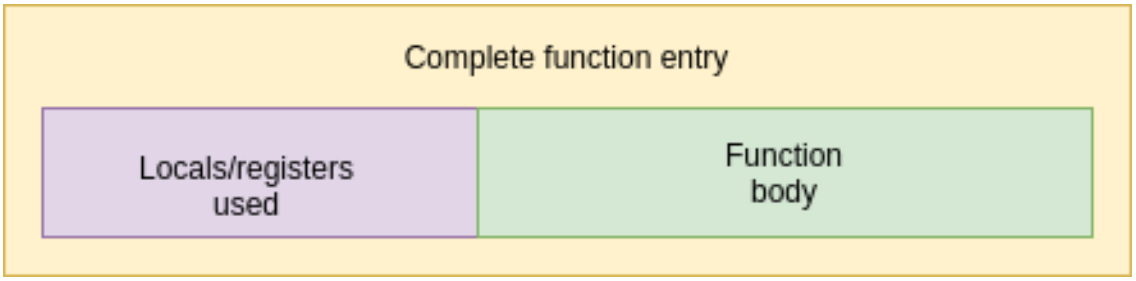
WASM -> MIR Proposal

Codegen

Registers/Local allocation

Locals are stored in the prologue of a function, the actual function body comes next:



Problem: We do not know how many locals/registers are used until the end of a function codegen. This means we need to insert them at a specific position in the code, making offsets unstable and means we use more bytes than needed (this is a side-effect due to fixed size indexes instead of LEB128 efficiency).

Control flow

The AIR of a simple if-else statement looks as follow:

```
%11!= block(void, {
        block
   %12 = load(u32, %1)
   %16 = cmp_eq(%12!, %15!)
   %27!= cond_br(%16!, {
     %24!
     %17!= dbg_stmt(5:9)
     %20!= store(%1, %19!)
     %21!= br(%11, @Ref.void_value)
   }, {
     %19!
     %22!= dbg_stmt(7:9)
     %25!= store(%1, %24!)
     %26!= br(%11, @Ref.void_value)
   })
 })
```

This means, we compare %12 and %15, and then emit a conditional branch.
Wasm only supports backwards jumping within blocks. This means we must insert an extra block, before the `cond_br` instruction. Right now, we save the offset of the initial `load` instruction and during the `cond_br` we insert a block at that offset. While this works, there's been times I ran into issues with this approach due to shifting of opcodes.

Linking

Relocations

Relocations for wasm contain 3 fields (and an optional field):

- Type (Tells if we are relocating a global, a function index, and how to encode this value (u32 4 byte or LEB128-fixed 5 bytes).
- Symbol index (The target symbol that is being relocated) i.e. for function calls, this represents the symbol that is being called.
- Offset: The offset within a section entry where the value lives that needs to be relocated. i.e. write the above symbol's value at offset.
- Addend (**optional**): Bytes added to an offset

Problem:
As relocations contain the offset into the generated machine/wasm code, the offsets must be stable. If not, we must recalculate the offsets each time we insert an opcode in the middle of our code, rather than appending it at the end. Boooo, inefficient, adds complexity and is prone to nasty bugs in both the linker and codegen.

Misc

Inline-assembly

Thanks to LLVM we already support inline-assembly for wasm. This looks as follow:

```
pub fn main() u8 {
    return asm volatile (
        \\i32.const 57
        \\local.set %[ret]
        : [ret] "=r" (-> u8),
    );
}
```

- load constant 57 onto the stack.
- Store stack value in %[ret]
- Return what's stored in [ret]

This would be easier to support thanks to MIR

Optimization

With MIR we could implement one or more easy optimization passes at a later point.
For example:

```
;this adds local 0 and local 1: 5 + 10
i32.const 5
local.set 0
i32.const 10
local.set 1
local.get 0
local.get 1
i32.add

;This could be optimized to:
i32.const 5
local.tee 0
i32.const 10
local.tee 1
i32.add

;local.tee sets a local without popping the stack value
```

Possible solutions

Register/local allocation pass

Insert a pass over the AIR that allocates locals/registers for the function. The idea is to loop over all AIR instructions, find which require a local, and allocate those in a map where the key is the instruction and the value is the index. Once finished, we can do the actual codegen of the function, and rather than allocate them again, we find the corresponding local/register using the instruction index.

+ This solves any problems we have with patching offsets and creates a stable list of wasm opcodes during codegen.
+ Very quick to implement with a massive improvement to usability.

- Does not necessarily allow for optimisation passes
- Does not create a way to allow for inline-assembly
- Does not provide an easier way to lower conditional branches

MIR (or similar) pass

+ Solves patching of offsets as we can allocate any locals/registers we require during the initial AIR pass and pass the local's index into the MIR instruction.
+ Allows for optimisation passes specific to WASM.
+ Allows us to easily implement inline-assembly
+ Allows us to re-order instructions when lowering to MIR to insert blocks before conditional branches.
+ Makes both AIR and MIR passes easier to maintain as both will have smaller code during each instruction as they have less reponsibility.

- A lot more effort to implement
- Additional runtime/memory cost for creating MIR instructions and emitting those.
- MIR is not much lower level than AIR so doesn't add big improvements to non-control-flow instructions as we can easily lower AIR to wasm already for those.