let error_msg = Fmt.str "%s: syntax error@." (
         ↪ print_error_position lexbuf) in
      Error (Error.of_string error_msg)

let pprint_parsed_ast ppf (prog : Parsed_ast.program) =
  Pprint_past.pprint_program ppf prog
```

This file also exposes the pretty-print function for the parsed AST in the library (`pprint_past.ml`). This is useful for debugging or expect tests (as this clip of an early version of Bolt demonstrates):

3.7 WHERE DOES THIS FIT IN THE BOLT PIPELINE?

This is the first stage of the frontend, which can be seen in the `compile_program_ir` function.

Listing 3.18: compile_program_ir.ml

```
let compile_program_ir ?(should_pprint_past = false) ?(
   ↪ should_pprint_tast = false)
    ?(should_pprint_dast = false) ?(should_pprint_drast =
       ↪ false)
    ?(should_pprint_fir = false) ?(ignore_data_races = false
       ↪ ) ?compile_out_file lexbuf =
  let open Result in
  parse_program lexbuf
  >>= maybe_pprint_ast should_pprint_past pprint_parsed_ast
  >>=  ...
```

I still haven't told you where the `lexbuf` comes from. You can either get it from an input channel, as in the `main` function, which reads in a Bolt file from the command line:

Listing 3.19: main.ml

```
In_channel.with_file filename ~f:(fun file_ic ->
            let lexbuf =
              Lexing.from_channel file_ic
              (*Create a lex buffer from the file to read in
                 ↪ tokens *) in
            compile_program_ir lexbuf ...
```

Or, as in the tests (`tests/frontend/expect`), you can read from a string using (`Lexing.from_string input_str`).

3.8 SUMMARY

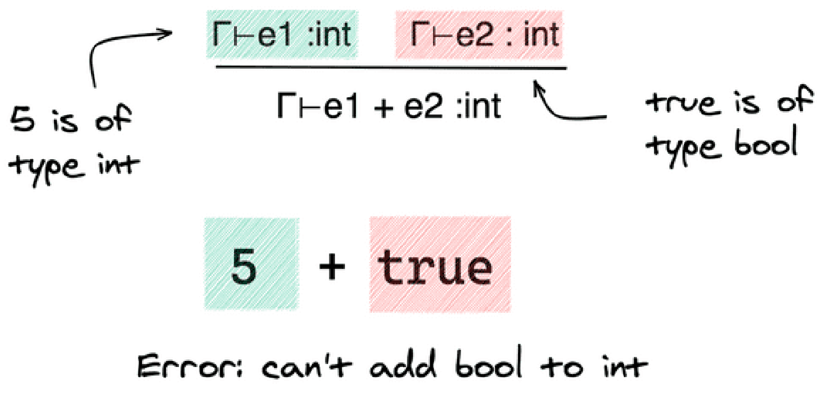
This wraps up the discussion of the lexer and parser. If you haven't already, fork the Bolt repo.

All the code linked in this post is in the `src/frontend/parsing` folder, plus some extra rules in the grammar to cover data-race freedom using *capabilities* (as discussed in my dissertation) and also *inheritance* and *generics*.

Sounds exciting? Next up, we'll talk type-checking and even implement a form of *local type inference*. The post'll be out within the next week, and if you enjoyed this post, I'm sure you'll *love* the next one! I'll explain how to read typing rules ($\Gamma \vdash e : \tau$) and how to implement the type checker for the core language.

4 AN ACCESSIBLE INTRODUCTION TO TYPE THEORY AND IMPLEMENTING A TYPE-CHECKER

June 15, 2020



This post is split into 2 halves: the first half explains the theory behind type-checkers, and the second half gives you a detailed deep-dive into how it's implemented in our compiler for our language Bolt. Even if you aren't interested in writing your own language, the first half is useful if you've ever heard about *type systems* and want to know how they even work!

4.1 WHAT ARE TYPES?

We'd discussed this briefly in the first part of the compiler series, however let's revisit this:

Types are a way of *grouping* program values based on the *behaviour* we'd like them to have. We have our own "types" in our day-to-day lives. E.g. grouping people as *adults* and *children*.

Grouping values together means the compiler doesn't have as many cases to handle. It means we don't need to look at the *actual value*, we just reason about it based on the *group* it's in (its type).

E.g. the values `0`, `1`, `2`, `3`... and all other integers are all given the type `int`. Values of type `int` can take part in arithmetic operations like multiplication on them but we can't concatenate them (like `string`s). See how the type (`int` vs `string`) defines the *behaviour* of the value (what we can do with them)?

It's even clearer in our custom types. E.g when you define a custom `Animal` class with `eat()` and `sleep()` methods, you're saying that objects of type `Animal` have `eat` and `sleep` behaviour. But we can't divide two `Animal` objects by each other – that's not the behaviour we'd allow. What would `dog / cat` even mean?

The role of the **type-checker** is to prevent this kind of nonsensical behaviour from happening. That's what we're building today.

All practical languages have type-checking in some form. **Statically** typed languages like Rust, Java or Haskell check the types at *compile-time*. Dynamically-typed languages like JS and Python **do** still have types – values are tagged with types at runtime, and they check types when executing. If you try to run `5 /`

`"Hello"`, it won't actually run the code, JS/Python will see `"Hello` has type `string` and will throw a runtime error instead of executing it.

Right now, this all feels a little hand-wavey. Let's formalise "preventing nonsensical behaviour". We want a list of rules to check a program against, and then if it passes those, we know our program is safe.

This collection of rules is called a *type system*.

4.2 TYPE SYSTEMS

Here's the deal: types define *safe behaviour*. So if we can give an expression a type, we can use its type to make sure it's being used in the correct way. The type system's rules (called *typing judgements*) therefore try to assign a type tt to an expression ee.

We start by defining seemingly obvious facts, like `true` is of type bool. In our type system, the rule is written as follows:

$\vdash$ **true** : $bool$

Don't be scared by the maths notation: $\vdash$ means "it follows that". You can read this as "it follows that the expression `true` has type `bool`. In general, $\vdash$ **e** : t can be read as "it follows that expression e has type t".

Here's a couple more obvious facts.

$\vdash$ **false** : $bool$

$\vdash n : int$ for any integer nn.

4.2.1 *Typing environments*

Thing is, looking at an expression on its own isn't always enough for us to type-check it. What's the type of a variable `x`? What should we put in place of the ? below?

$\vdash x : ?$

Well it depends on what type it was given when it was defined. So when type-checking, we'll need to keep track of the variables' types as they are defined. We call this the *typing environment*, represented in our rules as Γ . Think of Γ as a look-up function that maps variables to their types.

We update our typing rules to include Γ:

$\Gamma \vdash$ **x** : t.

Back to our typing rule here:

$\Gamma \vdash x : ?$

If $\Gamma(x) = t$, *then* we can say that:

$\Gamma \vdash$ **x** : t.

In our typing rules we stack these two statements to give one compound typing rule for variables. This is an example of an *inference* rule.

$\frac{\Gamma(x)=t}{\Gamma \vdash x:\ t}$

4.2.2 *Inference rules*

This odd stacked "fraction" is a way of representing deductive reasoning (like Sherlock Holmes!). If everything on the top holds, then we *infer* the bottom also holds.

Here's another rule. This says that if we know e_1 and e_2 are `int`s, then if we add them together the result is also an `int`.

$\frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 + e_2 :\ int}$

Using these *inference* rules, we can stack together our pieces of evidence about different parts of the program to reason about the whole program. This is how we build up our type system - we *stack* our rules.

Let's combine what we've learnt so far to type-check a little expression: `x + 1` where our environment Γ tells us `x` is an `int`.

$$\frac{\dfrac{\overline{\Gamma(x)=int}}{\Gamma\vdash x:\ int} \qquad \overline{\Gamma\vdash 1:\ int}}{\Gamma\vdash x+1:\ int}$$

Now if it turns out `x` is an `string`, then $\Gamma(x) = int$ doesn't hold, so all the rules stacked below it don't hold. Our type-checker would raise an *error*, as the types don't match up!

4.2.3 *Axioms*

It's convention here to represent the facts (*axioms*) with a line above them. You can think them as the "base case":

$$\overline{\Gamma\vdash 1:\ int}$$

Since they have nothing on the top, technically everything on the top is true. So the bottom **always** holds.

If you look back to our example and flip it upside down, it kind of looks like a tree. It also lays out our reasoning, so acts as a *proof* - you can just follow the steps. Why is `x+1` an int? Well `x` and `1` are `int`s? Why is `x` an int? Because… You get the idea. So what we've constructed here is called a *proof tree*.

Okay, let's look at another rule. What about an `if-else` block?

```
if (something) {  do_one_thing} else {  do_something_else}
```

Well, the `something` has to have type `bool` as it's a condition. What about the branches? We need to give this expression an overall type, but because we're *statically* type-checking, we don't know which branch will be executed at runtime. Therefore we need both branches to return the **same type** (call it tt), so the expression has the same type *regardless* of which branch we execute.

$$\frac{\Gamma\vdash e_1{:}bool \qquad \Gamma\vdash e_2{:}t \qquad \Gamma\vdash e_3{:}t}{\Gamma\vdash \textbf{if}\ e_1\ \{e_2\}\ \textbf{else}\ \{e_3\}:\ t}$$

JavaScript and Python **would** allow different types for each branch, since they're doing their type-checking at runtime (*dynamically* typed), so they know which branch was chosen.

4.2.4 *Typing the Overall Program*

This post would end up being too long if I were to list all the rules, but you can work through each of the cases. We look at the *grammar* we defined in the previous post and write the corresponding rules.

We want to show the program is *well-typed* (you can assign it a type) to show it is safe. So we essentially want a proof tree that shows it has some type tt (we don't care which):

$\{\} \vdash program : t$

There's just one thing missing here. Initially Γ = {} - it's empty as we haven't defined any variables. How do we **add variables** to Γ?

Remember, we add variables to Γ as we declare them in our program. The syntax for this is a `let` declaration. So say we have the following program:

```
// we have some gamma
let x = e1
// update gamma with x's type
// continue type-checking
e2
```

We type-check this in the order the program executes:

- We get the type of the expression e_1 , call it t1t1
- We then *extend* the environment Γ with the new mapping $x : t_1$.
- We use this extended environment (which we write as $\Gamma, x : t_1$) to type-check e_2 and give it type t_2.

The whole typing rule thus looks like:

$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma, x : t_1 \vdash e_2 : t_2}{\Gamma \vdash \textbf{let } x = e_1;\ e_2 : t_2}$$

4.2.5 *Type Checking vs Type Inference*

One finer point: here we wrote `let x = e`. We could have equally written this as `let x : t = e` – where the programmer *annotates* `x` with type `t`. e.g. `let x : int = 1 + 2`.

These lead to different typing algorithms – in the first case the compiler *infers* that `1+2` has type `int`, and in the second case, the compiler has to *check* that `1+2` has type `int` (since we specified the type `int` of `x`).

As you might imagine, type inference means that the programmer has to write fewer type annotations, but it is much more complex for the compiler – it's like filling a Sudoku from scratch versus checking a Sudoku solution is valid. OCaml and Haskell are examples of languages with type inference baked in.

In practice, most statically-typed languages do require some type annotations, but can infer some types (e.g. the `auto` keyword in C++). This is like completing a partially solved Sudoku puzzle and is much easier.

For Bolt, we're going to infer types *within* a function or method definition, but require programmers to annotate the parameter and return types. This is a nice middle ground.

```
function int something(int x, bool y){
  let z = ...
 // z's type is inferred
  ...
}
```

Okay, enough theory, let's get to implementing this type-checker!

4.3 IMPLEMENTING A TYPE-CHECKER

4.3.1 *Just give me the code!*

You'll need to clone the Bolt repository:

```
git clone https://github.com/mukul-rathi/bolt
```

The Bolt repository `master` branch contains a more complex type-checker, with support for inheritance, function/method overloading and generics. Each of these topics will get its own special post later in the series.

So instead, you'll want to look at the `simple-compiler-tutorial` release. You can do this as follows:

```
git checkout simple-compiler-tutorial
```

This contains a stripped-back version of Bolt from earlier in the development processs. (View this online)

The folder we care about is `src/frontend/typing`.

4.3.2 *A note on OCaml syntax*

In this tutorial we'll be using OCaml syntax. If you're unfamiliar with this, the main gist is that we'll:

a) Pattern match each of the cases (like `switch` statements in other languages). Here we have a variable `x` and we do different things based on each of its cases `A`, `B`

```
match x with  | A -> something  | B -> something_else
```

b) Use a Result monad: in essence, this has two values: `Ok something` and `Error`. We sequence each of the operations using the >>= operator – you don't need to know anything about monads for this tutorial, just think of this as the same as normal expression sequencing with `;`s, just that we're using another operator to represent that earlier expressions might raise an error.

4.3.3 *Types in Bolt*

Bolt has four main types: `int`, `bool`, `void` and user-defined classes. We represent these four options using a variant type `type_expr` in OCaml:

Listing 4.1: ast_types.mli

```
type type_expr = TEInt | TEClass of Class_name.t | TEVoid |
    ↪ TEBool
```

4.3.4 *Annotating our AST with types*

Let's recap our Abstract Syntax Tree from the last part of the series:

Listing 4.2: parsed_ast.mli

```
type identifier =
  | Variable of Var_name.t (* x *)
  | ObjField of Var_name.t * Field_name.t (* x.f *)

type expr =
  | Integer     of loc * int (* 1, 2 *)
  | Boolean     of loc * bool (* true*)
  | Identifier  of loc * identifier
  | Constructor of loc * Class_name.t * constructor_arg list
      ↪  (* new Class(args)*)
  | Let         of loc * type_expr option * Var_name.t *
      ↪ expr
    (** let x = e (optional type annotation) *)
  | Assign      of loc * identifier * expr
  | If          of loc * expr * block_expr * block_expr
    (** If ___ then ___ else ___ *)
  ...

and block_expr = Block of loc * expr list
```

This AST annotates each expression with `loc` – the line and position of the expression. In our type-checking phase, we'll be checking the types of each of the possible expressions. We'll want to store our results by *directly annotating* the AST, so the next compiler stage can view the types just by looking at the AST.

This AST gets the imaginative name `typed_ast`:

Listing 4.3: typed_ast.mli

```
type identifier =
  | Variable of type_expr * Var_name.t
  | ObjField of Class_name.t * Var_name.t * type_expr *
      ↪ Field_name.t
     (** class of the object, type of field *)

type expr =
  | Integer     of loc * int  (** no need to annotate as
      ↪ obviously TEInt *)
  | Boolean     of loc * bool  (** no need to annotate as
      ↪ obviously TEBool *)
  | Identifier  of loc * identifier  (** Type info
      ↪ associated with identifier *)
  | Constructor of loc * type_expr * Class_name.t *
      ↪ constructor_arg list
  | Let         of loc * type_expr * Var_name.t * expr
  | Assign      of loc * type_expr * identifier * expr
  | If          of loc * type_expr * expr * block_expr *
      ↪ block_expr
     (** the If-else type is that of the branch exprs *)
  ...

and block_expr =
  | Block of loc * type_expr * expr list  (** type is of the
      ↪  final expr in block *)
```

We don't annotate obvious types, like for `Integer` and `Boolean`, but we annotate the type of the overall expression for other expressions e.g. the type returned by an `if-else` statement.

A good rule of thumb when annotating the AST is, what would the next stage need to be told about the program that it can't guess from it being well-typed? For an `if-else` statement, if it is well-typed then the if-condition expression is clearly of type `bool`, but we'd need to be told the type of the branches.

4.3.5 *The Type Environment*

Recall that we used our type environment Γ to look up the types of variables. We can store this as a list of *bindings* (variable, type) pairs.

`type_env.ml` also contains a "environment" of helper functions that we'll use in this type-checking phase. These are mostly uninteresting getter methods that you can look at in the repo.

Listing 4.4: type_env.mli

```
type type_binding = Var_name.t * type_expr
type type_env = type_binding list

(** A bunch of getter methods used in type-checking the core
    ↪  language *)
```

```
val get_var_type : Var_name.t -> type_env -> loc ->
    ↪ type_expr Or_error.t
```

It also includes a couple of functions that check we can assign to an identifier (it's not `const` or the special identifier `this`) and that check we don't have duplicate variable declarations (shadowing) in the same scope. These again are conceptually straightforward but are necessary just to cover edge cases.

For example:

Listing 4.5: type_env.ml

```
let check_identifier_assignable class_defns id env loc =
  let open Result in
  match id with
  | Parsed_ast.Variable x ->
      if x = Var_name.of_string "this" then
        Error
          (Error.of_string
             (Fmt.str "%s Type error - Assigning expr to '
                ↪ this'.@." (string_of_loc loc)))
      else Ok ()
  | Parsed_ast.ObjField (obj_name, field_name) ->
      get_obj_class_defn obj_name env class_defns loc
      >>= fun class_defn ->
      get_class_field field_name class_defn loc
      >>= fun (TField (modifier, _, _, _)) ->
      if modifier = MConst then
        Error
          (Error.of_string
             (Fmt.str "%s Type error - Assigning expr to a
                ↪ const field.@."
                (string_of_loc loc)))
      else Ok ()
```

4.3.6 *Typing Expressions*

This. This is the *crux* of the implementation. So if you've been skimming the post, here's where you should pay attention.

What do we need to type-check an expression?

We need the class definitions and function definitions, in case we need to query the types of fields and function/method type signatures. We also need the expression itself, and the typing environment we're using to type-check it.

What do we return? The typed expression, along with its type (we return type separately to make recursive calls more straightforward). Or we return an error, if the expression is not well-typed.

Our function type signature captures this exactly:

Listing 4.6: type_expr.mli

```
val type_expr :
     Parsed_ast.class_defn list
  -> Parsed_ast.function_defn list
  -> Parsed_ast.expr
  -> type_env
  -> (Typed_ast.expr * type_expr) Or_error.t
```

In the second post in this series, I discussed why we're using OCaml. This stage of the compiler is one where it really pays off.

For example to type an identifier, we pattern match based on whether it is a variable `x` or an object field `x.f`. If it is a variable then we get its type from the environment (we pass in `loc` as line+position info for error messages). If it returns a `var_type` without an error, then we return the type-annotated variable.

If it is an object field `x.f`, then we need to look up the type of object `x` in the env and get its corresponding class definition. We can then look up the type of field `f` in the class definition. Then we annotate the identifier with the two bits of type information we've just learnt: the class of the object, and the field type.

Our code does exactly that, with no boilerplate:

Listing 4.7: type_expr.ml

```
let type_identifier class_defns identifier env loc =
  let open Result in
  match identifier with
    | Parsed_ast.Variable var_name ->
      get_var_type var_name env loc
      >>| fun var_type -> (Typed_ast.Variable (var_type,
          ↪ var_name), var_type)
  | Parsed_ast.ObjField (var_name, field_name) ->
      get_obj_class_defn var_name env class_defns loc
      >>= fun (Parsed_ast.TClass (class_name, _, _, _) as
          ↪ class_defn) ->
      get_class_field field_name class_defn loc
      >>| fun (TField (_, field_type, _, _)) ->
      (Typed_ast.ObjField (class_name, var_name, field_type,
          ↪  field_name), field_type)
```

Right, so let's look at expressions. This again is clean code that just reads like our typing judgements. We have `expr1 binop expr2` where `expr1` and `expr2` are being combined using some binary operator e.g. +.

Let's remind ourselves of the rule:

$$\frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 + e_2 : int}$$

What did this say? We type-check the subexpressions e_1 and e_2 first. If they're both `int`s then the overall expression is an `int`.

Here's the main conceptual jump: type-checking each of the expressions on the top of the judgements corresponds to *recursively* calling our type-checking function on the subexpressions. We then combine the results of these type-checking judgments using our rule.

So here, we type-check each of the subexpressions `expr1` and `expr2` (`type_with_defns` just makes the recursive call shorter).

```
let rec type_expr class_defns function_defns (expr :
    ↪ Parsed_ast.expr) env =
  let open Result in
  let type_with_defns = type_expr class_defns function_defns
      ↪  in
  let type_block_with_defns = type_block_expr class_defns
      ↪ function_defns in match expr with
  | Parsed_ast.BinOp (loc, bin_op, expr1, expr2) -> (
      type_with_defns expr1 env
      >>= fun (typed_expr1, expr1_type) ->
      type_with_defns expr2 env
      >>= fun (typed_expr2, expr2_type) ->
```

Recursive calls, check.

Next, because our code deals with more binary operators than just + we slightly generalise our typing judgement. Regardless of whether it is && + or >, the operands expr1 and expr2 have to have the same type. Then we check that this type is int for the + * / % arithmetic operator cases.

If so, then all Ok and we return TEInt as the type of the operand.

```
if not (expr1_type = expr2_type) then
        Error ... (* can't have different types *)
      else
        let type_mismatch_error expected_type actual_type =
            ↪ ...
        match bin_op with
        | BinOpPlus | BinOpMinus | BinOpMult | BinOpIntDiv |
            ↪  BinOpRem ->
            if expr1_type = TEInt then
              Ok (Typed_ast.BinOp (loc, TEInt, bin_op,
                  ↪ typed_expr1, typed_expr2), TEInt)
            else type_mismatch_error TEInt expr1_type
```

As another example of a binary operator, let's look at < <= >>> >=. These take in integers and return a bool, so we check that the operand expr1 has type TEInt and we return TEBool

```
| BinOpLessThan | BinOpLessThanEq | BinOpGreaterThan |
    ↪ BinOpGreaterThanEq ->
            if expr1_type = TEInt then
              Ok (Typed_ast.BinOp (loc, TEBool, bin_op,
                  ↪ typed_expr1, typed_expr2), TEBool)
            else type_mismatch_error TEInt expr1_type
```

Ok, still with me? This post is getting long, so let's just look at if-else statements and let expressions, and then you can look at the rest of the code in the repo.

Again, let's remind ourselves of our typing judgement for if-else.

$$\frac{\Gamma \vdash e_1 : bool \quad \Gamma \vdash e_2 : t \quad \Gamma \vdash e_3 : t}{\Gamma \vdash \textbf{if } e_1 \ \{e_2\} \textbf{ else } \{e_3\} : t}$$

That's three expressions on top of the inference rule.

What do these expressions correspond to? Say it with me, *recursive calls*!

```
| Parsed_ast.If (loc, cond_expr, then_expr, else_expr) -> (
      type_with_defns cond_expr env
      >>= fun (typed_cond_expr, cond_expr_type) ->
      type_block_with_defns then_expr env
      >>= fun (typed_then_expr, then_expr_type) ->
      type_block_with_defns else_expr env
      >>= fun (typed_else_expr, else_expr_type) ->
```

Now we need to check that the returned types are what we expected, i.e. the branches have the same type, and the condition expression has type TEBool. If that's the case, it's all Ok and we can return the type of the branch.

```
if not (then_expr_type = else_expr_type) then
```

```
        Error ...
      else
        match cond_expr_type with
        | TEBool ->
            Ok
              ( Typed_ast.If
                  (loc, then_expr_type, typed_cond_expr,
                   typed_then_expr, typed_else_expr)
              , then_expr_type )
        | _      ->
            Error ...
```

Right, final expression for this post, the `let` expression, which looks like this:

$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma, x:t_1 \vdash e_2 : t_2}{\Gamma \vdash \textbf{let } x = e_1;\ e_2 : t_2}$$

Again, this requires two recursive calls, but note that the second typing judgement requires the type t_1t1 of the first - we need to pass in an extended type environment (after we type-checked e_1e1) to account for this.

We also have an additional requirement: we want our `let` expressions to be *block-scoped*.

```
if {
  let x = ...
  // can access x here
} else {
  // shouldn't be able to access x here
}
// or here
```

So in essence, we want to only update the environment

1. if we have a `let` expression,
2. only for subsequent expressions in the block

We can encode this by *pattern-matching* on our block type-checking rule. If there are no subsequent expressions (i.e. we have no expressions (pattern-match on `[]`) or just one expression (pattern-match on `[expr]`) in the block), then we don't need to update our environment.

```
type_block_expr class_defns function_defns (Parsed_ast.Block
    (loc, exprs)) env =
  ...
  >>= fun () ->
  match exprs with
  | []                        -> Ok (Typed_ast.Block (loc,
      TEVoid, []), TEVoid) (* empty block has type void
      *)
  | [expr]                    ->
      type_with_defns expr env
      >>| fun (typed_expr, expr_type) ->
      (Typed_ast.Block (loc, expr_type, [typed_expr]),
        expr_type)
```

Only if we have at least two expressions left in our block (pattern-match on `expr1 :: expr2 :: exprs`), and the first of the two is a `let` expression do we update the environment for the rest of the block. We use this updated environment in a recursive call on `expr2::exprs` (the remaining expressions). We then combine the result of the typed blocks.

```
| expr1 :: expr2 :: exprs ->
    type_with_defns expr1 env
    >>= fun (typed_expr1, expr1_type) ->
    (let updated_env =
       match typed_expr1 with
       | Typed_ast.Let (_, _, var_name, _) -> (var_name,
          ↪ expr1_type) :: env
       | _ -> env in
     type_block_with_defns (Parsed_ast.Block (loc, expr2
        ↪ :: exprs)) updated_env)
    >>| fun (Typed_ast.Block (_, _, typed_exprs),
       ↪ block_expr_type) ->
    (Typed_ast.Block (loc, block_expr_type, typed_expr1 ::
       ↪  typed_exprs), block_expr_type)
```

4.3.7 *Typing Class and Function Definitions*

We've broken the back of the type-checking now. Let's just wrap up by checking class and function definitions.

We'll skip over the tedium of checking that there are no duplicate class definitions or duplicate field definitions in a class etc. (this is in the repo). Likewise, checking that the types annotated in fields and method/function type signatures are valids is just a matter of checking there is a corresponding class definition.

Unlike our main function, for other functions, we don't actually start with an empty environment when type-checking the function body as we *already* know the types of some variables – the parameters to the function!

Listing 4.8: caption text

```
let init_env_from_params params =
  List.map
    ~f:(function TParam (type_expr, param_name, _, _) -> (
       ↪ param_name, type_expr))
    params
...
type_block_expr class_defns function_defns body_expr(
    ↪ init_env_from_params params)
```

We also need to check that the type of the result of the body is the return type (or if it is void then we don't care about the type returned).

```
>>= fun (typed_body_expr, body_return_type) ->
  (* We throw away returned expr if return type is void *)
  if return_type = TEVoid || body_return_type = return_type
     ↪ then
    Ok
      (Typed_ast.TFunction
         (func_name, maybe_borrowed_ret_ref, return_type,
            ↪ params, typed_body_expr))
  else Error ...
```

For class methods, we can initialise the environment and check the return type in the same way. But we know the type of another special variable `this` – the class itself.

Listing 4.9: type_classes.ml

```
let init_env_from_method_params params class_name =
  let param_env =
    List.map
      ~f:(function TParam (type_expr, param_name, _, _) -> (
        ↪ param_name, type_expr))
      params in
  (Var_name.of_string "this", TEClass class_name) ::
    ↪ param_env
```

4.4 WHERE DOES THIS FIT IN THE BOLT PIPELINE?

We're two stages into the Bolt compiler pipeline - seen in the `compile_program_ir` function.

Listing 4.10: compile_program_ir.ml

```
let compile_program_ir ?(should_pprint_past = false) ?(
    ↪ should_pprint_tast = false)
    ?(should_pprint_dast = false) ?(should_pprint_drast =
        ↪ false)
    ?(should_pprint_fir = false) ?(ignore_data_races = false
        ↪ ) ?compile_out_file lexbuf =
  let open Result in
  parse_program lexbuf
  >>= maybe_pprint_ast should_pprint_past pprint_parsed_ast
  >>= type_program
  >>= maybe_pprint_ast should_pprint_tast pprint_typed_ast
  >>=  ...
```

4.5 TAKE AWAY: 3 ACTIONABLE STEPS

If you've got this far, amazing job! The Bolt repo contains the full code listing with all the typing judgements for each case of the compiler.

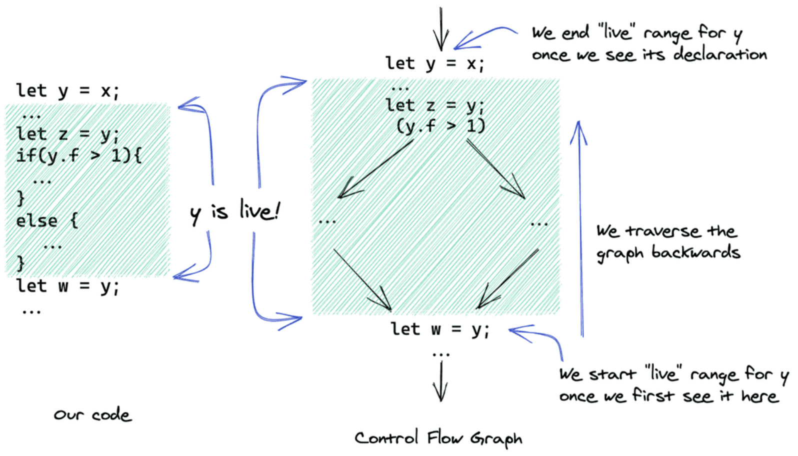
Let's recap what we've done so far:

1. Define what properties in your expression you want to type-check. E.g. a condition expression has type `bool`.
2. Formalise this with a typing judgement. Use the *inference* rule to reason about subexpressions.
3. Map the subexpressions in the inference rule to *recursive* calls, then use the inference rule to combine their results.

Next up, we'll talk about the dataflow analysis used to type-check our linear capabilities in Bolt. This is similar to how the Rust borrow checker uses "non-lexical lifetimes" to check borrowing.

5 A TUTORIAL ON LIVENESS AND ALIAS DATAFLOW ANALYSIS

June 27, 2020



5.1 DATAFLOW ANALYSIS – THE BIG PICTURE

In the previous post in the series, we looked at how type-checking the core language worked. This analysis is *flow-insensitive* – it does not depend on the flow of the execution of the program. If an expression is of type `int` then it will have type `int` regardless of what was executed before or after it.

Some program properties do however depend on the execution path taken by a program. Dataflow analysis tracks how a particular value might propagate as the program executes.

For example, a value might *alias* – you could have multiple references pointing to that value. Whether two references x and y alias can't be determined by looking at their assignments in isolation – we need to track the execution to determine if they have the same value assigned to them.

```
let x = someObject...
let y = someObject
// x and y alias as both point to same object
```

Another example is **liveness analysis** – we say a value is *live* if some point later in the program it could be used, and *dead* otherwise.

```
let x = someValue
// someValue is live...
print(x) // as we use someValue here
x = someOtherValue
// someOtherValue isn't used - so dead
```

Why do we care about alias analysis and liveness analysis? Well it turns out that Rust's *borrow-checker* uses a combination of these to determine when a reference is borrowed. Bolt has a *linear* capability in its type system which acts the same.

Both Rust's borrow checker and Bolt's capabilities prevent data races – an explanation why is in my dissertation.

In a nutshell, Rust works on the principle of **ownership**: you either have one reference (owner) that can read/write (a *mutable* reference) or you can *borrow* (aka alias) the reference.

To enforce this we care about 2 things:

1. When is a reference borrowed? (alias analysis)
2. For how long is it borrowed? (liveness analysis)

Let's elaborate on that second point. When is a reference borrowed? The first version of Rust's borrow checker said this was whilst an alias was in scope.

borrow_example

Copy

```
let x = something
// this is pseudocode not Rust
{  let y = x  ...
   // y will be used in future
  print(y)
  // no longer using y
  x = somethingElse
}// y is out of scope
```

That means we cannot reassign x in the example until y is out of scope. This is a "lexical lifetime" – y's borrow lifetime is determined by its lexical scope. So that means x can't be reassigned, since it is borrowed at that point. But y isn't being used so surely x isn't still borrowed?

That's the idea behind **non-lexical lifetimes** – the borrow ends when the value of y is dead – i.e. it is not being used.

In Bolt, we'll be tracking the lifetimes of references to *objects*.

Let's get implementing it!

5.2 ALIAS ANALYSIS

The first step is to determine when two values alias. How do we know two references will point to the same object without actually executing the program?

We use **abstract interpretation**.

5.2.1 *Abstract Interpretation*

Abstract interpretation is the process of simulating the execution of the program but only storing the program properties we care about. So in our case, we don't care about what the actual result of the program execution is, we're just tracking the *set of aliases*.

Implicit in this is a set of rules for how the program will execute, we call this its **operational semantics**. We will not go into details here, but it's things like:

- Do we evaluate expressions left-to-right or right-to-left?

- When calling a function, do we fully evaluate the argument expression and then call the function with the value of that argument, or do we just plug in the unevaluated argument expression directly into the function and only evaluate it at the point it is used in the function body? The former is called **call-by-value** and is used in most mainstream languages like Java and Python, the latter is called **call-by-name** and is used in Haskell and Lisp.

There's a lot more to say about this - perhaps it's worthy of its own blog post? Send me a tweet if you would like one!

When do we have aliasing? When a new variable is declared or when it is reassigned. For simplicity (and also because we don't want the lifetime of the alias to be larger than the original value) we will not allow aliasing via reassigning an existing variable (e.g. `x := y`).

So we care about expressions of this form:

Copy

```
let x = e
```

The expression e can *reduce* to a value when executing e.g. 1+2 reduces to 3. If when executing e it reduces to a reference y, then the overall expression would look like:

```
let x = y
```

And so x would alias y. So let's write a function that, given an expression, will return the list of identifiers that it could possible reduce to. That'll tell us what x could possibly alias.

We'll store this helper function in an "environment" of helper functions used in the data-race type-checking stage of the Bolt compiler: `data_race_checker_env.mli`

The type signature of this OCaml function is as we'd expect:

Listing 5.1: data_race_checker_env.mli

```
val reduce_expr_to_obj_ids : expr -> identifier list
```

A reminder of the type of `expr` defined in the previous post - `loc` encodes the line number and position of the expression - used for error messages.

Listing 5.2: typed_ast.mli

```
type identifier =
  | Variable of type_expr * Var_name.t
  | ObjField of Class_name.t * Var_name.t * type_expr *
    ↪ Field_name.t
      (** class of the object, type of field *)
type expr =
  | Integer     of loc * int
  | Boolean     of loc * bool
  | Identifier  of loc * identifier
  | Constructor of loc * type_expr * Class_name.t *
    ↪ constructor_arg list
  | Let         of loc * type_expr * Var_name.t * expr
  | Assign      of loc * type_expr * identifier * expr
  | If          of loc * type_expr * expr * block_expr *
    ↪ block_expr
  ...

and block_expr =
  | Block of loc * type_expr * expr list
```

Starting off with the simple cases, it's clear an integer or a boolean value doesn't reduce to an identifier, and an identifier expression reduces to that identifier. A `new SomeClass()` constructor also doesn't reduce to an identifier.

Listing 5.3: data_race_checker_env.ml

```
let rec reduce_expr_to_obj_ids expr =
  match expr with
  | Integer _ | Boolean _ -> []
  | Identifier (_, id) -> [id]
  | Constructor (_, _, _, _) -> []
```

If we have a let expression or assigning to an identifier, then it reduces to that identifier (e.g. `let x = ___` reduces to `x`, and `x.f:= __` reduces to `x.f`):

```
...
  | Let (_, _, _, bound_expr) -> reduce_expr_to_obj_ids
      ↪ bound_expr
  | Assign (_, _, _, assigned_expr) ->
      ↪ reduce_expr_to_obj_ids assigned_expr
```

We can continue doing this for other cases. But what about an `if` statement? Does this expression reduce to `x` or `y`?

```
if (someCondition) {  x} else {  y}
```

In general, without actually executing the expression, we don't know. So we'll have to approximate.

Let's remind ourselves of our goal – to mark a value as borrowed/not linear (Rust / Bolt equiv. terminology) if it is aliased. We're trying to eliminate data races, so we have to be *conservative* – assume it might be aliased even if it isn't. The worst thing would be to let a data race slip through. Abstract interpretation *always* errs on the side of soundness.

So we'll *overapproximate* the list of possible identifiers the expression could reduce to – we don't know which branch so we'll count the identifiers from *both* branches.

```
...
  | If (_, _, _, then_expr, else_expr) ->
      let then_ids = reduce_block_expr_to_obj_ids then_expr
          ↪ in
      let else_ids = reduce_block_expr_to_obj_ids else_expr
          ↪ in
      then_ids @ else_ids
```

So in the example above, we'd return a list `[x, y]`.

5.2.2 *Computing all aliases*

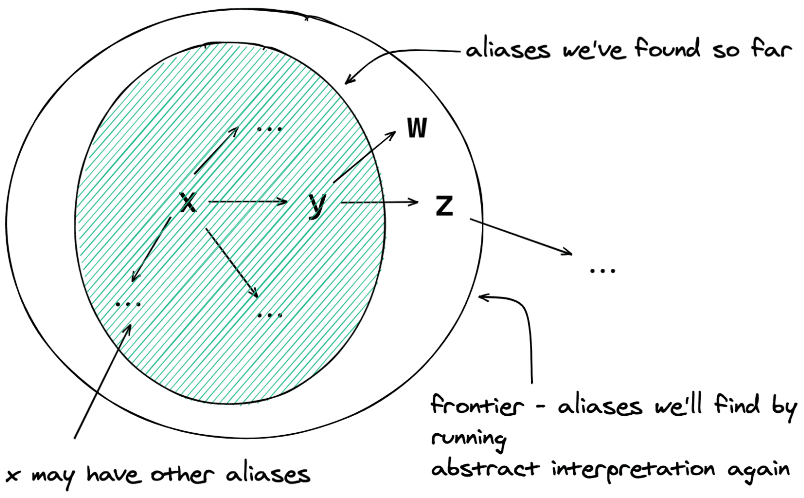
So we know if we have an expression `let y = e` and `e` reduces to `x`, then we have `let y = x` and so `y` is an alias of `x`.

In the expression below, we would also run abstract interpretation (simple in this case) to find that z and w are aliases of y.

```
let y = x...
let z = y
if(y.f > 1){  ...}
else {
}
...
let w = y
```

By transitivity, z and w must also be aliases of x.

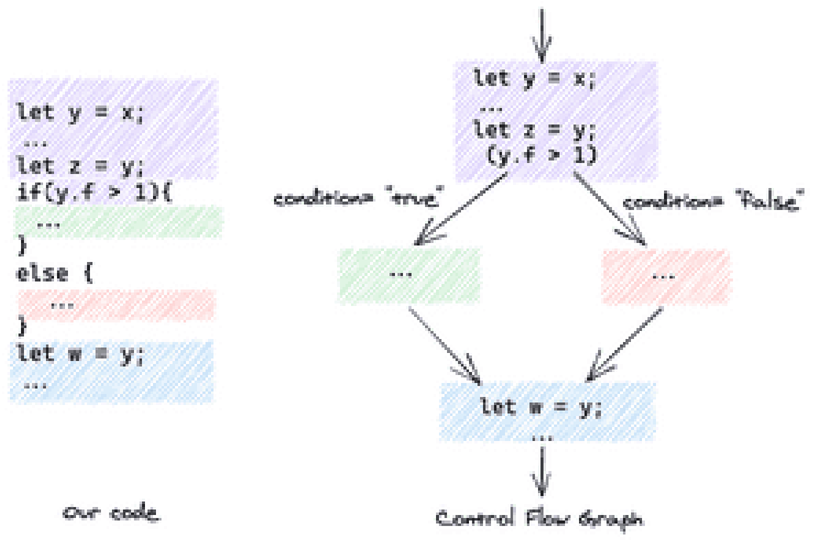
I like to think of this like a graph where each edge is a direct/immediate alias. We can find all aliases, by repeatedly applying abstract interpretation in a "breadth-first search" style - each iteration we're expanding the frontier. And each iteration we try to find aliases of the aliases we've found - so the first iteration we find aliases of x, then the next iteration we find aliases of x and y and so on...


aliases we've found so far
x
y
w
z
...
frontier - aliases we'll find by running abstract interpretation again
x may have other aliases

So we repeat, until we find no more aliases. The full code is linked in the repo, and includes an option of matching fields (i.e. should we consider aliases of x.f as well as x?) We won't in this case, but other aspects of Bolt's data-race type-checker do use this.

But the beauty of functional programming is that the function says what we're doing with no boilerplate:

```
let rec get_all_obj_aliases should_match_fields curr_aliases
    ↪  block_expr =
    find_immediate_aliases_in_block_expr should_match_fields
        ↪  name_to_match curr_aliases
      block_expr
    |> fun updated_aliases ->
    if var_lists_are_equal updated_aliases curr_aliases
       then curr_aliases (* we're done! *)
    else get_all_obj_aliases should_match_fields
        ↪ updated_aliases block_expr
    (* repeat with an expanded frontier *)
```



```
let y = x;
...
let z = y;
if(y.f > 1){
  ...
}
else {
  ...
}
let w = y;
...
```

Figure 1: Note how the control flow graph shows two different execution paths, corresponding to the if-else branch taken.

5.3 LIVENESS ANALYSIS

Alright, so we're halfway there - we've identified our aliases, now we need to find out when they're live. Remember a value is *live* if there's some program execution path in the future it will be used on.

5.3.1 *Control Flow Graph*

To do this, let's formalise the notion of "execution paths" in a program. We're going to be representing a program as a graph of instructions, with edges representing the steps in the execution. We call this the **Control Flow Graph** of the program.

However, if every statement represented a node on the graph, with edges between them, the graph would be incredibly large (a 100 line program would have 100 statements). We can be a bit cleverer and group together statements that will always execute right after each other.

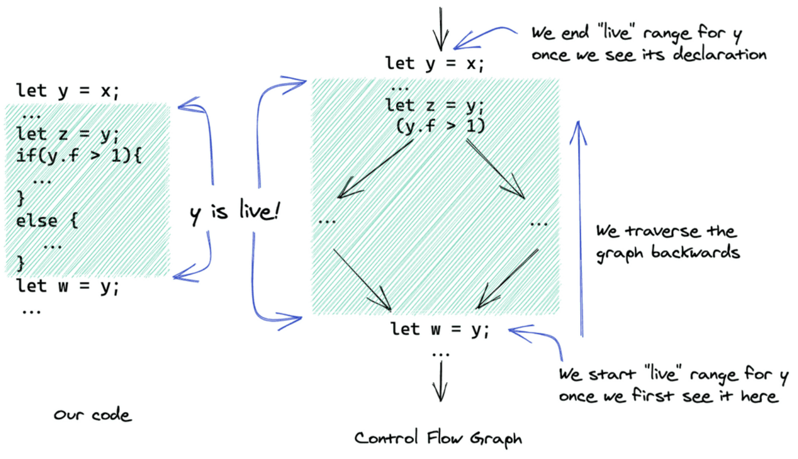
If there's only **one path** from the start statement to the end statement in a group of statements, we can represent this as a **single node** in our control flow graph. We call this node a **basic block** of statements. Below you can see the lines of code highlighted in different colours to show which basic block they lie in.

In our graph, we can pick any statement in the program and ask, is the value of variable y live at this point?

A naive way of checking this is to traverse the graph forwards until you see a use of this value of y (so y is live) or until you reach the end (you therefore know y is dead). But this is hopelessly wasteful - for every statement we have to go forward and traverse the graph to check if the variable's are used.

We can look at it another way, if we traverse *backwards*, then the first use of y that we encounter is the last use if we traversed it going forwards. So as soon as we see y being used when we go backwards, we know that whilst y is defined, it is live.

A picture helps so let's look at our graph:



So to summarise, in liveness analysis we:

- Traverse the Control Flow Graph backwards
- Keep track of the set of live values
- If we see a value for the first time, we add it to our set
- We remove a value if we see its definition

5.4 Implementing our alias liveness checker

Now we have our theoretical understanding in place, let's implement the checker. Whenever we encounter an identifier, we need to know what the original object reference was, the set of possible aliases, and separately those that are live.

Listing 5.4: type_alias_liveness.ml

```
let type_alias_liveness_identifier obj_name possible_aliases
  filter_linear_caps_fn live_aliases id =
  let id_name = get_identifier_name id in
```

Our function deals with three cases depending on the identifier name:

- It is the original object reference
- It is a possible alias
- It is another reference we don't care about

In the first case, we want to filter the linear capabilities (since the reference is not linear so can't use them) if there are live aliases. Along with the updated identifier, we also return the unchanged set of live aliases.

Listing 5.5: type_alias_liveness.ml

```
if id_name = obj_name then (* original object reference *)
    let maybe_updated_capabilities =
      update_capabilities_if_live_aliases
          ↪ filter_linear_caps_fn
      live_aliases (get_identifier_capabilities id) in
```

```
  (set_identifier_capabilities id
     ↪ maybe_updated_capabilities, live_aliases)
```

Otherwise, we need to check if the identifier is one of the possible aliases – if so we add it to the set of live aliases we are tracking. So we then return this updated set of live aliases along with the identifier.

```
else
  ( match
      List.find
        ~f:(fun poss_alias ->
                identifier_matches_var_name poss_alias id)
        possible_aliases
    with
  | Some alias -> alias :: live_aliases
  | None       -> live_aliases )
  |> fun updated_live_aliases -> (id, updated_live_aliases)
```

And the dual in our `type_alias_liveness_expr` function is when we see our `let x = e` expression. Here, remember we execute e first, and then assign it to x, so when traversing this backwards, we remove x from the set of live aliases first (since we've seen its definition), *then* we traverse the bound expression:

```
| Let (loc, type_expr, var_name, bound_expr) ->
      (* remove this var from the set of live aliases *)
      type_alias_liveness_expr_rec
        (List.filter ~f:(fun name -> not (var_name = name))
            ↪ live_aliases)
        bound_expr
      |> fun (updated_bound_expr, updated_live_aliases) ->
      (Let (loc, type_expr, var_name, updated_bound_expr),
          ↪ updated_live_aliases)
```

One interesting case in our Control Flow Graph is the `if-else` split. We treat each branch independently of each other, since they are independent paths in our graph.

```
| If (loc, type_expr, cond_expr, then_expr, else_expr) ->
      type_alias_liveness_block_expr_rec live_aliases
          ↪ then_expr
      |> fun (updated_then_expr, then_live_aliases) ->
      type_alias_liveness_block_expr_rec live_aliases
          ↪ else_expr
      |> fun (updated_else_expr, else_live_aliases) ->
```

How do we recombine the two branches? Well, the definition of liveness is if there is *some* path in which the value is used. Again, we don't know which branch is taken, because we can't execute the program, so we *over-approximate* – we assume both paths could have been taken – so *union* their sets of live aliases, when traversing the if-else condition expression:

```
type_alias_liveness_expr_rec (then_live_aliases @
    ↪ else_live_aliases) cond_expr
      |> fun (updated_cond_expr, cond_live_aliases) ->
      ( If (loc, type_expr, updated_cond_expr, updated_then_expr, updated_else_expr)
      , cond_live_aliases )
```

Another interesting case is when we have a while loop. We can't just traverse the loop once, since we might miss out on some values that could be used in a subsequent iteration and therefore are live. We don't know how many times we'll go round the loop so as ever we *over-approximate* – we'll keep going round the loop until the set of live aliases doesn't change (i.e. we've got all possible live aliases):

```
and type_alias_liveness_loop_expr aliased_obj_name possible_aliases
  filter_linear_caps_fn live_aliases loop_expr =
  type_alias_liveness_block_expr aliased_obj_name possible_aliases filter_linear_caps_fn
    live_aliases loop_expr
  |> fun (updated_loop_expr, updated_live_aliases) ->
  if var_lists_are_equal live_aliases updated_live_aliases then
     (* done! *)
    (updated_loop_expr, updated_live_aliases)
  else
  (* loop again! *)
    type_alias_liveness_loop_expr aliased_obj_name possible_aliases
      filter_linear_caps_fn updated_live_aliases updated_loop_expr
```

5.5 WHERE DOES THIS FIT INTO BOLT?

Our liveness analysis fits into the overall type-checking for linear capabilities in Bolt's data-race type-checker:

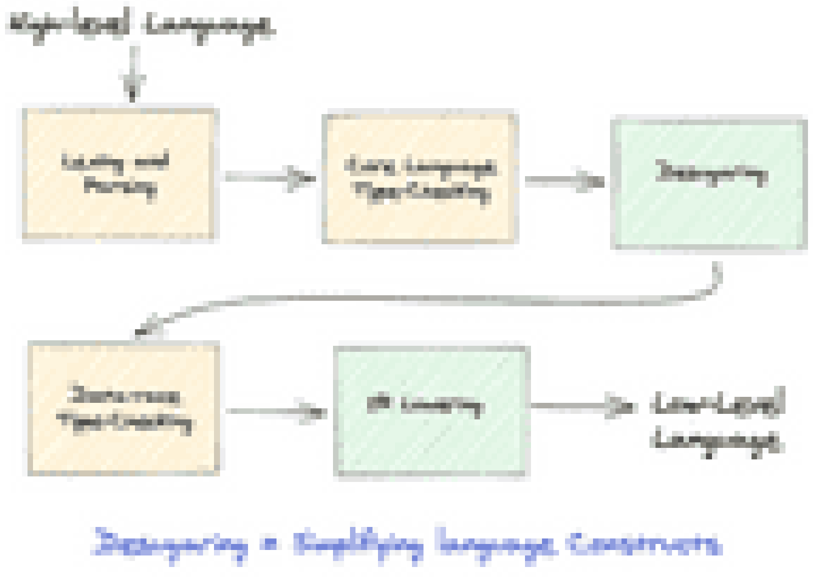
Listing 5.6: type_linear_capabilities.ml

```
let type_linear_object_references obj_name obj_class class_defns block_expr =
  let obj_aliases = ...
  ...
  |> fun updated_block_expr ->
  type_alias_liveness_block_expr obj_name obj_aliases
     filter_linear_caps_fn [] (* we start with empty set of live aliases *)
     updated_block_expr
  |> fun (typed_linear_obj_ref_block_expr, _) -> typed_linear_obj_ref_block_expr
```

We'll talk more about the other aspects of data-race type-checking later, however the next stage of the tutorial is on *desugaring* – the process of taking our high-level language and simplifying it to a lower-level representation. This will set us up nicely for targeting LLVM later in the compiler series.

6 DESUGARING – TAKING OUR HIGH-LEVEL LANGUAGE AND SIMPLIFYING IT!

July 01, 2020



6.1 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.

6.2 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.

6.3 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:

Listing 6.1: type_expr.ml

```
let rec type_expr class_defns function_defns (expr :
    ↪ Parsed_ast.expr) env =
...
| Parsed_ast.For
      (loc, start_expr, cond_expr, step_expr, Parsed_ast.
          ↪ Block (block_loc, loop_expr)) ->
      (* 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_expr]))
             ] ))
        env
```

6.4 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.*

6.4.1 *Removing variable shadowing*

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

```
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:

```
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 _.

Listing 6.2: remove_variable_shadowing.ml

```
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_name) var_name_map) in
  Var_name.of_string
    (Fmt.str "_%s%d" (Var_name.to_string var_name)
        ↪ num_times_var_declared)
```

6.4.2 *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 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:

Listing 6.3: type_expr.ml

```ocaml
| 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_types function_defns loc
    >>| fun (param_types, return_type) ->
    ( Typed_ast.FunctionApp (loc, return_type, param_types , func_name, typed_args_exprs)
    , 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`.

Listing 6.4: desugar_overloading.ml

```ocaml
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_name in
             Fmt.str "%d%s" (String.length class_name_str) class_name_str)
       param_types)
```

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

Listing 6.5: desugar_overloading.ml

```ocaml
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!

6.5 LOWERING TO IR

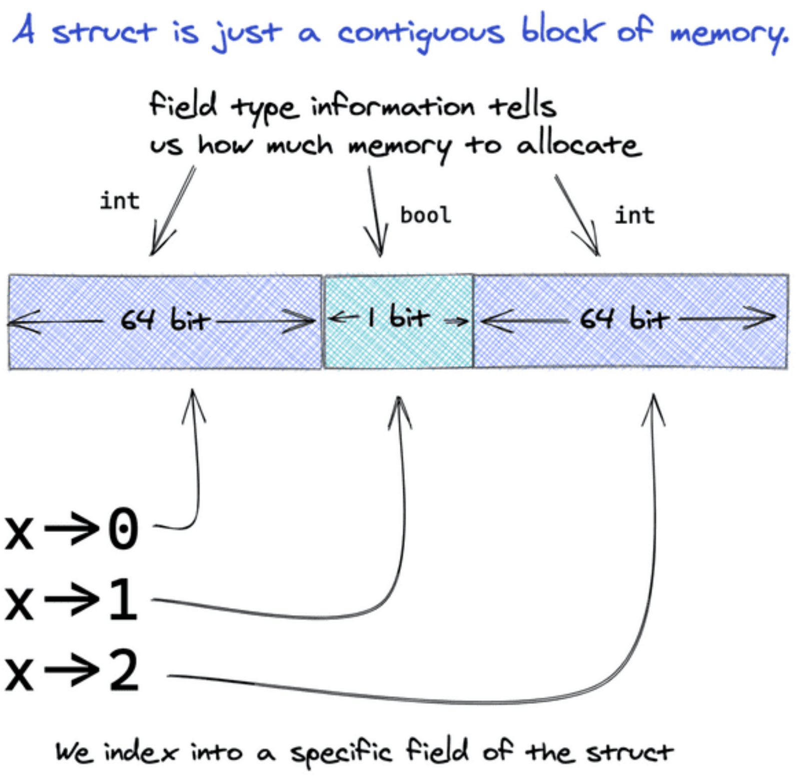
Recapping so far, we first looked at desugaring for loops - this occurs between the parsing and first stage of type-checking. We then looked at the desugaring stage which sits between the two stages of type-checking. We now look at IR lowering stage that occurs after type-checking.

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.

6.5.1 *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.


A struct is just a contiguous block of memory.
field type information tells us how much memory to allocate
int
bool
int
64 bit
1 bit
64 bit
x->0
x->1
x->2
We index into a specific field of the struct

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:

Listing 6.6: frontend_ir.mli

```
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.

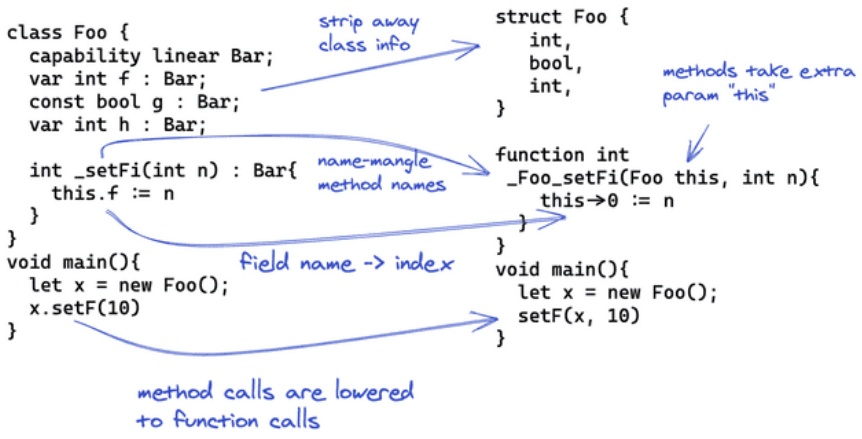
Listing 6.7: ir_gen_env.ml

```
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.



```
class Foo {
  capability linear Bar;
  var int f : Bar;
  const bool g : Bar;
  var int h : Bar;

  int _setFi(int n) : Bar{
    this.f := n
  }
}
void main(){
  let x = new Foo();
  x.setF(10)
}
```

```
struct Foo {
   int,
   bool,
   int,
}
function int
 _Foo_setFi(Foo this, int n){
   this->0 := n
  }
}
void main(){
  let x = new Foo();
  setF(x, 10)
}
```

strip away class info
methods take extra param "this"
name-mangle method names
field name -> index
method calls are lowered to function calls

6.5.2 *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.

Listing 6.8: frontend_ir.mli

```
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:

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

The corresponding generation code is:

Listing 6.9: ir_gen_class_and_function_defns.ml

```
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_expr)
       ; Frontend_ir.Unlock ("this", Frontend_ir.Writer)
       ; Frontend_ir.Identifier (Frontend_ir.Variable "retVal
           ↪ ", None) ]
   | 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_body_expr)
```

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:

Listing 6.10: ir_gen_expr.ml

```
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.
          ↪ Reader else None in
      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.
          ↪ Writer else None in
      Frontend_ir.Assign (ir_id, ir_assigned_expr, lock_held
          ↪ )
```

6.6 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!

7 A PROTOBUF TUTORIAL FOR OCAML AND C++

October 03, 2020

Now that we've desugared our language into a simple "Bolt IR" that is close to LLVM IR, we want to generate LLVM IR. One problem though, our Bolt IR is an OCaml value, but the LLVM API we're using is the native C++ API. We can't import the value directly into the C++ compiler backend, so we need to serialise it first into a **language-independent** data representation.

Hey there! If you came across this tutorial from Google and don't care about compilers, don't worry! This tutorial doesn't really involve anything compiler-specific (it's just the illustrative example). If you care about OCaml then the first half will be up your street, and if you're here for C++ you can skip the OCaml section.

7.1 PROTOCOL BUFFERS SCHEMA

Protocol buffers (aka *protobuf*) encodes your data as a series of binary messages according to a given **schema**. This schema (written in a `.proto` file) captures the structure of your data.

Each message contains a number of fields.

Fields are of the form `modifer type name = someIndex`

Each of the fields have a modifier: `required` / `optional` / `repeated`. The `repeated` modifier corresponds to a list/vector of that field. e.g. `repeated PhoneNumber` represents a value of type `List<PhoneNumber>`.

For example, the following schema would define a Person in a contact book. Each person has to have a name and id, and optionally could have an email address. They might have multiple phone numbers (hence `repeated PhoneNumber`), e.g. for their home, their mobile and work.

```
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;
  enum PhoneType {
    MOBILE = 0;
   HOME = 1;
    WORK = 2;
  }
  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }
  repeated PhoneNumber phones = 4;
}
```

Note here within the schema for a `Person` message, we also **number** the fields (`=1`, `=2`, `=3`, `=4`). This allows us to add fields later without altering the existing ordering of fields (so you can continue to parse these fields without having to know about new fields).

Every time we want to add a new "type" of field, we define a schema (just like how you might define a class to add a new type of object). We can even define a schema within another schema, e.g. within the schema for a `Person`, we defined the schema for the `PhoneNumber` field.

We can also define the schema for an **enum** type `PhoneType`. This enum acted as a "tag" for our phone number.

Now, just as you might want a field to be one of many options (specified by the enum type), you might want the message to contain exactly one of many fields.

For example in our compiler we might want to encode an identifier as *one of two options*. Here we don't want to store data in an index field if we've tagged it as a `Variable`.

Listing 7.1: frontend_ir.mli

```
type identifier =
    | Variable of string
    | ObjField of string * index (* name and field index *)
```

In our protobuf, we can add fields for each of these options, and use the keyword `oneof` to specify that only one of the fields in that group will be set at once.

Now, whilst this tells Protobuf that one of those values is set, we have no way of telling which one of them is set. So we can introduce a `tag` enum field, and query its value (`Variable` or `ObjField`) when deserialising the object.

```
message identifier {
  enum _tag {
    Variable_tag = 1;
    ObjField_tag = 2;
  }
  message _ObjField {
    required string objName = 1;
    required int64 fieldIndex = 2;
  }
  required _tag tag = 1;
  oneof value {
    string Variable = 2;
    _ObjField ObjField = 3;
  }
}
```

Note since the constructor `Variable of string` has only one argument of type `string`, we can just set the type of that field to `string`. We equally could have defined a message `_Variable` with schema:

```
message _Variable {
    required string varName = 1;
  }
  ...
  oneof value {
    _Variable Variable = 2;
    _ObjField ObjField = 3;
  }
```

7.2 MAPPING OCAML TYPE DEFINITIONS TO PROTOBUF SCHEMA

Now we've looked at the `identifier` schema, let's flesh out the rest of the frontend "Bolt IR" schema.

If you notice, whenever we have an OCaml variant type like `identifier` we have a straightforward formula to specify the corresponding schema. We:

- Add a tag field to specify the constructor the value has.
- Define a message schema if a constructor has multiple arguments e.g for `ObjField`. If the constructor has only one argument (`Variable`) we don't need to.
- Specify a `oneof` block containing the fields for each of the constructors.

We could manually write out all of the schema, but we'd have to rewrite these every time our language changed. However we have a hidden weapon up our sleeve - a library that will do this for us!

There's the full Protobuf Schema Guide if you want to learn more about other message types.

7.3 PPX DERIVING PROTOBUF

OCaml allows libraries to extend its syntax using a **ppx** hook. The ppx libraries can preprocess the files using these hooks to extend the language functionality. So we can tag our files with other information, such as which types need to have a protobuf schema generated.

We can update our Dune build file to preprocess our `ir_gen` library with the ppx-deriving protobuf library:

Listing 7.2: dune

```
(library
 (name ir_gen)
 (public_name ir_gen)
 (libraries core fmt desugaring ast)
 (flags
  (:standard -w -39))
 (preprocess
  (pps ppx_deriving_protobuf bisect_ppx --conditional))
```

Note the flags command suppresses warning 39 - "unused `rec` flag" - since the code the ppx_deriving_protobuf library generates raises a lot of those warnings. I would highly recommend that you also suppress these warnings!

An aside, `bisect_ppx` is the tool we use to calculate test coverage. It has a `--conditional` flag since we don't want to preprocess the file with coverage info if we're not computing the test coverage.

Telling PPX deriving protobuf that we want to serialise a type definition is easy - we just stick a `[@@deriving protobuf]` at the end of our type definition. For variant types, we have to specify a `[@key 1]`, `[@key 2]` for each of the variants.

For example, for our `identifier` type definition in our `.mli` file:

Listing 7.3: frontend_ir.mli

```
type identifier =
  | Variable of string [@key 1]
  | ObjField of string * int [@key 2]  [@@deriving protobuf]
```

We do the same thing in the `.ml` file, except we also specify the file to which we want to write the protobuf schema.

Listing 7.4: frontend_ir.mli

```
type identifier =
  | Variable of string [@key 1]
  | ObjField of string * int [@key 2][@@deriving protobuf {
     ↪ protoc= "../../frontend_ir.proto"}]
```

This path is relative to the src file `frontend_ir.ml`. So since this `frontend_ir.ml` file is in the `src/frontend/ir_gen/` folder of the repo, the protobuf schema file will be written to `src/frontend_ir.proto`. If you don't specify a file path to the `protoc` argument, then the `.proto` file will be written to the `_build` folder.

One extra tip, the ppx deriving protobuf library won't serialise lists of lists. For example you can't have:

```
type something = Foo of foo list list [@key 1] [@@deriving
   ↪ protobuf])
```

This is because if we have a message schema for `foo`, then to get a field of type `foo list` is straightforward - we use `repeated foo` in our field schema. But we can't say `repeated repeated foo` to get a list of a list of type foo. So to get around this, you'd have to define another type here:

```
type list_of_foo = foo list [@@deriving protobuf]
(*note no @key since not a variant type *)
type something = Foo of list_of_foo list [@key 1] [
   ↪ @@deriving protobuf]
```

Finally, to serialise our Bolt IR using this schema, PPX deriving protobuf provides us with `<type_name>_to_protobuf` serialisation functions. We use this to get a binary protobuf message that we write to an output channel (`out_chan`) as a sequence of bytes.

Listing 7.5: ir_gen_protobuf.ml

```
let ir_gen_protobuf program out_chan =
    let protobuf_message = Protobuf.Encoder.encode_exn
       ↪ program_to_protobuf program in
    output_bytes out_chan protobuf_message
```

In our overall Bolt compiler pipeline, I write the output to a `.ir` file if provided, or to `stdout`:

Listing 7.6: compile_program_ir.ml

```
...
 match compile_out_file with
    | Some file_name ->
        Out_channel.with_file file_name ~f:(fun file_oc ->
            ir_gen_protobuf program file_oc)
    | None           -> ir_gen_protobuf program stdout )
```

7.4 AUTO-GENERATED PROTOBUF SCHEMA

The README of the repository for the PPX Deriving Library has a thorough explanation of the mapping. We've already gone through an example of the autogenerated schema, so the schema shouldn't be too unfamiliar. There was a slight modification I made though. For (`ObjField of string * int`) I said the generated output was:

```
message _ObjField {
    required string objName = 1;
    required int64 fieldIndex = 2;
  }
```

This is unfortunately not quite true. I put in the objName and fieldIndex for clarity, but the library doesn't know what the semantic meaning of `string * int` is, so instead it looks like:

```
message _ObjField {
    required string _0 = 1;
    required int64 _1 = 2;
  }
```

That's the downside of the autogenerated schema. You get the generic field names `_0`, `_1`, `_2` and so on instead of `objName` and `fieldIndex`. And for `type block_expr = expr list`, you get the field name `_`:

```
message block_expr {  repeated expr _ = 1;}
```

7.5 DECODING PROTOBUF IN C++

Alright, so far we've looked at Protobuf schema definitions and how PPX Deriving Protobuf encodes messages and generates the schema. Now we need to decode it using C++. As with the encoding section, we don't need to know the details of how Protobuf represents our messages in binary.

7.6 GENERATING PROTOBUF DESERIALISATION FILES

The `protoc` compiler automatically generates classes and function definitions from the `.proto` file: this is exposed in the `.pb.h` header file. For `frontend_ir.proto`, the corresponding header file is `frontend_ir.pb.h`.

For Bolt, we're building the C++ compiler backend with the build system Bazel. Rather than manually running the `protoc` compiler to get our `.pb.h` header file and then linking in all the other generated files, we can instead take advantage of Bazel's integration with protoc and get Bazel to run protoc during the build process for us and link it in.

The Bazel build system works with many languages e.g. Java, Dart, Python, not just C++. So we specify two libraries – a language-independent proto_library, and then a specific C++ library (cc_proto) that wraps around that library:

```
load("@rules_cc//cc:defs.bzl", "cc_library")

# Convention:
# A cc_proto_library that wraps a proto_library named foo_proto
# should be called foo_cc_proto.
cc_proto_library(
    name = "frontend_ir_cc_proto",
    deps = [":frontend_ir_proto"],
    visibility = ["//src/llvm-backend/deserialise_ir:__pkg__"],

)

# Conventions:
# 1. One proto_library rule per .proto file.
# 2. A file named foo.proto will be in a rule named foo_proto.
proto_library(
    name = "frontend_ir_proto",
    srcs = ["frontend_ir.proto"],
)
```

We include the cc_proto_library(...) as a build dependency to whatever files that need to use the .pb.h file, and Bazel will compile the protobuf file for us. In our case, this is our deserialise_ir library.

```
load("@rules_cc//cc:defs.bzl", "cc_library")

cc_library(
    name = "deserialise_ir",
    srcs =  glob(["*.cc"]),
    hdrs = glob(["*.h"]),
    visibility = ["//src/llvm-backend:__pkg__", "//src/llvm-
        ↪ backend/llvm_ir_codegen:__pkg__", "//tests/llvm-
        ↪ backend/deserialise_ir:__pkg__"],
    deps = ["//src:frontend_ir_cc_proto", "@llvm"]
)
```

7.7 DESERIALISING A PROTOBUF SERIALISED FILE

The frontend_ir.pb.h file defines a namespace Frontend_ir, with each message mapping to a class. So for our Bolt program, represented by the program message in our frontend_ir.proto file, the corresponding Protobuf class is Frontend_ir::program.

To deserialise a message of a given type, we create an object of the corresponding class. We then call the ParseFromIstream method which deserialises the message from the input stream and stores it in the object. This method returns a boolean value indicating success/failure. We've defined our own custom DeserialiseProtobufException to handle failure:

Listing 7.7: deserialise_protobuf.cc

```
Frontend_ir::program deserialiseProtobufFile(std::string &
    ↪ filePath) {
  Frontend_ir::program program;
  std::fstream fileIn(filePath, std::ios::in | std::ios::
      ↪ binary);
  ...
  if (!program.ParseFromIstream(&fileIn)) {
    throw DeserialiseProtobufException("Protobuf not
        ↪ deserialised from file.");
  }
  return program;
}
```

So we're done!

Well, one caveat... this autogenerated class is a **dumb data placeholder**. We need to read the data out from the `program` object.

7.8 READING OUT PROTOBUF DATA FROM A PROTOBUF CLASS

If you try to read the protoc-generated file `frontend_ir.pb.h`, you'll realise it's a garguantuan monstrosity, which is not nearly as nice as the standard example in the official tutorial. So instead of trying to read the methods from the file, here is an explanation of the structure of the header file.

TIP: Make sure you have code-completion set-up in your IDE - it means you won't need to manually search through `frontend_ir.pb.h` for the right methods.

We'll be deserialising messages using the schema defined below:

Listing 7.8: frontend_ir.proto

```
package Frontend_ir;

message expr {
  enum _tag {
    Integer_tag = 1;
    Boolean_tag = 2;
    Identifier_tag = 3;
    ...
    IfElse_tag = 11;
    WhileLoop_tag = 12;
    Block_tag = 15;
  }
  message _Identifier {
    required identifier _0 = 1;
    optional lock_type _1 = 2;
  }
  ...
  message _IfElse {
    required expr _0 = 1;
    required block_expr _1 = 2;
    required block_expr _2 = 3;
  }
  message _WhileLoop {
    required expr _0 = 1;
    required block_expr _1 = 2;
  }
  ...
  required _tag tag = 1;
  oneof value {
    int64 Integer = 2;
    bool Boolean = 3;
    _Identifier Identifier = 4;
    ...
    _IfElse IfElse = 12;
    _WhileLoop WhileLoop = 13;
    block_expr Block = 16;
    ...
  }
}

message block_expr {
  repeated expr _ = 1;
}
```

PROTOBUF MESSAGE CLASSES AND ENUMS Each Protobuf message definition maps to a class definition. Protobuf enums map to C++ enums. To give each class/enum a globally unique name, they are prepended by the package name (`Frontend_ir`) and any classes they're nested in.

So message `expr` corresponds to class `Frontend_ir_expr`, and enum `_tag` which is nested within message `expr` maps to `Frontend_ir_expr__tag`.

In the repo, the `bin_op` message also has a nested `_tag` enum, and this maps to `Frontend_ir_bin_op__tag` (note how this namespacing means it doesn't clash with the `_tag` definition in the `expr` message).

This holds for arbitrary levels of nesting, e.g. the nested message `_IfElse` in the `expr` message maps to the class `Frontend_ir_expr__IfElse`.

Specific enum values for the enum `_tag` e.g. `Integer_tag` can be referred to as `Frontend_ir_expr__tag_Integer_tag`.

Rather than writing classes\enums by concatenating the package and class names with `_`, Protobuf also has a nested class type alias, so you can write these nested message classes as `Frontend_ir::expr` and `Frontend_ir::expr::_IfElse`.

ACCESSING SPECIFIC PROTOBUF FIELDS Each protobuf *required* field `field_name` has a corresponding accessor method `field_name()` (where the field name is converted to lower-case). So the field `tag` in the `expr` message would map to the method `tag()` in the `Frontend_ir::expr` class, and the field `Integer` would map to the method `integer()`, `IfElse` to `ifelse()` etc.

NB: to reiterate, don't get confused between the field name e.g `IfElse` and the type of the field `_IfElse` in the Protobuf message (note the prepended underscore). This is only because the OCaml PPX deriving protobuf library gave them those names - we could have chosen less confusing names if we were writing this proto schema manually.

For *optional* fields, we can access the fields in the same way, but we also have a `has_field_name()` boolean function to check if a field is there.

For *repeated* fields, we instead have a `field_name_size()` function to query the number of items, and we can access item i using the `field_name(i)` method. So for a field name `_1` the corresponding methods are `_1_size()` and `_1(i)`. For field name `_` in the autogenerated schema, the methods would correspondingly be `__size()` and `_(i)`.

7.9 SANITISING OUR FRONTEND IR

Let's put this in practice by sanitising our protobuf classes into C++ classes directly mapping our OCaml type definitions. Each of the OCaml `expr` variants maps to a subclass of an abstract `ExprIR` class.

```
struct ExprIR {
  virtual ~ExprIR() = default;
  ...
};

struct ExprIfElseIR : public ExprIR {
  std::unique_ptr<ExprIR> condExpr;
  std::vector<std::unique_ptr<ExprIR>> thenBlock;
  std::vector<std::unique_ptr<ExprIR>> elseBlock;
  ExprIfElseIR(const Frontend_ir::expr::_IfElse &expr);
  ...
};
struct ExprWhileLoopIR : public ExprIR {
  std::unique_ptr<ExprIR> condExpr;
  std::vector<std::unique_ptr<ExprIR>> loopExpr;
  ExprWhileLoopIR(const Frontend_ir::expr::_WhileLoop &expr)
      ↪ ;
  ...
};
...
```

I use `std::unique_ptr` all over the compiler backend to avoid explicitly managing memory. You can use standard pointers too - this is just a personal preference!

Remember how we had a specific `tag` field to distinguish between variants of the `expr` type. We have a `switch` statement on the value of the `tag` field. This `tag` tells us which of the `oneof{}` fields is set. We then access the correspondingly set field e.g. `expr.ifelse()` for the `IfElse_tag` case:

Listing 7.9: expr_ir.cc

```
std::unique_ptr<ExprIR> deserialiseExpr(const Frontend_ir::
    ↪ expr &expr){
  switch (expr.tag()) {
    case Frontend_ir::expr__tag_IfElse_tag:
      return std::unique_ptr<ExprIR>(new ExprIfElseIR(expr.
          ↪ ifelse()));
    case Frontend_ir::expr__tag_WhileLoop_tag:
      return std::unique_ptr<ExprIR>(new ExprWhileLoopIR(
          ↪ expr.whileloop()));
    ...
  }
}
```

Looking at the message schema for the `_IfElse` and `block_expr` messages:

Listing 7.10: frontend_ir.proto

```
message _IfElse {
    required expr _0 = 1;
    required block_expr _1 = 2;
    required block_expr _2 = 3;
  }
  message block_expr {
  repeated expr _ = 1;
}
```

Our constructor reads each of the `_IfElse` message's fields in turn. We can then use our `deserialiseExpr` to directly deserialise the `_0` field. However, because the message `block_expr` has a repeated field `_`, we have to iterate through the list of expr messages in that field in a for loop:

Listing 7.11: expr_ir.cc

```
ExprIfElseIR::ExprIfElseIR(const Frontend_ir::expr::_IfElse
    ↪ &expr) {
  Frontend_ir::expr condMsg = expr._0();
  Frontend_ir::block_expr thenBlockMsg = expr._1();
  Frontend_ir::block_expr elseBlockMsg = expr._2();

  condExpr = deserialiseExpr(condMsg);
  for (int i = 0; i < thenBlockMsg.__size(); i++) {
    thenExpr.push_back(deserialiseExpr(thenBlockMsg._(i)));
  }
  for (int i = 0; i < elseBlockMsg.__size(); i++) {
    elseExpr.push_back(deserialiseExpr(elseBlockMsg._(i)));
  }
}
```

The rest of the deserialisation code for the Bolt schema follows the same pattern.

7.10 WRAP UP

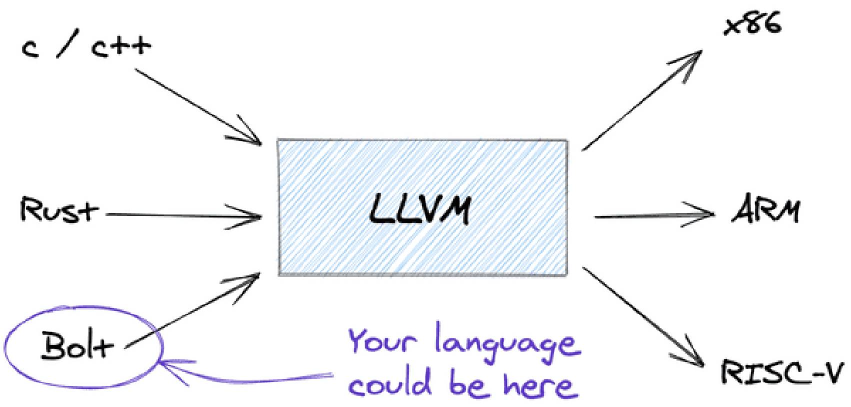
In this post we've converted from our OCaml frontend IR to the equivalent C++ classes. We'll use these C++ classes to generate LLVM IR code in the next part of the tutorial.

Part III

C++ AND LLVM/IR BACK END

8 A COMPLETE GUIDE TO LLVM FOR PROGRAMMING LANGUAGE CREATORS

December 24, 2020



Update: this post has now taken off on Hacker News and Reddit. Thank you all!

8.1 WHO'S THIS TUTORIAL FOR?

This series of compiler tutorials is for people who don't just want to create a *toy* language. You want objects. You want polymorphism. You want concurrency. You want garbage collection. Wait you don't want GC? Okay, no worries, we won't do that :P

If you've just joined the series at this stage, here's a quick recap. We're designing a Java-esque concurrent object-oriented programming language *Bolt*. We've gone through the compiler frontend, where we've done the parsing, type-checking and dataflow analysis. We've desugared our language to get it ready for LLVM – the main takeaway is that objects have been desugared to structs, and their methods desugared to functions.

Learn about LLVM and you'll be the envy of your friends. Rust uses LLVM for its backend, so it must be cool. You'll beat them on all those performance benchmarks, without having to hand-optimise your code or write machine assembly code. Shhhh, I won't tell them.

8.2 JUST GIVE ME THE CODE!

All the code can be found in the Bolt compiler repository.

The C++ class definitions for our desugared representation (we call this *Bolt IR*) can be found in deserialise_ir folder. The code for this post (the LLVM IR generation) can be found in the llvm_ir_codegen folder. The repo uses the Visitor design pattern and ample use of `std::unique_ptr` to make memory management easier.

To cut through the boilerplate, to find out how to generate LLVM IR for a particular language expression, search for the `IRCodegenVisitor::codegen` method that takes in the corresponding `ExprIR` object. e.g. for if-else statements:

```
Value *IRCodegenVisitor::codegen(const ExprIfElseIR &expr) {
      ...
 // this is the LLVM IR generation
}
```

8.3 UNDERSTANDING LLVM IR

LLVM sits in the **middle-end** of our compiler, *after* we've desugared our language features, but *before* the backends that target specific machine architectures (x86, ARM etc.)

LLVM's IR is pretty low-level, it can't contain language features present in some languages but not others (e.g. classes are present in C++ but not C). If you've come across instruction sets before, LLVM IR is a RISC instruction set.

The upshot of it is that LLVM IR looks like a more *readable* form of assembly. As LLVM IR is **machine independent**, we don't need to worry about the number of registers, size of datatypes, calling conventions or other machine-specific details.

So instead of a fixed number of physical registers, in LLVM IR we have an unlimited set of *virtual* registers (labelled `%0`, `%1`, `%2`, `%3`... we can write and read from. It's the backend's job to map from virtual to physical registers.

And rather than allocating specific sizes of datatypes, we retain **types** in LLVM IR. Again, the backend will take this type information and map it to the size of the datatype. LLVM has types for different sizes of `int`s and floats, e.g. `int32`, `int8`, `int1` etc. It also has derived types: like **pointer** types, **array** types, **struct** types, **function** types. To find out more, check out the Type documentation.

Now, built into LLVM are a set of optimisations we can run over the LLVM IR e.g. dead-code elimination, function inlining, common subexpression elimination etc. The details of these algorithms are irrelevant: LLVM implements them for us.

Our side of the bargain is that we write LLVM IR in Static Single Assignment (SSA) form, as SSA form makes life easier for optimisation writers. SSA form sounds fancy, but it just means we define variables before use and assign to variables **only once**. In SSA form, we cannot reassign to a variable, e.g. `x = x+1`; instead we assign to a fresh variable each time (`x2 = x1 + 1`).

So in short: LLVM IR looks like assembly with **types**, minus the messy machine-specific details. LLVM IR must be in SSA form, which makes it easier to optimise. Let's look at an example!

8.3.1 *An example: Factorial*

Let's look at a simple factorial function in our language Bolt:

Listing 8.1: factorial.bolt

```
function int factorial(int n){
  if (n==0) {    1   }
  else{    n * factorial(n - 1)   }
}
```

The corresponding LLVM IR is as follows:

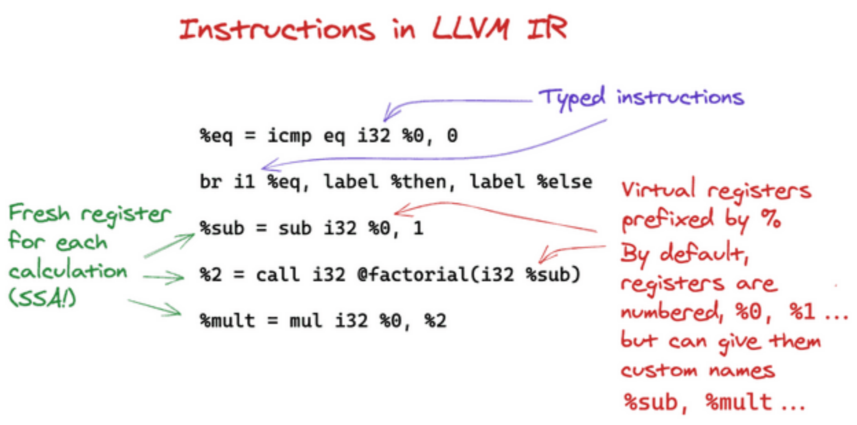
Listing 8.2: factorial.ll

```
define i32 @factorial(i32) {
entry:
  %eq = icmp eq i32 %0, 0    // n == 0
  br i1 %eq, label %then, label %else

then:                                                ; preds = %entry
  br label %ifcont

else:                                                ; preds = %entry
  %sub = sub i32 %0, 1    // n - 1
  %2 = call i32 @factorial(i32 %sub) // factorial(n-1)
  %mult = mul i32 %0, %2   // n * factorial(n-1)
  br label %ifcont

ifcont:                                              ; preds = %else, %then
  %iftmp = phi i32 [ 1, %then ], [ %mult, %else ]
  ret i32 %iftmp
}
```

Note the `.ll` extension is for **human-readable** LLVM IR output. There's also `.bc` for bit-code, a more compact machine representation of LLVM IR.

We can walk through this IR in 4 levels of detail:

AT THE INSTRUCTION LEVEL: Notice how LLVM IR contains assembly instructions like `br` and `icmp`, but abstracts the machine-specific messy details of function calling conventions with a single `call` instruction.

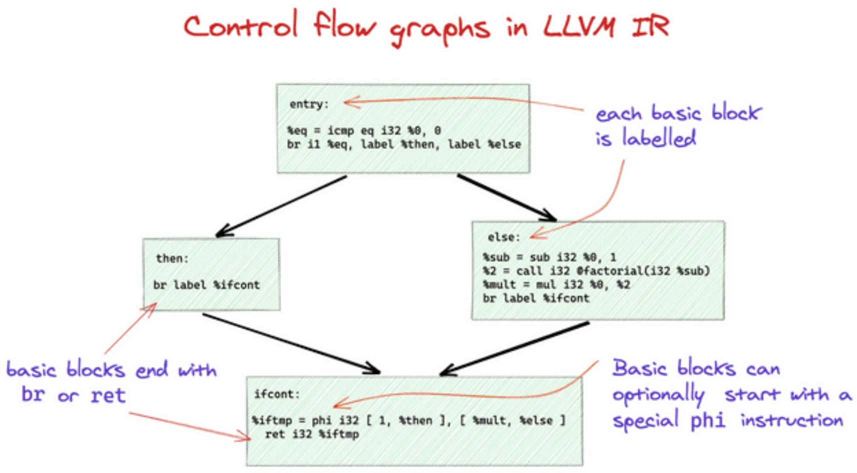

Instructions in LLVM IR
Typed instructions
Fresh register for each calculation (SSA!)
Virtual registers prefixed by %
By default, registers are numbered, %0, %1 ...
but can give them custom names
%sub, %mult ...

```
%eq = icmp eq i32 %0, 0
br i1 %eq, label %then, label %else
%sub = sub i32 %0, 1
%2 = call i32 @factorial(i32 %sub)
%mult = mul i32 %0, %2
```

AT THE CONTROL FLOW GRAPH LEVEL: If we take a step back, you can see the IR defines the **control flow graph** of the program. IR instructions are grouped into labeled **basic blocks**, and the `preds` labels for each block represent incoming edges to that block. e.g. the `ifcont` basic block has predecessors `then` and `else`:

At this point, I'm going to assume you have come across Control Flow Graphs and basic blocks. We introduced Control Flow Graphs in a previous post in the series, where we used them to perform different dataflow analyses on the program.

I'd recommend you go and check the CFG section of that dataflow analysis post now. I'll wait here :)


Control flow graphs in LLVM IR
each basic block is labelled
basic blocks end with br or ret
Basic blocks can optionally start with a special phi instruction

```
entry:

%eq = icmp eq i32 %0, 0
br i1 %eq, label %then, label %else
```

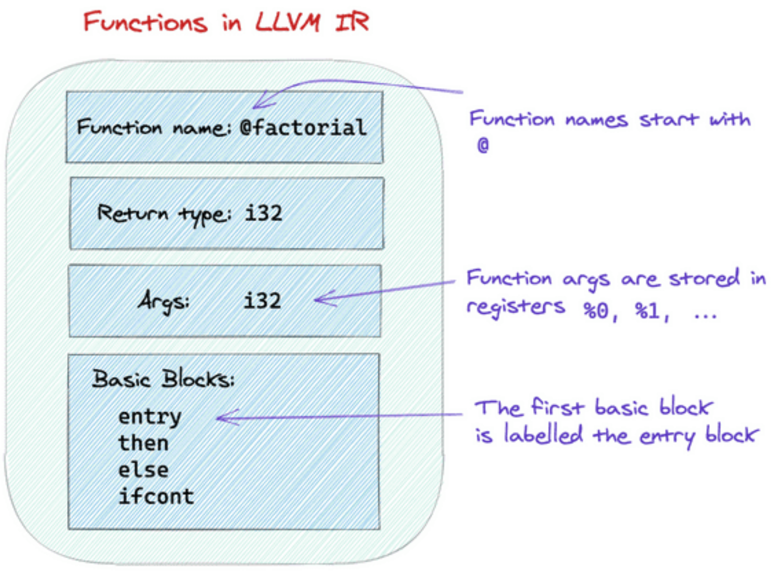
```
then:

br label %ifcont
```

```
else:
%sub = sub i32 %0, 1
%2 = call i32 @factorial(i32 %sub)
%mult = mul i32 %0, %2
br label %ifcont
```

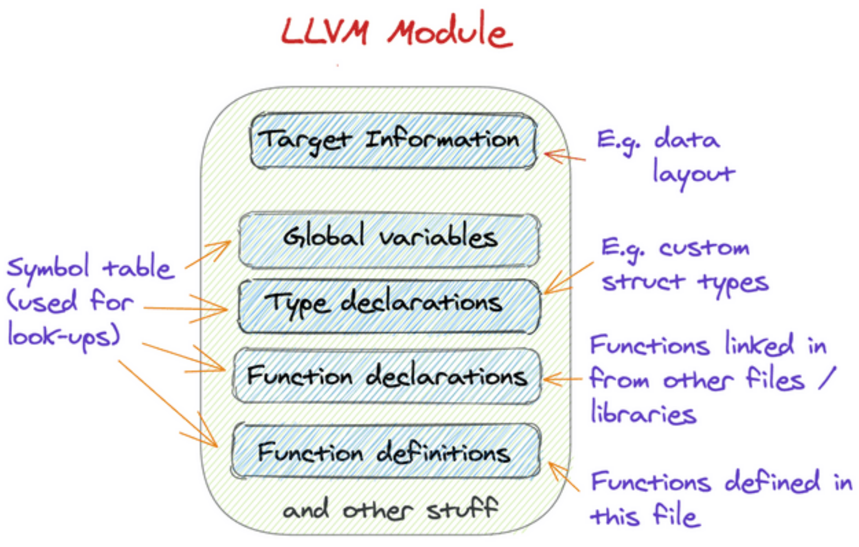
```
ifcont:

%iftmp = phi i32 [ 1, %then ], [ %mult, %else ]
  ret i32 %iftmp
```

The phi instruction represents **conditional assignment**: assigning different values depending on which preceding basic block we've just come from. It is of the form `phi type [val1, predecessor1], [val2, predecessor2], ...` In the example above, we set `%iftmp` to 1 if we've come from the `then` block, and `%mult` if we've come from the `else` block. Phi nodes must be at the **start** of a block, and include one entry for each predecessor.

AT THE FUNCTION LEVEL: Taking another step back, the overall structure of a function in LLVM IR is as follows:


Functions in LLVM IR
Function name: @factorial
Return type: i32
Args: i32
Basic Blocks:
entry
then
else
ifcont
Function names start with @
Function args are stored in registers %0, %1, ...
The first basic block is labelled the entry block

AT THE MODULE LEVEL: An LLVM **module** contains all the information associated with a program file. (For multi-file programs, we'd link together their corresponding modules.)



Our `factorial` function is just one function definition in our module. If we want to execute the program, e.g. to compute `factorial(10)` we need to define a `main` function, which will be the entrypoint for our program's execution. The `main` function's signature is a hangover from C (we return 0 to indicate successful execution):

Listing 8.3: example_program.c

```c
// a C main function
int main(){
  factorial(10);
  return 0;
}
```

We specify that we want to compile for an Intel Macbook Pro in the module target info:

Listing 8.4: example_module.ll

```llvm
source_filename = "Module"
target triple = "x86_64-apple-darwin18.7.0"
...
define i32 @factorial(i32) {
  ...
}
define i32 @main() {
entry:
  %0 = call i32 @factorial(i32 10)
  ret i32 0
}
```

8.4 THE LLVM API: KEY CONCEPTS

Now we've got the basics of LLVM IR down, let's introduce the LLVM API. We'll go through the key concepts, then introduce more of the API as we explore LLVM IR further.

LLVM defines a whole host of classes that map to the concepts we've talked about.

- `Value`
- `Module`
- `Type`
- `Function`
- `BasicBlock`
- `BranchInst` ...

These are all in the namespace `llvm`. In the Bolt repo, I chose to make this namespacing explicit by referring to them as `llvm::Value`, `llvm::Module` etc.)

Most of the LLVM API is quite mechanical. Now you've seen the diagrams that define modules, functions and basic blocks, the relationship between their corresponding classes in the API falls out nicely. You can query a `Module` object to get a list of its `Function` objects, and query a `Function` to get the list of its `BasicBlock`s, and the other way round: you can query a `BasicBlock` to get its parent `Function` object.

`Value` is the base class for any value computed by the program. This could be a function (`Function` subclasses `Value`), a basic block (`BasicBlock` also subclasses `Value`), an instruction, or the result of an intermediate computation.

Each of the expression `codegen` methods returns a `Value *`: the result of executing that expression. You can think of these `codegen` methods as generating the IR for that expression and the `Value *` representing the virtual register containing the expression's result.

Listing 8.5: ir_codegen_visitor.h

```
virtual Value *codegen(const ExprIntegerIR &expr) override;
virtual Value *codegen(const ExprBooleanIR &expr) override;
virtual Value *codegen(const ExprIdentifierIR &expr)
    ↪ override;
virtual Value *codegen(const ExprConstructorIR &expr)
    ↪ override;
virtual Value *codegen(const ExprLetIR &expr) override;
virtual Value *codegen(const ExprAssignIR &expr) override;
```

How do we generate the IR for these expressions? We create a **unique** `Context` object to tie our whole code generation together. We use this `Context` to get access to core LLVM data structures e.g LLVM modules and `IRBuilder` objects.

We'll use the context to create just one module, which we'll imaginatively name `"Module"`.

Listing 8.6: ir_codegen_visitor.cc

```
context = make_unique<LLVMContext>();
builder = std::unique_ptr<IRBuilder<>>(new IRBuilder<>(*
    ↪ context));
module = make_unique<Module>("Module", *context);
```

8.4.1 *IRBuilder*

We use the `IRBuilder` object to incrementally build up our IR. It is intuitively the equivalent of a file pointer when reading/writing a file – it carries around *implicit* state, e.g. the last instruction added, the basic block of that instruction etc. Like moving around a file pointer, you can set the `builder` object to insert instructions at the end of a particular Basic Block with the `SetInsertPoint(BasicBlock *TheBB)` method. Likewise you can get the current basic block with `GetInsertBlock()`.

The builder object has `Create___()` methods for each of the IR instructions. e.g. `CreateLoad` for a `load` instruction , `CreateSub`, `CreateFSub` for integer and floating point `sub` instructions respectively etc. Some `Create__()` instructions take an optional `Twine` argument: this is used to give the result's register a custom name. e.g. `iftmp` is the twine for the following instruction:

`%iftmp = phi i32 [ 1, %then ], [ %mult, %else]`

Use the IRBuilder docs to find the method corresponding to your instruction.

8.4.2 *Types and Constants*

We don't directly construct these, instead we `get__()` them from their corresponding classes. (LLVM keeps track of how a unique instance of each type / constant class is used).

For example, we `getSigned` to get a constant signed integer of a particular type and value, and `getInt32Ty` to get the `int32` type.

Listing 8.7: expr_codegen.cc

```
Value *IRCodegenVisitor::codegen(const ExprIntegerIR &expr){
 return ConstantInt::getSigned((Type::getInt32Ty(*context)),
                                               expr.val);
};
```

Function types are similar: we can use `FunctionType::get`. Function types consist of the return type, an array of the types of the params and whether the function is variadic:

Listing 8.8: function_codegen.cc

```
FunctionType::get(returnType, paramTypes, false /* doesn't
    ↪ have variadic args */);
```

8.4.3 *Type declarations*

We can declare our own custom struct types.

e.g. a Tree with a `int` value, and pointers to left and right subtrees:

```
%Tree = type {i32, Tree*, Tree* }
```

Defining a custom struct type is a two-stage process.

First we create the type with that name. This adds it to the module's **symbol table**. This type is **opaque**: we can now reference in other type declarations e.g. function types, or other struct types, but we can't create structs of that type (as we don't know what's in it).

```
StructType *treeType = StructType::create(*context,
   ↪ StringRef("Tree"));
```

LLVM boxes up strings and arrays using `StringRef` and `ArrayRef`. You can directly pass in a string where the docs require a StringRef, but I choose to make this `StringRef` explicit above.

The second step is to specify an array of types that go in the struct body. Note since we've defined the opaque `Tree` type, we can get a `Tree *` type using the `Tree` type's `getPointerTo()` method.

Copy

```
treeType->setBody(ArrayRef<Type *>({Type::getInt32Ty(*context);, treeType->getPointer
```

So if you have custom struct types referring to other custom struct types in their bodies, the best approach is to declare all of the opaque custom struct types, *then* fill in each of the structs' bodies.

Listing 8.9: class_codegen.cc

```
void IRCodegenVisitor::codegenClasses(
    const std::vector<std::unique_ptr<ClassIR>> &classes) {
  // create (opaque) struct types for each of the classes
  for (auto &currClass : classes) {
    StructType::create(*context, StringRef(currClass->
        ↪ className));
  }
  // fill in struct bodies
  for (auto &currClass : classes) {
    std::vector<Type *> bodyTypes;
    for (auto &field : currClass->fields) {
          // add field type
          bodyTypes.push_back(field->codegen(*this));
    }
    // get opaque class struct type from module symbol table
    StructType *classType =
        module->getTypeByName(StringRef(currClass->className
            ↪ ));
    classType->setBody(ArrayRef<Type *>(bodyTypes));
  }
```

Functions

Functions operate in a similar two step process:

1. Define the function prototypes
2. Fill in their function bodies (skip this if you're linking in an external function!)

The function prototype consists of the function name, the function type, the "linkage" information and the module whose symbol table we want to add the function to. We choose External linkage - this means the function prototype is viewable externally. This means that we can link in an external function definition (e.g. if using a library function), or expose our function definition in another module. You can see the full enum of linkage options here.

Listing 8.10: function_codegen.cc

```
Function::Create(functionType, Function::ExternalLinkage,
                           function->functionName, module.
                              ↪ get());
```

To generate the function definition we just need to use the API to construct the control flow graph we discussed in our `factorial` example:

Listing 8.11: function_codegen.cc

```
void IRCodegenVisitor::codegenFunctionDefn(const FunctionIR
    ↪ &function) {
  // look up function in module symbol definition
  Function *llvmFun =
      module->getFunction(function.functionName);
  BasicBlock *entryBasicBlock =
      BasicBlock::Create(*context, "entry", llvmFun);
  builder->SetInsertPoint(entryBasicBlock);
  ...
```

The official Kaleidoscope tutorial has an excellent explanation of how to construct a control flow graph for an if-else statement.

8.5 MORE LLVM IR CONCEPTS

Now we've covered the basics of LLVM IR and the API, we're going to look at some more LLVM IR concepts and introduce the corresponding API function calls alongside them.

8.6 STACK ALLOCATION

There are two ways we can store values in local variables in LLVM IR. We've seen the first: **assignment to virtual registers**. The second is **dynamic memory allocation** to the stack using the `alloca` instruction. Whilst we can store ints, floats and pointers to either the stack or virtual registers, **aggregate** datatypes, like structs and arrays, don't fit in registers so have to be stored on the stack.

Yes, you read that right. Unlike most programming language memory models, where we use the heap for dynamic memory allocation, in LLVM we just have a stack.

Heaps are not provided by LLVM – they are a *library* feature. For single-threaded applications, stack allocation is sufficient. We'll talk about the need for a global heap in multi-threaded programs in the next post (where we extend Bolt to support concurrency).

We've seen struct types e.g. `{i32, i1, i32}`. Array types are of the form `[num_elems x elem_type]`. Note `num_elems` is a constant – you need to provide this when generating the IR, not at runtime. So `[3 x int32]` is valid but `[n x int32]` is not.

We give `alloca` a type and it allocates a block of memory on the stack and returns a *pointer* to it, which we can store in a register. We can use this pointer to load and store values from/onto the stack.

For example, storing a 32-bit int on the stack:

```
%p = alloca i32 // store i32* pointer in %p
```

```
store i32 1, i32* %p
%1 = load i32, i32* %p
```

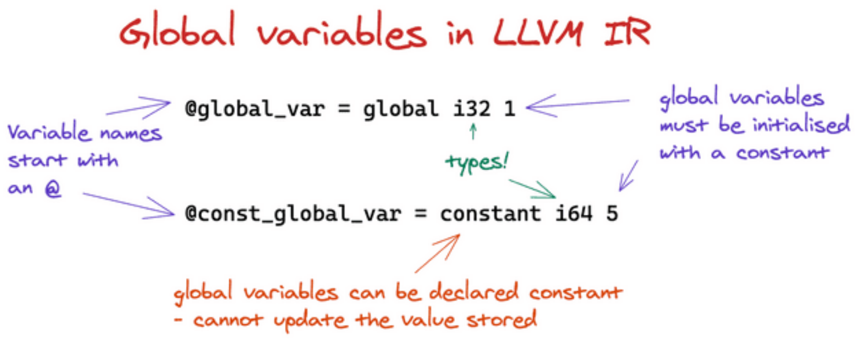
The corresponding builder instructions are... you guessed it `CreateAlloca`, `CreateLoad`, `CreateStore`. `CreateAlloca` returns a special subclass of `Value *`: an `AllocaInst *`:

```
AllocaInst *ptr = builder->CreateAlloca(Type::getInt32Ty(*
    ↪ context),
                                        /* Twine */ "p");
// AllocaInst has additional methods e.g. to query type
ptr->getAllocatedType(); // returns i32
builder->CreateLoad(ptr);
builder->CreateStore(someVal, ptr);
```

8.7 GLOBAL VARIABLES

Just as we `alloca` local variables on a stack, we can create global variables and `load` from them and `store` to them.

Global variables are declared at the start of a module, and are part of the module symbol table.


Global variables in LLVM IR
@global_var = global i32 1
@const_global_var = constant i64 5
Variable names start with an @
types!
global variables must be initialised with a constant
global variables can be declared constant - cannot update the value stored

We can use the `module` object to create named global variables, and to query them.

```
module->getOrInsertGlobal(globalVarName, globalVarType);
...
GlobalVariable *globalVar = module->getNamedGlobal(globalVarName);
```

Global variables **must** be initialised with a constant value (not a variable):

```
globalVar->setInitializer(initValue);
```

Alternatively we can do this in one command using the `GlobalVariable` constructor:

```
GlobalVariable *globalVar = new GlobalVariable(module,
    ↪ globalVarType, /*isConstant*/ false,
    ↪ GlobalValue::ExternalLinkage, initValue,
    ↪ globalVarName)
```

As before we can load and store:

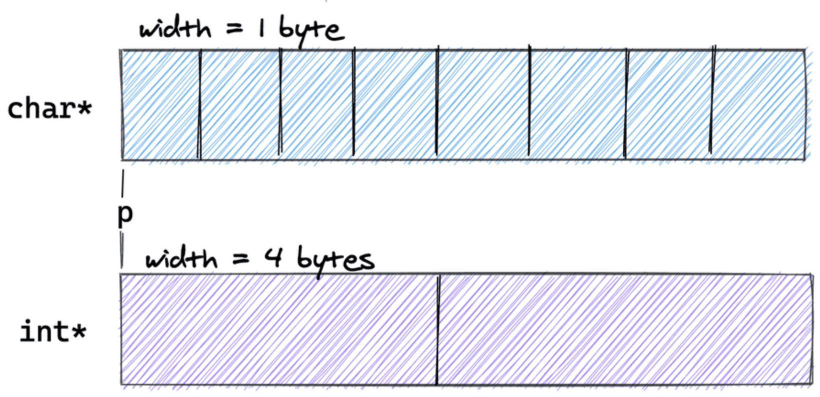
```
builder->CreateLoad(globalVar);
builder->CreateStore(someVal, globalVar); // not for consts!
```

8.8 GEPS

We get a **base pointer** to the aggregate type (array / struct) on the stack or in global memory, but what if we want a pointer to a **specific element**? We'd need to find the **pointer offset** of that element within the aggregate, and then add this to the base pointer to get the address of that element. Calculating the pointer offset is machine-specific e.g. depends on the size of the datatypes, the struct padding etc.

The Get Element Pointer (GEP) instruction is an instruction to apply the pointer offset to the base pointer and return the resultant **pointer.**

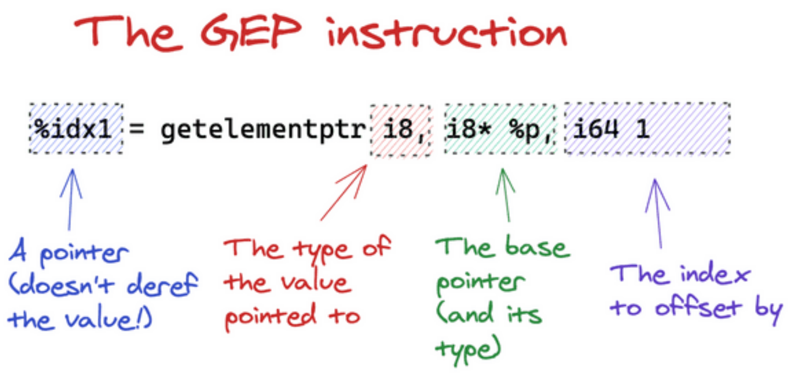
Consider two arrays starting at p. Following C convention, we can represent a pointer to that array as char* or int*.



Below we show the GEP instruction to calculate the pointer p+1 in each of the arrays:

```
// char = 8 bit integer = i8
%idx1 = getelementptr i8, i8* %p, i64 1 // p + 1 for char*
%idx2 = getelementptr i32, i32* %p, i64 1 // p + 1 for int*
```

This GEP instruction is a bit of a mouthful so here's a breakdown:


The GEP instruction
%idx1 = getelementptr i8, i8* %p, i64 1
A pointer (doesn't deref the value!)
The type of the value pointed to
The base pointer (and its type)
The index to offset by

This `i64 1` index adds **multiples** of the base type to the base pointer. `p+1` for `i8` would add 1 byte, whereas as `p+1` for `i32` would add 4 bytes to `p`. If the index was `i64 0` we'd return `p` itself.

The LLVM API instruction for creating a GEP is... `CreateGEP`.

```
Value *ptr = builder->CreateGEP(baseType, basePtr, arrayofIndices);
```

Wait? *Array* of indices? Yes, the GEP instruction can have multiple indices passed to it. We've looked at a simple example where we only needed one index.

Before we look at the case where we pass multiple indices, I want to reiterate the purpose of this first index:

A pointer of type `Foo *` can represent in C the base pointer of an array of type `Foo`. The first index adds multiples of this base type Foo to traverse this array.

8.8.1 *GEPS with Structs*

Okay, now let's look at structs. So take a struct of type `Foo`:

```
%Foo = type { i32, [4 x i32], i32}
```

We want to index **specific** fields in the struct. The natural way would be to label them field `0`, `1` and `2`. We can access field `2` by passing this into the GEP instruction as **another index**.

```
%ThirdFieldPtr = getelementptr  %Foo, %Foo* %ptr, i64 0, i64
    ↪  2
```

The returned pointer is then calculated as: `ptr + 0 * (size of Foo) + offset 2 * (fields of Foo)`.

For structs, you'll likely always pass the first index as `0`. The biggest confusion with GEPs is that this `0` can seem redundant, as we want the field `2`, so why are we passing a `0` index first? Hopefully you can see from the first example why we need that `0`. Think of it as passing to GEP the base pointer of an implicit `Foo` array of size 1.

To avoid the confusion, LLVM has a special `CreateStructGEP` instruction that asks only for field index (this is the `CreateGEP` instruction with a `0` added for you):

```
Value *thirdFieldPtr = builder->CreateStructGEP(baseStructType, basePtr, fieldIndex);
```

The more nested our aggregate structure, the more indices we can provide. E.g. for element index 2 of Foo's second field (the 4 element int array):

```
getelementptr  %Foo, %Foo* %ptr, i64 0, i64 1, i64 2
```

The pointer returned is: `ptr + 0 * (size of Foo) + offset 1 * (field of Foo) + offset 2 * (elems of array)`. (In terms of the corresponding API, we'd use `CreateGEP` and pass the array `{0,1,2}`.) A Good talk that explains GEP well:https://youtu.be/m8G_S5LwlTo?t=1753

8.9 mem2reg

If you remember, LLVM IR must be written in SSA form. But what happens if the Bolt source program we're trying to map to LLVM IR is not in SSA form? For example, if we're reassigning `x`:

```
let x = 1
x = x + 1
```

One option would be for us to rewrite the program in SSA form in an earlier compiler stage. Every time we reassign a variable, we'd have to create a fresh variable. We'd also have to introduce `phi` nodes for conditional statements. For our example, this is straightforward, but in general this extra rewrite is a pain we would rather not deal with.

```
// Valid SSA: assign fresh variables
let x1 = 1
    x2 = x1 + 1
```

We can use **pointers** to avoid assigning fresh variables. Note here we **aren't reassigning** the pointer `x`, just updating **the value it pointed to**. So this is valid SSA.

```
// valid SSA: use a pointer and update the value it points to
let x = &1;
   *x = *x + 1
```

This switch to pointers is a much easier transformation than variable renaming. It also has a really nice LLVM IR equivalent: allocating *stack memory* (and manipulating the pointers to the stack) instead of reading from *registers*.

So whenever we declare a local variable, we use `alloca` to get a pointer to freshly allocated stack space. We use the `load` and `store` instructions to read and update the value pointed to by the pointer:

```
%x = alloca i32
store i32 1, i32* %x

%1 = load i32, i32* %x
%2 = add i32 %1, 1
store i32 %2, i32* %x
```

Let's revisit the LLVM IR if we were to rewrite the Bolt program to use fresh variables. It's only *2* instructions, compared to the *5* instructions needed if using the stack. Moreover, we avoid the expensive `load` and `store` memory access instructions.

```
%x1 = 1
%x2 = add i32 %x1, 1   // let x2 = x1 + 1
```

So while we've made our lives easier as compiler writers by avoiding a rewrite-to-SSA pass, this has come at the expense of performance.

Happily, LLVM lets us have our cake and eat it.

LLVM provides a `mem2reg` optimisation that optimises stack memory accesses into register accesses. We just need to ensure we declare all our `alloca`s for local variables in the **entry basic block** for the function.

How do we do this if the local variable declaration occurs midway through the function, in another block? Let's look at an example:

```
// BOLT variable declaration
let x : int = someVal;
// translated to LLVM IR
%x = alloca i32
store i32 someVal, i32* %x
```

We can actually move the `alloca`. It doesn't matter where we allocate the stack space so long as it is allocated before use. So let's write the `alloca` at the very start of the parent function this local variable declaration occurs.

How do we do this in the API? Well, remember the analogy of the builder being like a file pointer? We can have multiple file pointers pointing to different places in the file. Likewise, we instantiate a new `IRBuilder` to point to the start of the `entry` basic block of the parent function, and insert the `alloca` instructions using that builder.

Listing 8.12: expr_codegen.cc

```
Function *parentFunction = builder->GetInsertBlock()
                                ->getParent();
// create temp builder to point to start of function
IRBuilder<> TmpBuilder(&(parentFunction->getEntryBlock()),
                        parentFunction->getEntryBlock().
                            ↪ begin());
// .begin() inserts this alloca at beginning of block
AllocaInst *var = TmpBuilder.CreateAlloca(boundVal->getType
    ↪ ());
// resume our current position by using orig. builder
builder->CreateStore(someVal, var);
```

Listing 8.13: ir_codegen_visitor.cc

```
std::unique_ptr<legacy::FunctionPassManager>
    ↪ functionPassManager =
      make_unique<legacy::FunctionPassManager>(module.get())
          ↪ ;

  // Promote allocas to registers.
  functionPassManager->add(createPromoteMemoryToRegisterPass
      ↪ ());
  // Do simple "peephole" optimizations
  functionPassManager->add(createInstructionCombiningPass())
      ↪ ;
```

```
// Reassociate expressions.
functionPassManager->add(createReassociatePass());
// Eliminate Common SubExpressions.
functionPassManager->add(createGVNPass());
// Simplify the control flow graph (deleting unreachable
    ↪ blocks etc).
functionPassManager->add(createCFGSimplificationPass());

functionPassManager->doInitialization();
```

We run this on each of the program's functions:

Listing 8.14: ir_codegen_visitor.cc

```
for (auto &function : functions) {
    Function *llvmFun =
     module->getFunction(StringRef(function->functionName));
    functionPassManager->run(*llvmFun);
  }
  Function *llvmMainFun = module->getFunction(StringRef("
    ↪ main"));
  functionPassManager->run(*llvmMainFun);
```

In particular, let's look at the the `factorial` LLVM IR output by our Bolt compiler before and after. You can find them in the repo:

Listing 8.15: factorial-unoptimised.ll

```
define i32 @factorial(i32) {
entry:
  %n = alloca i32
  store i32 %0, i32* %n
  %1 = load i32, i32* %n
  %eq = icmp eq i32 %1, 0
  br i1 %eq, label %then, label %else

then:                                              ; preds =
    ↪ %entry
  br label %ifcont

else:                                ; preds = %entry
  %2 = load i32, i32* %n
  %3 = load i32, i32* %n
  %sub = sub i32 %3, 1
  %4 = call i32 @factorial(i32 %sub)
  %mult = mul i32 %2, %4
  br label %ifcont

ifcont:                         ; preds = %else, %then
  %iftmp = phi i32 [ 1, %then ], [ %mult, %else ]
  ret i32 %iftmp
}
```

And the optimised version:

Listing 8.16: factorial-optimised.ll

```
define i32 @factorial(i32) {
entry:
  %eq = icmp eq i32 %0, 0
  br i1 %eq, label %ifcont, label %else

else:                                             ; preds =
    ↪ %entry
  %sub = add i32 %0, -1
  %1 = call i32 @factorial(i32 %sub)
  %mult = mul i32 %1, %0
  br label %ifcont

ifcont:                                           ; preds =
    ↪ %entry, %else
  %iftmp = phi i32 [ %mult, %else ], [ 1, %entry ]
  ret i32 %iftmp
}
```

Notice how we've actually got rid of the `alloca` and the associated `load` and `store` instructions, and also removed the `then` basic block!

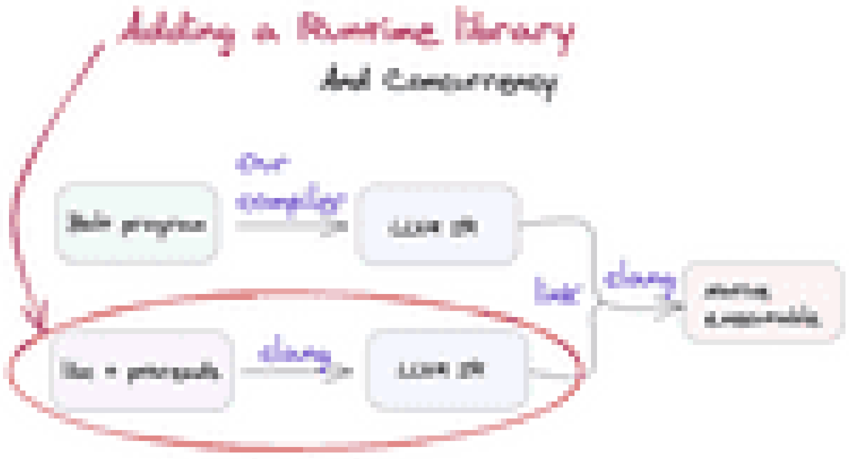
8.10 WRAP UP

This last example shows you the power of LLVM and its optimisations. You can find the top-level code that runs the LLVM code generation and optimisation in the main.cc file in the Bolt repository.

In the next few posts we'll be looked at some more advanced language features: generics, inheritance and method overriding and concurrency! Stay tuned for when they come out!

9 IMPLEMENTING CONCURRENCY AND OUR RUNTIME LIBRARY

December 28, 2020



9.1 THE ROLE OF A RUNTIME LIBRARY

Up till now, we've translated constructs in our language Bolt directly into LLVM IR instructions. The biggest misconception I had with concurrency was that it worked in the same manner. That there was some `spawn` instruction that compilers could emit that would create a new thread. This isn't the case. LLVM doesn't have a single instruction to create threads for you. Why not, you ask? What's so special about threads?

Creating a thread is a complex routine of instructions that are **platform-specific**. Threads are managed by the OS kernel, so creating a thread involves creating system calls following that platform's conventions.

LLVM draws a line here. There's a limit to how much functionality LLVM can provide without itself becoming *huge*. Just think of the variety of platforms out there that it would have to support, from embedded systems to mobile to the different desktop OSs.

Enter your **runtime library**. Your language's runtime library provides the *implementation* for these routines that interact with the platform / runtime environment. The compiler inserts calls to these runtime library functions when compiling our Bolt program. Once we have our compiled LLVM IR, we **link** in the runtime library function implementations, so the executable can call these.

You know what the best part is about compiling to LLVM IR? We don't have to write our own runtime library. C compiles to LLVM IR. Let's just use C functions to bootstrap our runtime library and `clang` to link them in!

So which kinds of functions are present in our runtime library?

- I/O e.g. `printf`
- Memory management. (Remember in the previous post I mentioned that the LLVM didn't provide a heap.) Either implemented manually (`malloc` and `free`) or through a garbage collection algorithm (e.g. *mark-and-sweep*).

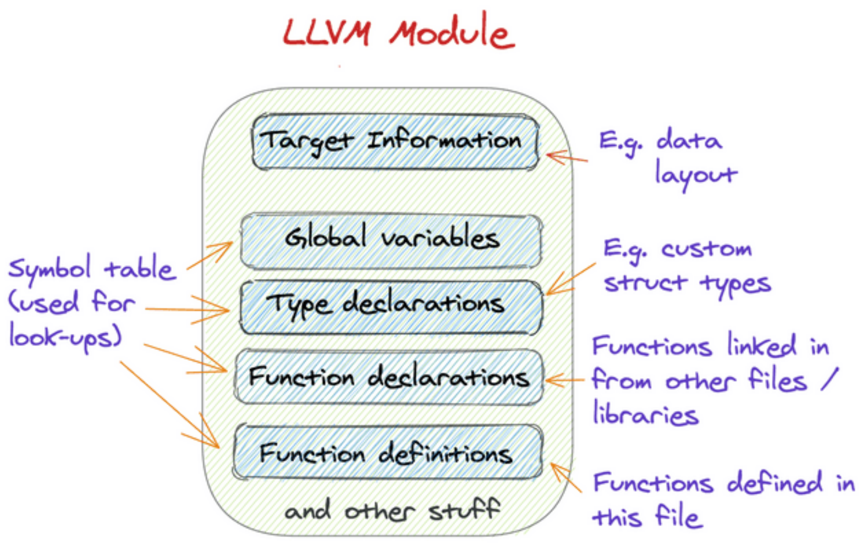
- And of course **threads** via the C pthread API. Pthread is short for POSIX Thread, a standardised API for these thread system calls.

We aren't just limited to the functions provided by C. We can write our own C functions and link them in using the techniques described in this post.

We'll look at `printf`

9.2 RECAP: LLVM MODULE

Here's a quick sketch of the structure of an LLVM module (from the previous post). As the diagram shows, the part of the module we're interested in is the **function declarations**. To use a C library function, we need to insert its function signature in our module's function declarations symbol table.



9.3 PRINTF

Let's warm up with `printf`. The C function type signature is:

```
int printf ( const char* format, ... );
```

To translate this C type signature to an LLVM `FunctionType`:

- drop the `const` qualifier
- Convert C types to equivalent LLVM types: `int` and `char` map to `i32` and `i8` respectively
- the `...` indicates `printf` is variadic. So the LLVM API code is as follows:

Listing 9.1: extern_functions_codegen.cc

```
module->getOrInsertFunction( "printf",  FunctionType::get(
    ↪    IntegerType::getInt32Ty(*context),    Type::
    ↪ getInt8Ty(*context)->getPointerTo(),    true /* this
    ↪ is variadic func */  ));
```

And the corresponding code to call the `printf` function:

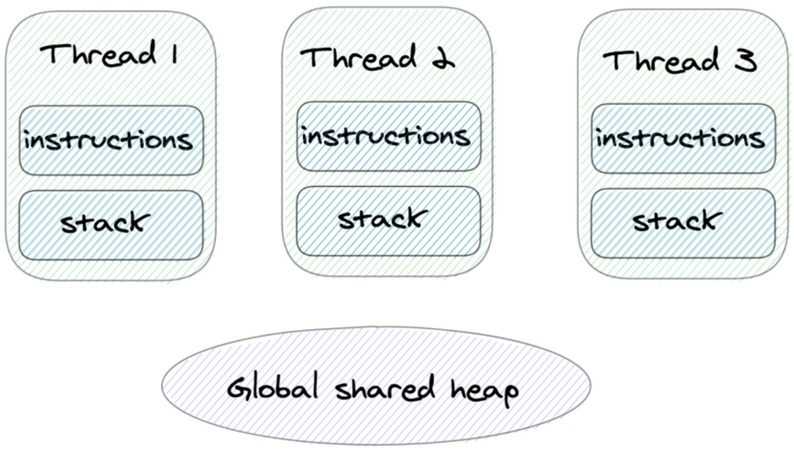
Listing 9.2: expr_codegen.cc

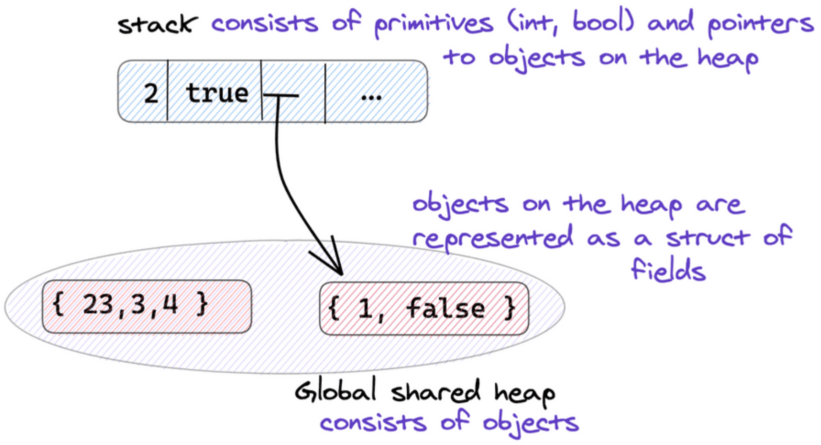
```
Value *IRCodegenVisitor::codegen(const ExprPrintfIR &expr) {
  Function *printf = module->getFunction("printf");
  std::vector<Value *> printfArgs;
  Value *formatStrVal = builder->CreateGlobalStringPtr(expr.
      ↪ formatStr);
  printfArgs.push_back(formatStrVal);
  // add variadic arguments
  for (auto &arg : expr.arguments) {
    printfArgs.push_back(arg->codegen(*this););
  }
  return builder->CreateCall(printf, printfArgs);
};
```

CreateGlobalStringPtr is a useful `IRBuilder` method that takes in a string and returns an `i8*` pointer (so we have a argument of the right type).

9.4 THE BOLT MEMORY MODEL

Each thread has **its own stack**. To share objects between threads, we'll introduce a **global heap** of objects. We'll use the stack to store primitives like ints, bools and to store pointers to objects on the heap.




stack consists of primitives (int, bool) and pointers to objects on the heap
2
true
...
objects on the heap are represented as a struct of fields
{ 23,3,4 }
{ 1, false }
Global shared heap consists of objects

9.4.1 *Malloc*

We can use `malloc` to allocate objects to the heap. (This is what C++'s `new` keyword does under the hood!)

The type signature for `malloc` is as follows:

```
void *malloc(size_t size);
```

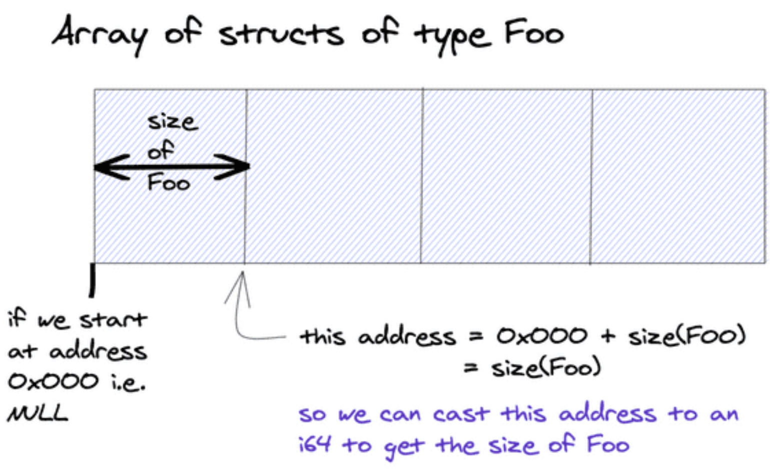
Converting this to the equivalent LLVM IR types, `void *` and `size_t` map to `i8 *` and `i64`. The LLVM API code falls out once you've determined the LLVM types.

Listing 9.3: extern_functions_codegen.cc

```
Type *voidPtrTy = Type::getInt8Ty(*context)->getPointerTo();
module->getOrInsertFunction(
  "malloc",  FunctionType::get(
    voidPtrTy,
    IntegerType::getInt64Ty(*context),
    /* has variadic args */ false  ));
```

There is just one issue though. When we create a `struct` on the heap, `malloc` requires us to specify the number of bytes we want to allocate. However the size of that `struct` is machine-specific information (it depends on the size of datatype, struct padding etc.). How do we do this in LLVM IR, which is machine-independent?

We can compute this through the following *hack*. We know that an array of structs of type `Foo` is just a contiguous block of memory. Pointers to adjacent indices are `size(Foo)` bytes apart. Therefore, if we start an array at address `0x0000` (the special `NULL` address) then the first index of the array is at address `size(Foo)`.


Array of structs of type Foo
size of Foo
if we start at address 0x000 i.e. NULL
this address = 0x000 + size(Foo) = size(Foo)
so we can cast this address to an i64 to get the size of Foo

We can use the `getelementptr` (GEP) instruction to compute this array address, and pass in the base pointer of the array as the `null` value. You might be thinking, hold on, this array doesn't exist. Won't this cause a seg fault?

Remember, the role of the GEP instruction is just to calculate pointer offsets. Not to check if the resultant pointer is valid. Not to actually access the memory. Just to perform this calculation. No memory access = no seg fault.

Listing 9.4: expr_codegen.cc

```
// calculate index of array[1] using GEP instructionValue *
    ↪ objDummyPtr = builder->CreateConstGEP1_64(
    ↪ Constant::getNullValue(objType->getPointerTo()), 1, "
    ↪ objsize");// cast to i64 for mallocValue *objSize =
    ↪     builder->CreatePointerCast(objDummyPtr,Type::
    ↪ getInt64Ty(*context));
```

We pass `objSize` to `malloc`. `malloc` returns a `void *` pointer, however since we will later want to access the struct's fields, we need to cast the type to `objType*`. Remember, LLVM needs explicit types!

Listing 9.5: expr_codegen.cc

```
// allocate the object on the heap
Value *objVoidPtr =
    builder->CreateCall(module->getFunction("malloc"),
        ↪ objSize);
// cast (void *)  to  (objType *)
Value *obj =    builder->CreatePointerCast(objVoidPtr,
    ↪ objType->getPointerTo());
```

9.4.2 *Bonus: Garbage Collection*

Swap out `malloc` for `GC malloc`. If we use GC_malloc, we get garbage collection for free! How cool is that?! We don't need to `free()` our object.

If you want to implement your own garbage collector, check this LLVM page out.

9.5 IMPLEMENTING HARDWARE THREADS WITH PTHREADS

So far we've looked at `printf` and `malloc`. We've found that the biggest hurdle to declaring their function signatures is translating C types to LLVM IR types. Once you have the LLVM IR types, everything falls out. With the `pthread` API the process is the same, only the translation from C types to LLVM IR types is a little more involved.

9.5.1 *Understanding the Pthread API*

The two functions we'd like to use to create and join threads are pthread_create and pthread_join. The linked Linux manual pages give a full description of the functions, but they are a bit dense.

Let's unpack the relevant information, starting with the function signatures:

```c
int pthread_create(pthread_t *thread, const pthread_attr_t *
    ↪ attr,
                          void *(*start_routine) (void *),
                              ↪ void *arg);
int pthread_join(pthread_t thread, void **retval);
```

Both `pthread_create` and `pthread_join` are idiomatic C functions in that:

- they return an `int`, where the value `0` = success and other values = error codes.
- to return additional values e.g a `val` of type `Foo`, we pass in a *pointer* `p` of type `Foo*` as an argument. The function will update the pointer's value to that returned value (`*p=val`). We can then access the returned value by dereferencing the pointer (`*p`). See this tutorial if you're not familiar with this *"pass-by-pointer"* pattern.

If you're not familiar with C pointer syntax, `void *(*start_routine) (void *)` is quite a mouthful. This says `start_routine` is a pointer to a function that takes in a `void *` argument and returns a `void *` value. `void *` is the generic type representing *any* pointer (it's super flexible - we can cast it to any type we'd like e.g. `int *` or `Foo *`).

`pthread_create` creates a thread, which will asynchronously execute the function pointed to by `start_routine` with the `arg` argument. The opaque `pthread_t` type represents a handle to a thread object (can think of it like a thread id). We pass a `pthread_t *` pointer, and `pthread_create` will assign the `pthread_t` handle corresponding to the created thread to this object. The opaque `pthread_attr_t` type represents any attributes we want the thread to have. The `pthread_attr_t *` parameter lets us specify the attributes we want the created thread to have. Passing `NULL` will initialise the thread with default attributes, which is good enough for us.

We pass the `pthread_t` handle to `pthread_join` to tell it which thread we are joining (waiting on to finish). `pthread_join` updates the `void **` pointer parameter with the `void *` return value of `start_routine(arg)` executing on that thread. We can pass `NULL` if we don't want this return value.

Here's an excellent minimal C example that demonstrates the use of Pthreads.

9.5.2 *Translating Pthread types into LLVM IR*

We've seen `int` and `void *` before: they may to `i32` and `i8*`. `void **` follows as `i8*`. We're in a bit of a pickle with `pthread_t` and `pthread_attr_t` as their type definitions are opaque.

Aw shucks, we're stuck. The solution (as with most cases when you're stuck with LLVM IR) is to **experiment in C and look at the compiled LLVM IR output**.

We can compile that excellent minimal C example to LLVM IR using `clang`. The command to do this for a `foo.c` file is:

```
clang -S -emit-llvm -O1 foo.c
```

The Clang LLVM IR output is quite messy. The best way to read the output is to find the lines of LLVM IR that correspond to the interesting lines of code in the C program, and ignore the noise around them. More information about experimenting with C and C++ to understand LLVM IR in this excellent Reddit comment.

For us, the interesting lines are those that allocate a `pthread_t` stack variable, and the `pthread_create` and `pthread_join` calls:

Listing 9.6: C

```
// C
pthread_t inc_x_thread;
...
pthread_create(&inc_x_thread, NULL, inc_x, &x)
...
pthread_join(inc_x_thread, NULL)
```

Listing 9.7: LLVM IR

```
%4 = alloca %struct._opaque_pthread_t*, align 8...
%9 = call i32 @pthread_create(%struct._opaque_pthread_t** %4
    ↪ , %struct._opaque_pthread_attr_t* null, i8* (i8*)*
    ↪ @inc_x, i8* %8)...
%23 = call i32 @pthread_join(%struct._opaque_pthread_t* %22,
    ↪  i8** null)
```

If we match up our type definitions for the functions:

```
// C type -> LLVM IR
typepthread_t = %struct._opaque_pthread_t*pthread_attr_t =
    ↪ %struct._opaque_pthread_attr_t
```

Great, we've determined `pthread_t` is a pointer to a struct of type `%struct._opaque_pthread_t`. What's the type of this struct? Let's look at the type definitions defined earlier in the file:

```
struct.__sFILE = type { i8*, i32, i32, i16, i16, %struct.
    ↪ __sbuf, i32, i8*, i32(i8*)*, i32 (i8*, i8*, i32)*,
    ↪ i64 (i8*, i64, i32)*, i32 (i8*, i8*, i32)*, %struct.
    ↪ __sbuf, %struct.__sFILEX*, i32, [3 x i8], [1 x i8],
    ↪ %struct.__sbuf, i32, i64 }%struct.__sFILEX = type
    ↪ opaque%struct.__sbuf = type { i8*, i32 }%struct.
    ↪ _opaque_pthread_t = type { i64, %struct.
    ↪ __darwin_pthread_handler_rec*, [8176 x i8] }%struct.
    ↪ __darwin_pthread_handler_rec = type { void (i8*)*, i8
    ↪ *, %struct.__darwin_pthread_handler_rec* }%struct.
    ↪ _opaque_pthread_attr_t = type { i64, [56 x i8] }
```

Yikes, this is a mess. Here's the thing. We don't have to declare the internals of the struct because we **aren't using them** in our program. So just as `%struct.__sFILEX` was defined as an opaque struct above, we can define our own opaque structs. The `pthread` library's files will specify the bodies of the struct types as it actually manipulates their internals.

```
Type *pthread_t = StructType::create(*context, "
    ↪ struct_pthread_t") ->getPointerTo();
Type *pthread_attr_t = StructType::create(*context,"
    ↪ struct_pthread_attr_t")
```

The eagle-eyed amongst you might notice these struct names don't match the names in the file e.g. `struct_pthread_t` vs `struct._opaque_pthread_t`. What gives?

LLVM's types are resolved **structurally** not by name. So even if our program has two separate structs `Foo` and `Bar`, if the types of their fields are the same, LLVM will treat them the same. The name doesn't matter - we can use one in place of the other without any errors:

```
// Foo == Bar
%Foo = type {i32, i1}
%Bar = type {i32, i1}
```

It turns out we can exploit the structural nature of LLVM's type system to simplify our types further.

See `pthread_attr_t` is only used in one place: `pthread_create`, and there we pass `NULL` as the `pthread_attr_t *` argument. `NULL` is the same value regardless of type, so rather than defining the type `pthread_attr_t *`, we can use `void *` to represent a generic `NULL` pointer.

Let's look at `pthread_t` next. We know that `pthread_t` is a **pointer** to some opaque struct, but we never access that struct anywhere in our program. In fact, the only place `pthread_t`'s type matters is when we're allocating memory on the stack for it - we need to know the type to know how many bytes to allocate.

Listing 9.8: expr_codegen.cc

```
Type *pthreadTy = codegenPthreadTy();Value *pthreadPtr =
    ↪ builder->CreateAlloca(pthreadTy, nullptr, "pthread");
```

Here's the thing: *all* pointers have the same size, regardless of type, as they all store memory addresses. So we can use a generic pointer type `void *` for `pthread_t` too.

Listing 9.9: extern_functions_codegen.cc

```
Type *IRCodegenVisitor::codegenPthreadTy() {   return Type::
    ↪ getInt8Ty(*context)->getPointerTo();}
```

The LLVM API code to declare the `pthread_create` and `pthread_join` functions is therefore as follows:

Listing 9.10: extern_functions_codegen.cc

```
Type *voidPtrTy = Type::getInt8Ty(*context)->getPointerTo();
Type *pthreadTy = codegenPthreadTy();
Type *pthreadPtrTy = pthreadTy->getPointerTo();
// (void *) fn (void * arg)
FunctionType *funVoidPtrVoidPtrTy = FunctionType::get(
    ↪ voidPtrTy, ArrayRef<Type *>({voidPtrTy}),  /* has
    ↪ variadic args */ false);
// int pthread_create(pthread_t * thread, const
    ↪ pthread_attr_t * attr,
//                    void * (*start_routine)(void *), void *
    ↪ arg)
// we use a void * in place of pthread_attr_t *
FunctionType *pthreadCreateTy = FunctionType::get(  Type::
    ↪ getInt32Ty(*context),  ArrayRef<Type *>({pthreadPtrTy
    ↪ , voidPtrTy,                 (funVoidPtrVoidPtrTy)->
    ↪ getPointerTo(),                voidPtrTy}),  /* has
    ↪ variadic args */ false);
module->getOrInsertFunction("pthread_create",
    ↪ pthreadCreateTy);
// int pthread_join(pthread_t thread, void **value_ptr)
FunctionType *pthreadJoinTy = FunctionType::get(  Type::
    ↪ getInt32Ty(*context),  ArrayRef<Type *>({pthreadTy,
    ↪ voidPtrTy->getPointerTo()}),  /* has variadic args */
    ↪  false);
module->getOrInsertFunction("pthread_join", pthreadJoinTy);
```

9.6 linking in the runtime library

We can use `clang` to link in the libraries when we compile our `foo.ll` file to a `./foo` executable.

We link in `pthread` with the `-pthread` flag.

To link `GC_malloc` we need to do two things:

- Include its header files (here they're in the folder `/usr/local/include/gc/`). We use the `-I` flag to add its folder.
- Add the static library `.a` file: `/usr/local/lib/libgc.a` to the list of files being compiled.

compile_program.sh
Copy

```
clang -O3 -pthread -I/usr/local/include/gc/ foo.ll  /usr/local/lib/libgc.a  -o foo
```

9.7 generic approach to bootstrapping with c functions

We've seen 3 examples of C functions used when bootstrapping our runtime library: `printf`, `malloc` and the `pthread` API. The generic approach (can skip steps 2 - 4 if you already understand the function)

1. Get the C function type signature

2. Write a minimal example in C
3. Compile the C example to LLVM IR and match up the key lines of your example with the corresponding lines of IR e.g. function calls
4. Simplify any opaque types into more generic types: only define as much type info as you need! E.g. if you don't need to know it's a `struct *` pointer (because you're not loading it or performing GEP instructions), use a `void *`.
5. Translate the C types into LLVM IR types
6. Declare the function prototype in your module
7. Call the function in LLVM IR wherever you need it!
8. Link the function in when compiling the executable

9.8 IMPLEMENTING CONCURRENCY IN BOLT

9.8.1 *The Finish-Async Construct*

Bolt uses the `finish-async` construct for concurrency. The `async` keyword spawns (creates) a thread and sets it to execute the expressions in that `async` block. The `finish` block scopes the lifetime of any threads spawned inside it using the `async` keyword - so all spawned threads must **join** at the end of the block. This way we've clearly defined the lifetime of our threads to be that of the `finish` block.

Listing 9.11: example_concurrent_program.bolt

```
before();
finish{
  async{
    f();
  }
  async{
    g();
    h();
  }
  during();
}
after();
```

Illustrating the execution graphically:

9.8.2 *Creating our Threads*

COMPUTE FREE VARIABLES When inside our `async` block, we can access any **objects** in scope at the start of the `finish` block (before the `async` thread was spawned). For example:

example_concurrent_program.bolt

Copy

```
let a = new Foo();
let b = new Bar();
let y = true;
let z = 2;
finish{
  async{
```

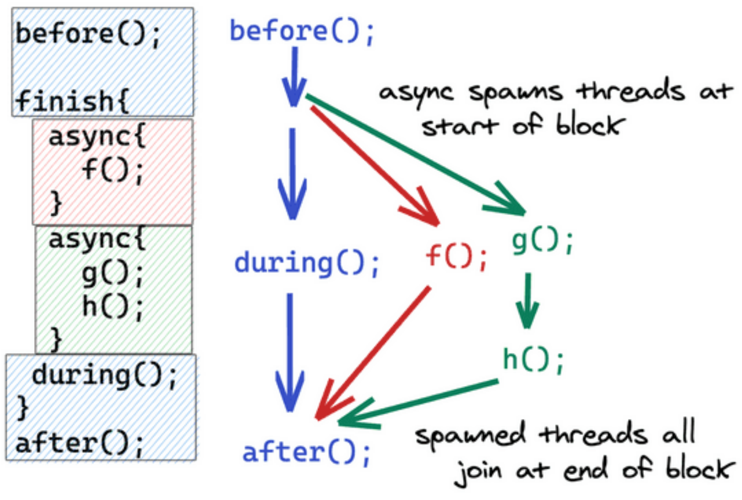


Figure 2: A Bolt concurrent program illustrated – each colour represents a different thread.

```
    // This thread accesses a, b
    let w = 1;
    f(a, b, w);
  }
  ...}
```

However, there's an issue. In Bolt's memory model, each thread has its own stack. The `let a = ...` definition occurs on the main thread, so the pointer to `a`'s object is stored on the main thread's stack. Likewise for `b`. When we spawn the second thread using `async`, this new thread has its own stack, which is empty (and so doesn't contain `a` or `b`).

The first step is to compute the free variables that need to be copied across to the new stack. Here's a link to the code if you're interested; we'll skip the details as it's quite mechanical. Link to the previous post on desugaring if you want to look back at the desugaring stage.

```
async{
  //  a, b are free variables
    let w = 1;
    f(a, b, w);
  }
```

CONVERTING THE ASYNC BLOCK INTO A FUNCTION CALL Now we need to somehow convert this expression into a function that `pthread_create` can run.

```
async{
    let w = 1;
    f(a, b, w);  }
//  need to convert this to a function
void *asyncFun(void *arg){
  let w = 1;
  f(a, b, w);
  return null;
 // we return null but
 // you could return the last value instead
}
```

Since we need all the variables in the function body to be defined, we need to pass the free variables as arguments to the function:

```
function void *asyncFun(Foo a, Bar b){
  let w = 1;
  f(a, b, w);
}
let a = new Foo();
let b = new Bar();
let y = true;
let z = 2;
finish{  asyncFun(a, b);  ...}
```

However, this doesn't match the argument type we're looking for: `asyncFun` can only take a *single* `void*` argument.

The solution: create a *single struct* that contains all of the values. We can cast the `struct *` to and from a `void *` pointer to match types.

Copy

```
ArgStructType *argStruct = {a, b};// we can cast ArgStructType * to void *asyncFun(argStr
```

Great, now we have the `void *asyncFun(void *argStruct)` function type, as `pthread_create` requires.

We need to unpack this inside the function:

```
function void *asyncFun(void * arg){
  // cast pointer
  ArgStructType *argStruct = (ArgStructType *) arg;
  // unpack variables
  let a = argStruct.a;
  let b = argStruct.b;
  // execute body of function
  let w = 1;
  f(a, b, w);
}
```

CREATING PTHREADS Finally, having defined our arg and our async function, we can invoke `pthread_create`.

The high-level structure of this is as follows:

Listing 9.12: pthread_codegen.cc

```
// create async function and argument
  StructType *argStructTy = codegenAsyncFunArgStructType(freeVarList);
  Value *argStruct = codegenAsyncFunArgStruct(asyncExpr, argStructTy);
  Function *asyncFun = codegenAsyncFunction(asyncExpr, argStructTy);
  ...
  // spawn thread
  Function *pthread_create =
      module->getFunction("pthread_create");
  Value *voidPtrNull = Constant::getNullValue(
      Type::getInt8Ty(*context)->getPointerTo());
  Value *args[4] = {
      pthread,
      voidPtrNull,
      asyncFun,
      builder->CreatePointerCast(argStruct, voidPtrTy),  };
  builder->CreateCall(pthread_create, args);
```

codegenAsyncFunArgStructType, codegenAsyncFunArgStruct and codegenAsyncFunction just implement the steps we've outlined in prose.

9.8.3 *Joining Pthreads*

We join each of the pthread_t handles for each of the async expressions' threads. As we mentioned earlier, we aren't returning anything from the asyncFun, so we can pass in NULL as the second argument:

Listing 9.13: pthread_codegen.cc

```
void IRCodegenVisitor::codegenJoinPThreads(
    const std::vector<Value *> pthreadPtrs) {
  Function *pthread_join =
      module->getFunction("pthread_join");
  Type *voidPtrPtrTy =
      Type::getInt8Ty(*context)->getPointerTo()->
          ↪ getPointerTo();
  for (auto &pthreadPtr : pthreadPtrs) {
    Value *pthread = builder->CreateLoad(pthreadPtr);
    builder->CreateCall(pthread_join,
                        {pthread, Constant::getNullValue(
                            ↪ voidPtrPtrTy)});
  }
```

9.8.4 *Implementing Finish-Async in LLVM IR*

Now we've talked about how we create threads and how we join threads, we can give the overall code generation for the finish-async concurrency construct:

Listing 9.14: expr_codegen.cc

```
Value *IRCodegenVisitor::codegen(
    const ExprFinishAsyncIR &finishAsyncExpr) {
  std::vector<Value *> pthreadPtrs;
  // spawn each of the pthreads
  for (auto &asyncExpr : finishAsyncExpr.asyncExprs) {
    Type *pthreadTy = codegenPthreadTy();
    Value *pthreadPtr =
        builder->CreateAlloca(pthreadTy, nullptr, Twine("
            ↪ pthread"));
    pthreadPtrs.push_back(pthreadPtr);
    codegenCreatePThread(pthreadPtr, *asyncExpr);  };

  // execute the current thread's expressions
  Value *exprVal;
  for (auto &expr : finishAsyncExpr.currentThreadExpr) {
    exprVal = expr->codegen(*this);
  }
  // join the threads at the end of the finish block
  codegenJoinPThreads(pthreadPtrs);
  return exprVal;
```

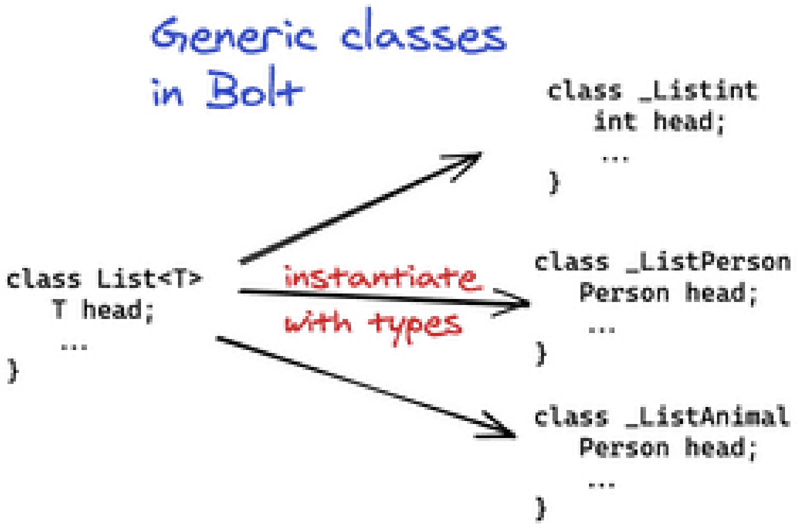
9.9 WRAP UP

In this post, we've looked at the role of a runtime library and how we can bootstrap our Bolt runtime with C functions. We looked at a generic way of adding C function declarations to our modules, and then link them with our compiled `.ll` file. I'd encourage you to add further C function to the runtime library. e.g. `scanf` to go with our `printf` function.

The second half of this post was a deep-dive into how Bolt does concurrency. We've used `pthread` to spawn a hardware thread for each `async` expression. An extension might be to use a thread pool instead!

GENERICS - ADDING POLYMORPHISM TO BOLT

January 23, 2021



Onward with more features that any “proper” programming language needs. Today we’re implementing **generics**. Generics allow you to reuse code for multiple types. Take a `List` for example. A list has the same operations regardless of types: it’d be a pain to write out a new class for each list.

```
class ListInt{  ...}
class ListPerson{  ...}
class ListAnimal{  ...}
```

The generic class for that would be `List<T>`. We call `T` the generic type *parameter*. Think of it like a variable, which we assign a type to when we instantiate the class: `List<int>()`. Let’s build it! We’d like to compile this program:

```
class List<T>{
    void add(T a){    ...  }
    T getHead(){    ...  }
    int size(){    ...  }
}

void main(){
  let list1 = new List<int>();
  list1.add(4);
  ...
}
```

10.1 JUST GIVE ME THE CODE!

As ever, the code is in the Bolt repo. The generics are handled in the typing and desugaring stages of the compiler. The code is in the files that contain `generics` in their name e.g. `type_generics.ml`. You could even just search "generic" in the repo!

10.2 TYPE PARAMETERS ARE JUST LIKE OTHER TYPES!

Rejoice, we don't need to rewrite our type-checker! This tutorial is much shorter than you think. Here's all that changes in the type-checker:

- **Within** our generic class, we can treate our type parameter `T` as an opaque type `TEGeneric`. So type-check the class as before, just don't make any assumptions about `T`.
- Outside a generic class we can't use generic types `T`, so raise an error if we see that.
- Whenever we use an object instantiated with a type, e.g. `List<int>`, we can replace all occurrences of `T` with the instantiated type `int`!

That's all that's changed. Seriously!

10.2.1 *Treat the generic type parameter as an opaque type*

We've added to the list of Bolt types a `TEGeneric` type to represent this opaque type `T`.

When we call objects of generic classes, we don't have an object of just `List`, it's `List<int>`, `List<Person>` etc. So we update class types `TClass` to carry around this instantiated type parameter `int`, `Person` etc. if they're generic.

Listing 10.1: ast_types.ml

```ocaml
type type_expr =  | TEInt
                 | TEClass  of Class_name.t * type_expr option
                   (** optionally specify instantiation of type parameter *)
                 | TEVoid  | TEBool  | TEGeneric
```

Next we need to update the `class_defn` type to distinguish between non-generic and generic classes. We define a special type as I think it's more instructive to see `Some Generic | None` rather than `true | false`. As before, ignore the `capability list` if you're not interested in the data-race prevention! (See my dissertation for a full explanation if you are).

Listing 10.2: ast_types.ml

```ocaml
type generic_type = Generic
```

Listing 10.3: parsed_ast.ml

```ocaml
type class_defn = TClass of     Class_name.t     * generic_type option      * capability list (* for data-races (see dissertation) *)      * field_defn list      * method_defn list
```

And now, within a generic class `List`, we instantiate `this` to be of type `List<T>` (remember we treat `T` as an opaque type `TEGeneric`):

Listing 10.4: type_generics.ml

```
let instantiate_maybe_in_generic_class_this (Parsed_ast.TClass (class_name, maybe_in_generic_class, _, _, _, _)) =
  let maybe_type_param = (* use generic type T inside class *)
    match maybe_in_generic_class with
     Some Generic -> Some TEGeneric
    | None -> None in (Var_name.of_string "this", TEClass (class_name, maybe_type_param))
```

And then the type-checking works as before!

10.2.2 *Check usage of generic types*

Outside a generic class we can't use generic types. I hate to bore you, this code is quite mechanical - it's a lot of recursively going through each of the subexpressions. For a class, check each of the fields, methods etc. For a function, check its type signature and then its body. And so on.

Here's a snippet of a function that checks a type. If we're in a generic class it's all fine, otherwise check we aren't using a generic type. Click the link to the `type_generics.ml` file below to see the full code.

Listing 10.5: type_generics.ml

```
let rec type_generics_usage_type type_expr maybe_in_generic_class error_prefix_str =
  match maybe_generic with
  | Some Generic -> Ok () (* can have generics in generic class *)
  | None -> ( (* recursively check there aren't nested uninitialised type parameters *)
    match type_expr with
    | TEInt | TEBool | TEVoid     -> Ok ()
    | TEGeneric                   ->     Error (Error.of_string     (Fmt.str "%s Type error: Use of generic type but not in a generic class@."     error_prefix_str))
    | TEClass (_, maybe_type_param) -> (
      match maybe_type_param with
      | Some type_param ->     type_generics_usage_type type_param maybe_in_generic_class error_prefix_str
      | None            -> Ok () ) )
```

10.2.3 *Instantiate generic objects*

We check first that we should be instantiating with a type-parameter. If we're trying to instantiate a non-generic class with a type param, raise an Error, and likewise if we haven't provided a concrete type for a generic class, raise an error. If we do have a generic class, then recursively replace all instances of a generic type with the concrete type: the fields and then the methods etc. Again, full details are in the repo:

Listing 10.6: type_generics.ml

```
let instantiate_maybe_generic_class_defn maybe_type_param
    ↪    ( Parsed_ast.TClass        (class_name,
    ↪ maybe_generic, caps, field_defns, method_defns) as
    ↪    class_defn ) loc =
match (maybe_generic, maybe_type_param) with
  | None, None  (* non-generic class *) ->  Ok class_defn
  | None, Some type_param          ->   Error ...
  | Some Generic, None             -> Error ...
  | Some Generic, Some type_param ->   List.map ~f:(
      ↪ instantiate_maybe_generic_field_defn type_param)
      ↪ field_defns      |> fun instantiated_field_defns ->
      ↪       List.map ~f:(
      ↪ instantiate_maybe_generic_method_defn type_param)
      ↪ method_defns      |> fun instantiated_method_defns
      ↪ ->      Ok        (Parsed_ast.TClass           (
      ↪ class_name            , maybe_generic           ,
      ↪ caps             , instantiated_field_defns
      ↪           , instantiated_method_defns ))
```

10.3 DESUGARING GENERICS

Ok, so we've type-checked our generics, and they pass our checks. What now? What do we tell our LLVM compiler backend to do when it encounters a T? You can't allocate a "generic" block of memory.

So we *desugar* away all mentions of generic types. What the compiler backend doesn't know about, it doesn't have to deal with.

Remember, we did this for function overloading in our desugaring post:

```
function int test(int f) {  ...}
function int test(bool b){  ...}
// DESUGARED (name-mangle functions)
function int testi(int f) {  ...}
function int testb(bool b){  ...}
```

The compiler backend doesn't need to worry about multiple functions with the same name, because we handled it in the desugaring stage.

Remember how I said it'd be a pain to write out a new class for each list? It would be *for us*, as we're doing it by hand. It isn't for the compiler: it can automate it! To avoid any name-clashes, we'll prepend each compiler-generated class with an _.

```
class _Listint{  ...}
class _ListPerson{  ...}
class _ListAnimal{  ...}
```

So our desugaring stage has 3 steps to handle generics:

- Count all instantiations of generics

- Create a special class for each of the instantiations (identical to how we instantiated generic objects earlier)
- Replace each generic class' constructor with its instantiated class. So `List<int>` goes to the class `_Listint`.

As before, let's dive into the code!

10.3.1 *Count all instantiations of generics*

This is in the `count_generics_instantiations.ml` file in the repo (creative name I know!).

We go through the code recursively, and every time we see a constructor with a concrete type param e.g. `List<int>`, we add that instantiation `int` to the total instantiations. In the code below, `class_insts` is a list containing pairs `(class_name, list_of_types_instantiated_with)`:

Listing 10.7: count_generics_instantiations.ml

```
let rec count_generics_instantiations_expr class_defns expr
    ↪ class_insts =  match expr with   | Typed_ast.
    ↪ Constructor (_, class_name, maybe_type_param,
    ↪ constructor_args) ->       ( match maybe_type_param
    ↪ with      | Some TEGeneric   -> class_insts (* only
    ↪ consider concrete type params *)       | Some
    ↪ type_param -> add_instantiation class_defns
    ↪ type_param class_name class_insts      | None
    ↪            -> class_insts )   ... (* recursive calls
    ↪ *)
```

An aside: we can be overly conservative with our counting, as if we instantiate classes that don't actually get used, then LLVM will optimise them away. So we could have brute-forced all possible combinations – this would have slowed the compiler down, but it wouldn't have affected the code output.

10.3.2 *Replace generic classes with instantiated classes*

The first step is to replace the class definitions: below we instantiate all the generic classes with concrete types, then filter the original generic classes out and return the updated list of classes.

Listing 10.8: replace_generic_with_instantiated_class_defns.ml

```
let replace_generic_with_instantiated_class_defns
    ↪ class_defns class_insts =  List.map    ~f:(fun (
    ↪ class_name, type_params) ->      List.find_exn
    ↪         ~f:(fun (Typed_ast.TClass (name, _, _, _, _, _
    ↪ )) -> name = class_name)         class_defns      |>
    ↪ fun class_defn -> instantiate_generic_class_defn
    ↪ type_params class_defn)    class_insts   |> fun
    ↪ instantiated_class_defns ->  (* get rid of
    ↪ uninitialised generic classes *)  List.filter    ~f:(
    ↪ fun (Typed_ast.TClass (_, maybe_generic, _, _, _, _))
    ↪  ->       match maybe_generic with Some Generic ->
    ↪ false | None -> true)    class_defns   |> fun
    ↪ non_generic_class_defns ->  List.concat (
    ↪ non_generic_class_defns :: instantiated_class_defns)
```

We then need to replace all references to generic classes in the program with the special instances (we name-mangle them). Inside the compiler we convert `List<int>` to `_Listint`. Again, the code is mechanical and a lot of recursive cases replacing generic class names with the new instantiated class:

Listing 10.9: name_mangle_generics.ml

```
let name_mangle_generic_class class_name type_param =
↪ Class_name.of_string    (Fmt.str "_%s%s" (Class_name.
↪ to_string class_name) (string_of_type type_param))
```

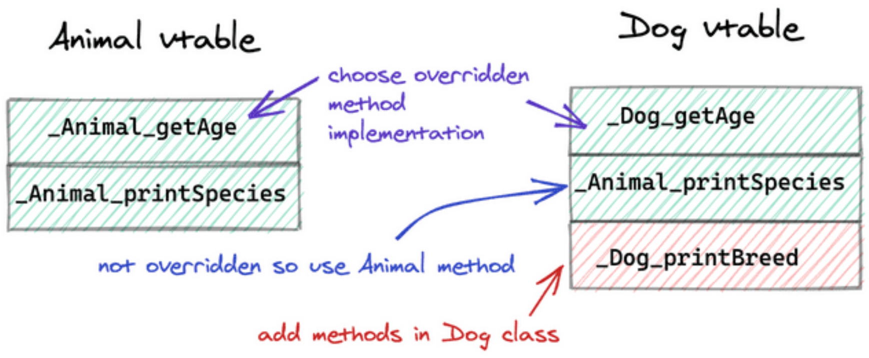
10.4 SUMMARY

That's it! We only had to modify our type-checker and desugaring stage to handle generics, and most of the code was just going through each sub-expression recursively.

This approach of replacing a generic class with specialised instances (one for each concrete type) is called **monomorphism** and it is what C++ does with its templates. If you want to find out more about how other languages implement generics, check out this blog post for a more technical read.

11 ADDING INHERITANCE AND METHOD OVERRIDING TO OUR LANGUAGE

January 25, 2021



Welcome back to part 11 of the series! We've got concurrency and generics in our language, but we're still missing a key component: **inheritance**. Remember, the four main OOP principles are

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

Ah yes, there's polymorphism to tackle too. We've seen *ad-hoc* polymorphism, when we implemented method overloading in the desugaring stage. We've seen *parameteric* polymorphism, when we implemented generics last time round. In this post we'll cover the third type of polymorphism: subtype polymorphism – **method overriding**.

By the end of the tutorial, we'll be able to compile the following program. We'll refer to this example to *motivate* the changes needed to the compiler, so that not only will you understand how the changes work, but you'll understand *why* they are implemented in that way.

And hopefully, you'll be able to see the general principles we've used to add generics in the last post, and inheritance and method overriding in this post, so you'll be able to add even more language features going forward!

```
class Breed {...}
class Species { ... }
class Animal {
   int age;
   Species species;
   int getAge() {    return this.age   }
   void printSpecies(){ ... }
}

class Dog extends Animal {
    Breed breed;
    int getAge() {    return 7*this.age // dog years!  }
    void printBreed(){ ... }
}

function void printAge(Animal a){
   printf("I'm %d years old!", a.getAge());
}
void main() {
  let animal = new Animal(age: 2);
  let dog = new Dog(age: 2);
  printAge(animal) // print 2
  printAge(dog) // print 14
}
```

11.1 JUST GIVE ME THE CODE!

As with the rest of the series, all the code can be found in the Bolt repository.

If you want to see the specific commits that were needed to implement inheritance, check out this pull request and this pull request.

11.2 AST DEFINITIONS

We need to store this inheritance relationship in our Abstract Syntax Tree for our compiler to access. The easiest way is to store the name of the superclass in the class definition, since we're reading it in when we parse `CLASS Someclass EXTENDS Otherclass`:

Listing 11.1: parsed_ast.mli

```
type class_defn = TClass of      Class_name.t       *
    ↪ generic_type option      * Class_name.t option (*
    ↪ optional superclass *)      * capability list      *
    ↪ field_defn list      * method_defn list
```

11.3 TYPE-CHECKER

When we're adding a new language feature, we need to determine what effect it has on the existing typing rules. With inheritance and method overriding we have the following new rules:

- Subclasses like `Dog` have access to not only their own fields and methods, but also their superclass `Animal`'s fields and methods, (and the fields and methods of their superclass' superclass, and so on up the inheritance hierarchy).
- Overridden methods (`getAge`) need to have the same type signature as their superclass (their return types need to match), since they're being used in the same contexts.
- If a superclass is generic, then the subclass must also be generic – otherwise how could it access a field of generic type `T` in the superclass?
- Subclasses like `Dog` are **subtypes** of their superclass (`Animal`). We have some new typing rules to handle when you can use `Dog` in place of `Animal`.

11.3.1 *Accessing superclass' methods*

We just need to update the `get_class_methods`, `get_class_fields`, methods etc. to recursively look up the method/field first in the current class, then its superclass and so on.

Here's a simple example: each class has a list of capabilities. With inheritance, we do a recursive check to get the superclass' capabilities too. The other `get_class_` methods are similar:

Listing 11.2: type_env.ml

```
let rec get_class_capabilities class_name class_defns =  let
    ↪  open Result in  get_class_defn class_name
    ↪ class_defns Lexing.dummy_pos  >>= fun (Parsed_ast.
    ↪ TClass (_, _, maybe_superclass, capabilities, _, _))
    ↪ ->  ( match maybe_superclass with  | Some superclass
    ↪ -> get_class_capabilities superclass class_defns  |
    ↪ None              -> Ok [] )  >>| fun superclass_caps
    ↪ -> List.concat [superclass_caps; capabilities]
```

11.3.2 *Generics and Inheritance*

We pattern-match - if the current class isn't generic, and the superclass is generic, raise a type error!

Listing 11.3: type_inheritance.ml

```
let type_generics_inheritance class_name
    ↪ curr_class_maybe_generic    (Parsed_ast.TClass (
    ↪ superclass_name, superclass_maybe_generic, _, _, _, _
    ↪ )) =  match (curr_class_maybe_generic,
    ↪ superclass_maybe_generic) with  | None, None | Some
    ↪ Generic, None | Some Generic, Some Generic -> Ok ()
    ↪ | None, Some Generic ->      Error         (Error.
    ↪ of_string             (Fmt.str                "Type error
    ↪ : class %s must be generic since superclass %s is
    ↪ generic@."                (Class_name.to_string
    ↪ class_name)                (Class_name.to_string
    ↪ superclass_name)))
```

11.3.3 *Method overriding*

Remember, a method *overrides* another if it has the same name and parameter types (as `getAge` does in our example). To check the overriding methods have the right type, we look up the method type signatures of the current class and the superclass' methods. We raise an error if the current class has a method that overrides an inherited method (same method name and parameter types) but differs in return type.

Listing 11.4: type_inheritance.ml

```
let type_method_overriding class_name class_defns
    ↪ method_defns superclass_defn =  let open Result in
    ↪ get_methods_type_sigs method_defns  |> fun
    ↪ methods_type_sigs ->  get_methods_type_sigs  (
    ↪ get_class_methods class_defns superclass_defn None
    ↪ Lexing.dummy_pos)  |> fun inherited_methods_type_sigs
    ↪  ->  List.filter    ~f:(fun (meth_name, ret_type,
    ↪ param_types) ->      List.exists        ~f:(fun (
    ↪ inherited_meth_name, inherited_ret_type,
    ↪ inherited_param_types) ->          meth_name =
    ↪ inherited_meth_name          && param_types =
    ↪ inherited_param_types          && not (ret_type =
    ↪ inherited_ret_type))
    ↪ inherited_methods_type_sigs)    methods_type_sigs  |>
    ↪  function   (* we find any methods that violate
    ↪ overriding types *)  | []                      -> Ok
    ↪ ()  (* no violating methods *)  | (meth_name, _, _)
    ↪ :: _ ->      Error ...
```

11.3.4 *Subtyping*

We say a type A *subtypes* type B, if we can use A **in place of** B. That occurs if they're equal (obvious) or if A is a subclass of B e.g. Dog in place of Animal. Equivalently, we can say B is the *supertype* of A.

type_inheritance.ml

Copy

Listing 11.5: type_inheritance.ml

```
let is_subtype_of class_defns type_1 type_2 =  type_1 =
    ↪ type_2  ||  match (type_1, type_2) with  | TEClass (
    ↪ class_1, type_param_1), TEClass (class_2,
    ↪ type_param_2) ->      type_param_1 = type_param_2 &&
    ↪ is_subclass_of class_defns class_1 class_2  | _ ->
    ↪ false
```

Intuitively the subtype, A, has **more info** than B: the Dog class has all the Animal behaviour and then some more behaviour specific to dogs.

When do we use subtyping?

SUBTYPING IN VARIABLE ASSIGNMENTS We can assign subtypes to variables e.g. let x: Animal = new Dog(). So in our let expression typing judgement, we ensure the bound expr is a subtype of the type annotation:

Listing 11.6: type_inheritance.ml

```
| Parsed_ast.Let (loc, maybe_type_annot, var_name,
    ↪ bound_expr) ->      ...      type_with_defns
    ↪ bound_expr env      >>= fun (typed_bound_expr,
    ↪ bound_expr_type) ->        ( match maybe_type_annot
    ↪ with      | Some type_annot ->           if
    ↪ is_subtype_of class_defns bound_expr_type type_annot
    ↪ then Ok type_annot      ...
```

SUBTYPING IN FUNCTIONS Subtyping rules for functions are a little more complicated, so we'll use our intuitive notion of subtypes having more info to help us out here.

We can pass subtypes as arguments, as if the function `printAge` was expecting an `Animal` and we give it a `Dog`, it can ignore the extra info like `breed`. Here's the snippet of code that does the check:

Listing 11.7: type_overloading.ml

```
...if are_subtypes_of class_defns args_types param_types
   ↪ then         Ok (param_types, return_type)...
```

Hold on, you ask, why is this subtype check in the `type_overloading` file?

Well, as you add language features, they start to interact with each other. Earlier in the series, we added function overloading: i.e. multiple functions with the same name but different parameter types. We choose the correct overloaded function based on the argument types: we pick the function whose parameter types *match*. Before, our definition of *match* was that the types were equal. Now, we say they *match* if the argument types are **subtypes** of the param types. All the details are in the code.

Likewise, when type-checking our function return type, the body type *matches* the function return type, if it is a subtype. (If the function returns `void` we don't care about the body type.)

Listing 11.8: type_functions.ml

```
...>>= fun (typed_body_expr, body_return_type) -> if
   ↪ return_type = TEVoid || is_subtype_of class_defns
   ↪ body_return_type return_type then    Ok ...else Error
   ↪  ...
```

An identical check is done to type-check methods.

11.4 LOWERING TO LLVM

Now we've done the type-checking, we need to output LLVM IR in order to run the program. Let's quickly remind ourselves of the program we're trying to compile:

Copy

```
class Animal {
  int age;
  Species species;
 int getAge() {    return this.age  }
   void printSpecies(){ ... }
}

class Dog extends Animal {
  Breed breed;
  int getAge() {    return 7*this.age // dog years!  }
  void printBreed(){ ... }
}

function void printAge(Animal a){
   printf("I'm %d years old!", a.getAge());
}
void main() {
  let animal = new Animal(age: 2);
  let dog = new Dog(age: 2);
  printAge(animal) // print 2
  printAge(dog) // print 14
}
```

11.4.1 *Inheritance and Structs*

Both the Animaland Dog classes are desugared to structs containing their fields. This desugaring is straightforward for Animal:
Copy

```
struct Animal {  int age;  Species *species;}
```

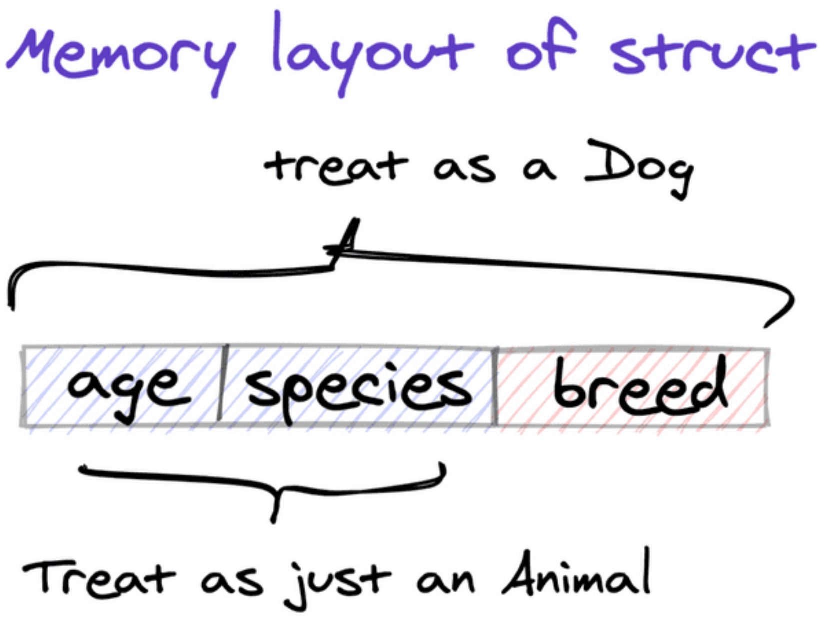
Accessing the age field is desugared into getting a pointer to field 0 of the struct, and likewise species is field 1.

On to the struct for Dog. We have three requirements:

- The struct needs to contain the fields in the Dog class.
- It also needs to contain the fields in the Animal class.
- A Dog struct should be able to be used wherever an Animal struct is expected.

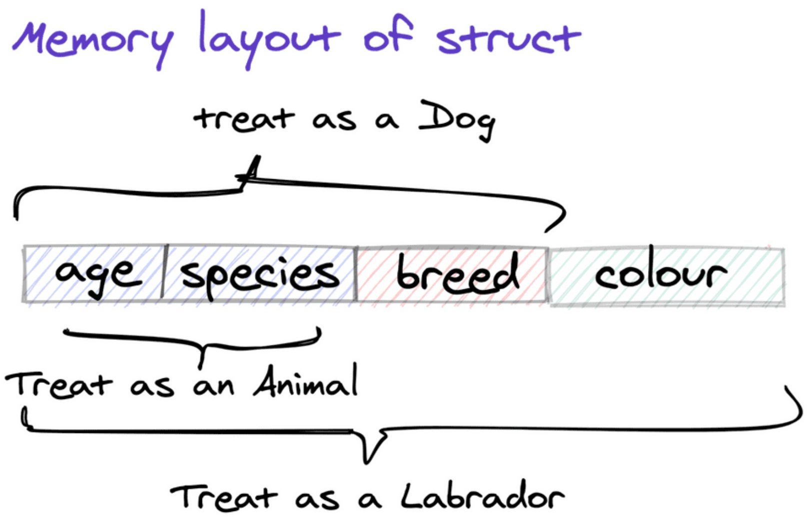
The last point is the critical one. Say I have a dog object being treated as an Animal. When I query dog.age and dog.species, I'll expect them to be in field 0 and 1 of the struct, since it's of type Animal. So the Dog struct needs to preserve that field indexing. So the only place the field breed can go is at index 2.
Copy

```
struct Dog {  int age;  Species *species;  Breed *breed;}
```



In general, we order the struct so all the superclass's fields go first, *then* any fields declared in the current class. If we added a subclass Labrador:

```
class Labrador extends Dog{  Colour colour;}
// Desugared
struct Labrador {
  int age;
  Species *species;
  Breed *breed;
  Colour *colour;
}
```



If we want to treat a Labrador as an Animal, only look at the first 2 fields. If you want to treat it as a Dog look at the first 3 fields. And so on.

You can see this ordering in our get_class_fields method during our IR generation stage, which puts superclass_fields first.

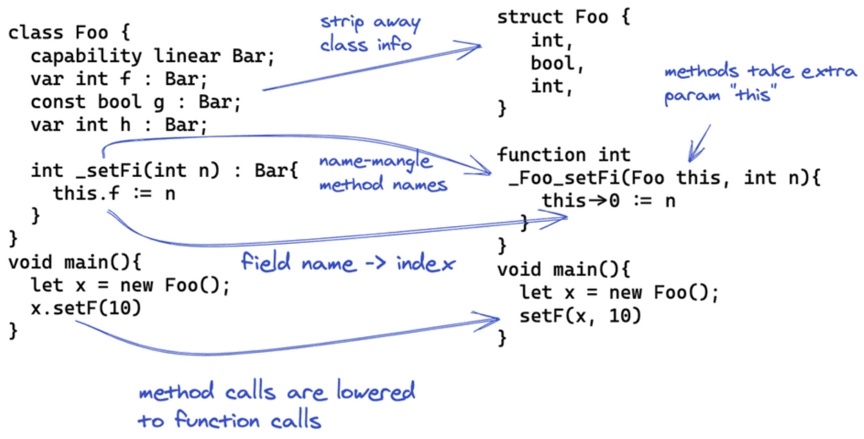
Listing 11.9: ir_gen_env.ml

```
let rec get_class_fields class_name class_defns =
    ↪ get_class_defn class_name class_defns  |> fun (TClass
    ↪  (_, maybe_superclass, _, fields, _)) ->  ( match
    ↪ maybe_superclass with  | Some super_class ->
    ↪ get_class_fields super_class class_defns  | None
    ↪            -> [] )  |> fun superclass_fields -> List
    ↪ .concat [superclass_fields; fields]
```

Since we've handled this in the Bolt IR gen stage of the compiler frontend, we don't need to change our LLVM backend, right?

Almost right. Just one hitch: LLVM IR is **typed**, so it will complain if we pass a Dog * pointer to a function that expects a Animal * pointer. Remember, LLVM has no notion of inheritance, just raw structs, so it can't see that Dog is a subclass of Animal. We thus *explicitly cast* the argument pointer to the function's expected param type before we pass it to the function:

Listing 11.10: expr_codegen.cc



```
class Foo {
  capability linear Bar;
  var int f : Bar;
  const bool g : Bar;
  var int h : Bar;

  int _setFi(int n) : Bar{
    this.f := n
  }
}
void main(){
  let x = new Foo();
  x.setF(10)
}
```

```
struct Foo {
   int,
   bool,
   int,
}

function int
 _Foo_setFi(Foo this, int n){
   this->0 := n
 }
}
void main(){
  let x = new Foo();
  setF(x, 10)
}
```

strip away class info
methods take extra param "this"
name-mangle method names
field name -> index
method calls are lowered to function calls

Figure 3: The diagram from our previous desugaring post.

```
std::vector<Value *> argVals;
for (int i = 0; i < expr.arguments.size(); i++) {
    Value *argVal = expr.arguments[i]->codegen(*this);
    Type *paramTy = calleeFunTy->getParamType(i);
    Value *bitCastArgVal = builder->CreateBitCast(argVal,
        ↪ paramTy);
    argVals.push_back(bitCastArgVal);
}
```

11.4.2 *Method Overriding and Virtual Tables*

Method overriding is a bit tricky, so take a breather. From our previous post, we have that methods are desugared to regular functions:

In our running example in this post we have getAge overridden, so we have these two functions corresponding to the implementation of getAge in each of the classes:

```
Animal_getAge(Animal this){  return this.age}
Dog_getAge(Dog this){  return 7 * this.age}
```

When we call a.age() in our printAge function, we want to call Animal_getAge if the underlying object is an Animal, and Dog_getAge if the object is actually a Dog. The thing is, in general we don't know this information at compile-time. Here's an example where type of dog is only known at runtime:

```
let dog = new Animal()
if (someCondition) {
  dog = new Dog()
}
dog.getAge() // which function do we call?
```

So how do we insert the right call? The problem is not too dissimilar to this:

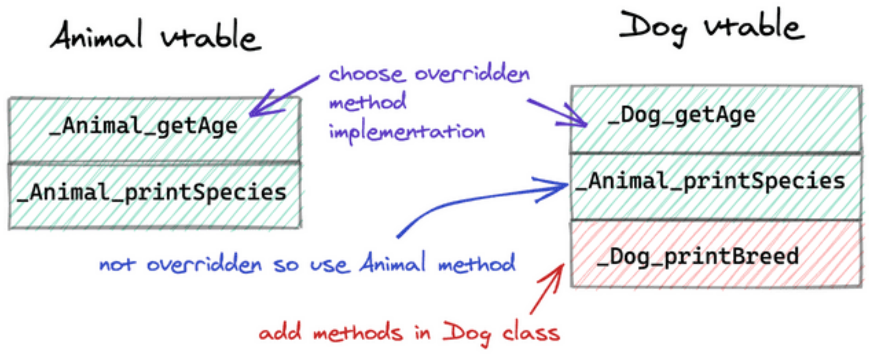
```
let dog = new Animal()
```

```
dog.age = 7
if (someCondition) {
  dog.age = 14
}
dog.age // which value does age have?
```

We know how to do this: look up the value by indexing into the *list of fields* in the dog struct at runtime (index `0` for age).

Since we've solved the problem for fields, let's do the same thing for methods. We can create a *list of (pointers to) methods* for each class, called the **virtual table** or **vtable**. Instead of determining the function call at compile-time, we instead provide an *index* into the table and call the function pointed to at that index at runtime.

For similar reasons to the struct fields before, superclasses' methods come before the current class's methods in the vtable. So we have `getAge`, `printSpecies` first as they're from `Animal`, then we have `getBreed` from `Dog`. To override a method, we just replace its entry in the table:



Now we can just say for `dog.getAge()`, look at index `0` of the dog's vtable, and execute whichever function's there. Or index `1` for `dog.printSpecies()`. **Problem solved!**

VIRTUAL TABLES IN OUR STRUCTS Unlike fields, which are different for each object, vtables are the same for *all* objects of a *given class*. Having a copy of the same vtable in each object is wasteful: instead have just **one** global vtable **per class**, and have the objects store a *pointer* to that.

We've reserved the first field in our object struct for the vtable pointer:

```
struct Animal {
  AnimalVTable *vtableptr;
  int age;
  Species *species;
}
struct Dog {
  DogVTable *vtableptr;
  int age;
  Species *species;
  Breed *breed;
}
```

Okay, let's implement it with some code! To generate the vtable, we get a list of the class' methods, **annotated** with the class they came from (so we get the right overridden method). That is, we don't have `getAge` we have the pair `(Dog, getAge)`. We use this pair to generate the name-mangled function name: `_Dog_getAge`. Our vtable is just a list of these name-mangled function names.

Listing 11.11: ir_gen_env.ml

```ocaml
let ir_gen_vtable class_name class_defns = get_class_annotated_methods class_name class_defns |> fun class_annotated_methods -> List.map  ~f:(fun (class_annot, meth_name) -> name_mangle_method_name meth_name class_annot) class_annotated_methods
```

LLVM IMPLEMENTATION OF VIRTUAL TABLES We implement VTables as a struct of function pointers. Below are the type definitions, and the corresponding global variable declaration.

Listing 11.12: vtable.ll

```llvm
%_VtableAnimal = type { i32 (%Animal*)*, void (%Animal*)* }
%_VtableDog = type { i32 (%Dog*)*, void (%Animal*)*, void (%Dog*)*}
@_VtableAnimal = global %_VtableAnimal { i32 (%Animal*)* @_Animal__getAge, void (%Animal*)* @_Animal__printSpecies }
@_VtableDog = global %_VtableDog { i32 (%Dog*)* @_Dog__getAge, void (%Animal*)* @_Animal__printSpecies, void (%Dog*)* @_Dog__printBreed }
%Animal = type { %_VtableAnimal*, i8*, i32, i32, i32, %Species* }
%Dog = type { %_VtableDog*, %Species*, %Breed* }
```

Creating the global vtables follows the format we mentioned for global variables in the LLVM post. We create the table as a `ConstantStruct`, and populate the (`vector<Constant *>`) body with `Function *` pointers to each of the methods.

Listing 11.13: class_codegen.cc

```cpp
void IRCodegenVisitor::codegenVTables(const std::vector<std::unique_ptr<ClassIR>> &classes) {
  for (auto &currClass : classes) {
    std::string vTableName = "_Vtable" + currClass->className;
    StructType *vTableTy = module->getTypeByName(StringRef(vTableName));
    std::vector<Constant *> vTableMethods;
    std::vector<Type *> vTableMethodTys;
    for (auto &methodName : currClass->vtable) {
      Function *method = module->getFunction(StringRef(methodName));
      vTableMethods.push_back(method);
      vTableMethodTys.push_back(method->getType());
    }
    vTableTy->setBody(vTableMethodTys);
    module->getOrInsertGlobal(vTableName, vTableTy);

    GlobalVariable *vTable = module->getNamedGlobal(
        ↪ vTableName);
    vTable->setInitializer(      ConstantStruct::get(
        ↪ vTableTy, vTableMethods));
  }
}
```

Then to call the function, we're replacing our static function call:

```
*calleeMethod =      module->getFunction(expr.methodName);
```

with a vtable lookup. This is just two Struct GEP lookups: first we get the vtable pointer by reading index `0` of the object, then we get the method pointer by using the `methodIndex` into the vtable. (Again, check the LLVM post for a refresher on LLVM).

Listing 11.14: expr_codegen.cc

```
Value *vTablePtr = builder->CreateLoad(builder->
    ↪ CreateStructGEP(
    thisObj->getType()
        ->getPointerElementType() /* get type of element on
            ↪ heap*/,
    thisObj, 0));Value *calleeMethodPtr =
    builder->CreateStructGEP(vTablePtr->getType()->
        ↪ getPointerElementType(),
                                vTablePtr, expr.methodIndex);
Value *calleeMethod = builder->CreateLoad(calleeMethodPtr);
```

An minor detail, why is `calleeMethod` now of type `Value *` not `Function *`? I asked the LLVM dev mailing list this question - it's because `Function *` refers to a function known at compile-time. Our vtable functions are looked up at runtime, so aren't of type `Function *`.

11.5 SUMMARY

This post on inheritance and method overriding brings to a close the Bolt compiler series for now. These tutorials cover the general language features of the Bolt language at the time I submitted my dissertation.

There's always more language features to add, perhaps later down the line I'll write another tutorial on arrays! In the meantime, I've got a host of new posts on other content coming this year - 40 new posts coming in 2021!