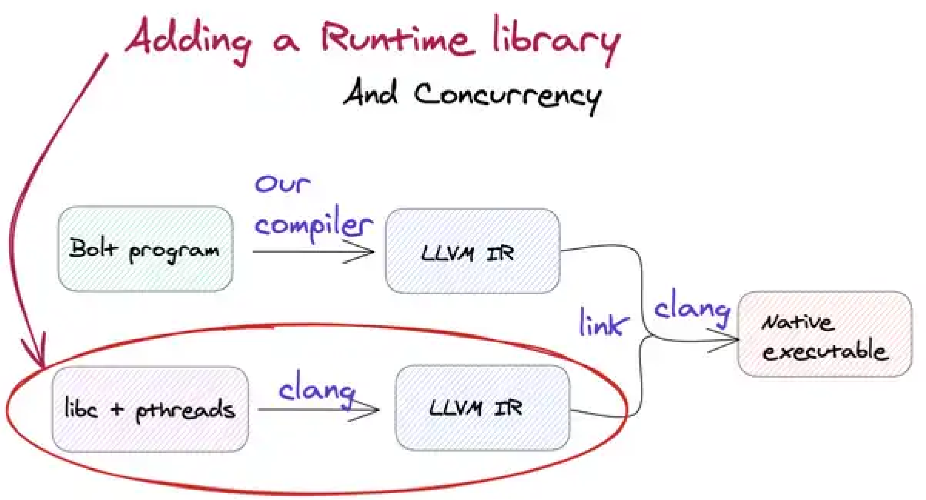
CREATING THE BOLT COMPILER: PART 9

Implementing Concurrency and our Runtime Library

DECEMBER 28, 2020 9 MIN READ



SERIES: CREATING THE BOLT COMPILER

- Part 1: How I wrote my own "proper" programming language
- Part 2: So how do you structure a compiler project?
- Part 3: Writing a Lexer and Parser using OCamllex and Menhir
- Part 4: An accessible introduction to type theory and implementing a type-checker
- Part 5: A tutorial on liveness and alias dataflow analysis
- Part 6: Desugaring - taking our high-level language and simplifying it!
- Part 7: A Protobuf tutorial for OCaml and C++
- Part 8: A Complete Guide to LLVM for Programming Language Creators
- **Part 9: Implementing Concurrency and our Runtime Library**

TABLE OF CONTENTS

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?

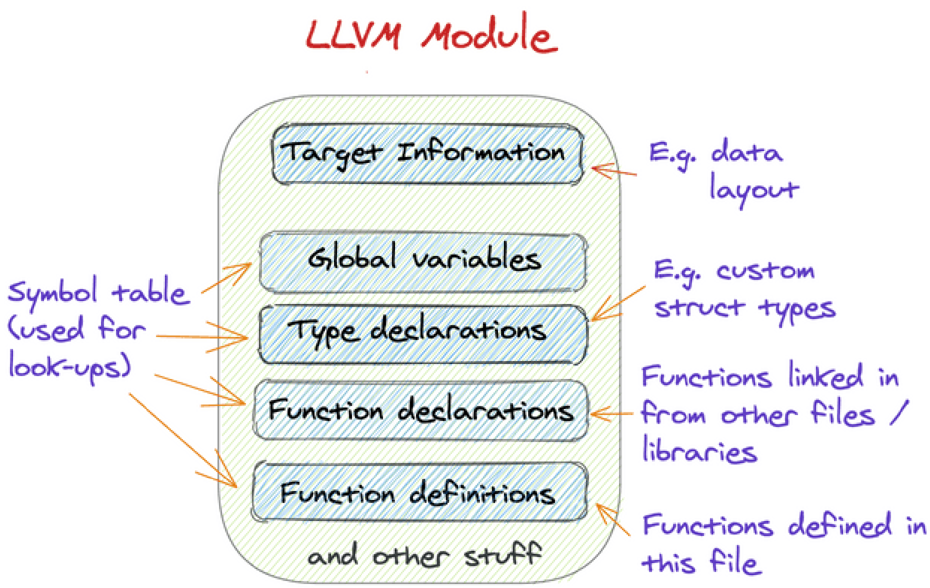
- I/O e.g. `printf`
- Memory management. (Remember in the previous post I mentioned that the LLVM didn't provide a heap.) Either implemented manually (`malloc` and `free`) or through a garbage collection algorithm (e.g. *mark-and-sweep*).
- And of course **threads** via the C `pthread` API. Pthread is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

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.



Printf

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

Copy

```
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:

extern_functions_codegen.cc

Copy

```
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:

expr_codegen.cc

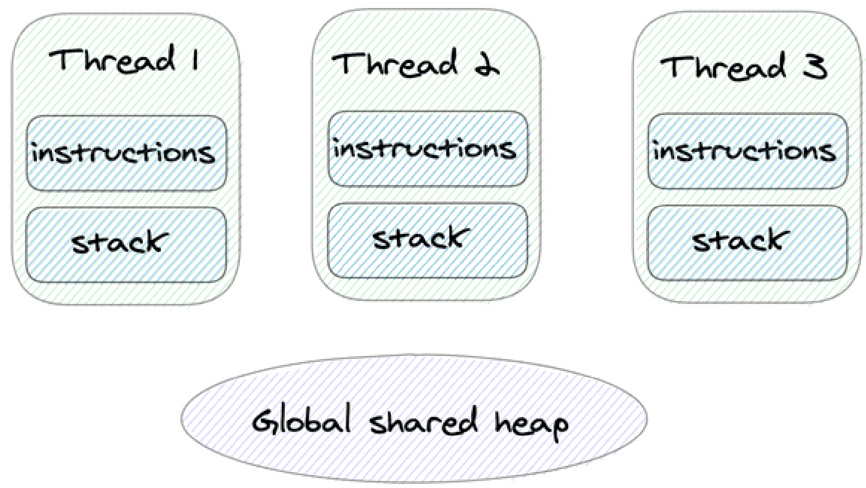
Copy

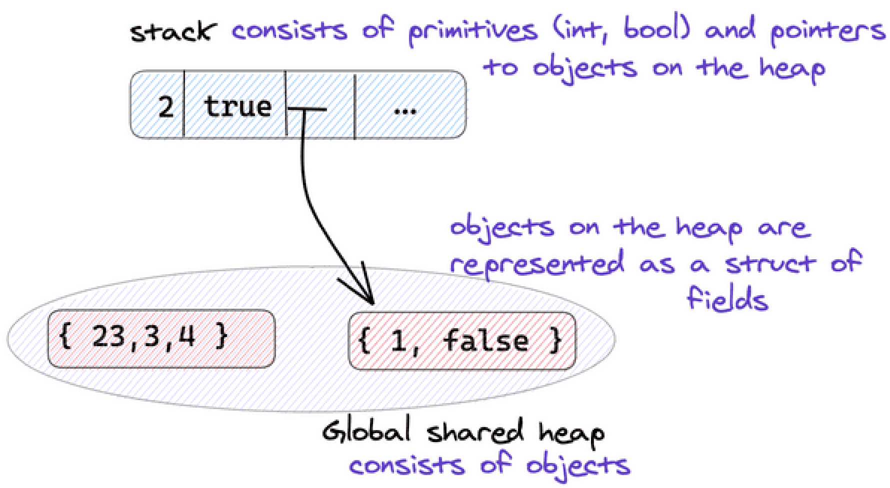
```
Value *IRCodegenVisitor::codegen(const ExprPrintfIR &expr) {
  Function *printf = module->getFunction("printf");
  std::vector<Value *> printfArgs;
  Value *formatStrVal = builder->CreateGlobalStringPtr(expr.for
  printfArgs.push_back(formatStrVal);
  // add variadic arguments
  for (auto &arg :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

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.





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:

Copy

```
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.

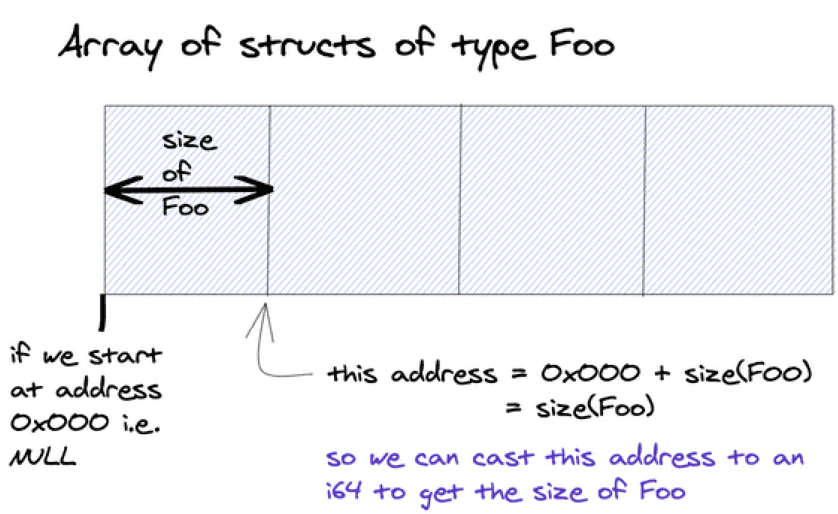
extern_functions_codegen.cc

Copy

```
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)` .



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.

expr_codegen.cc

Copy

```
// calculate index of array[1] using GEP instruction
Value *objDummyPtr = builder->CreateConstGEP1_64(
    Constant::getNullValue(objType->getPointerTo()), 1, "objsiz
// cast to i64 for malloc
Value *objSize =
    builder->CreatePointerCast(objDummyPtr,Type::getInt64Ty(*cc
```

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!

expr_codegen.cc

Copy

```
// allocate the object on the heap
Value *objVoidPtr =
    builder->CreateCall(module->getFunction("malloc"), objSize)

// cast (void *)  to  (objType *)
Value *obj =
    builder->CreatePointerCast(objVoidPtr, objType->getPointerT
```

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.

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.

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:

Copy

```
int pthread_create(pthread_t *thread, const pthread_attr_t *att
                          void *(*start_routine) (void *), void

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.

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:

Copy

```
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:

Copy

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

```
// LLVM IR
%4 = alloca %struct._opaque_pthread_t*, align 8
...
%9 = call i32 @pthread_create(%struct._opaque_pthread_t** %4, %
...
%23 = call i32 @pthread_join(%struct._opaque_pthread_t* %22, i8
```

If we match up our type definitions for the functions:

Copy

```
// C type -> LLVM IR type
pthread_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:

Copy

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

Copy

```
Type *pthread_t = StructType::create(*context, "struct_pthread_

Type *pthread_attr_t = StructType::create(*context,"struct_pthr
```

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:

Copy

```
// 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.

expr_codegen.cc

Copy

```
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.

[extern_functions_codegen.cc]

Copy

```
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:

[extern_functions_codegen.cc]

Copy

```
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
//                   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);
```

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/
```

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

Implementing Concurrency in Bolt

The Finish-Async Construct

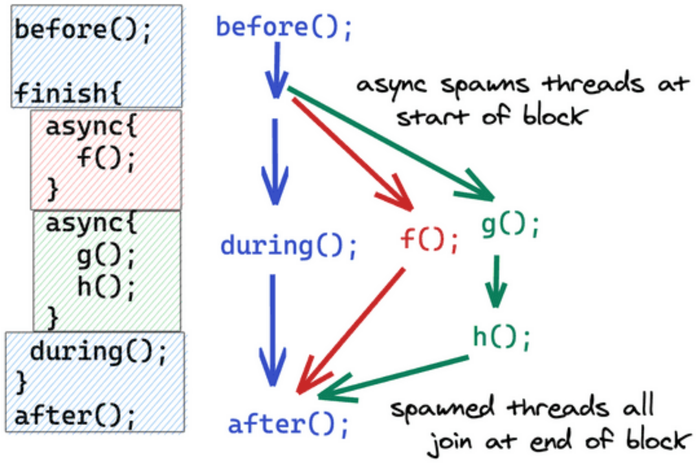
Bolt uses the `finish-async` construct for concurrency. The `async` keyword spawns (creates) a thread and sets it to execute the expressions in that `async` block. The `finish` block scopes the lifetime of any threads spawned inside it using the `async` keyword - so all spawned threads must **join**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

example_concurrent_program.bolt

Copy

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

Illustrating the execution graphically:



A Bolt concurrent program illustrated - each colour represents a different thread.

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{
    // 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.

Copy

```
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.

Copy

```
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:

Copy

```
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(argStruct);
```

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

We need to unpack this inside the function:

Copy

```
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:

[pthread_codegen.cc]

Copy

```
// create async function and argument
  StructType *argStructTy = codegenAsyncFunArgStructType(freeVa
  Value *argStruct = codegenAsyncFunArgStruct(asyncExpr, argStr
  Function *asyncFun = codegenAsyncFunction(asyncExpr, argStruc
  ...
  // 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.

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:

pthread_codegen.cc

Copy

```
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(voidPt
  }
```

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:

expr_codegen.cc

Copy

```
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("pthrea
    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;
```

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!

SERIES: CREATING THE BOLT COMPILER

← A Complete Guide to LLVM for Programming Language Creators

Write It and They Will (Eventually) Come →