Elm first-class functions in Wasm

I’ve been a bit fascinated for the past few months with trying to understand how a future version of the Elm compiler might be able to target WebAssembly. What are the major differences from generating JavaScript? What are the hard parts, what approaches would make sense?

I think one of the most interesting questions is: how do you implement first-class functions in WebAssembly? JavaScript has them built in, but WebAssembly doesn’t. Treating functions as values is a pretty high level of abstraction, and WebAssembly is a very low-level language.

# Contents

[Elm and WebAssembly](#_mc5ire1nsq9d)

[Elm’s first-class functions](#_tgklt94qefz0)

[Key WebAssembly concepts](#_84ible818h8w)

[Representing closures as bytes](#_z74gw1k8x13s)

[Function application](#_i0n1snxa42ol)

[Lexical closure](#_p3yl1tiqa7wi)

[Code generation](#_bf7qqayjjhai)

[Summary](#_jyiy65ycqmva)

[What’s next?](#_3wumb8j74qia)

[Links](#_q8brvgvolhmm)

# Elm and WebAssembly

Before we get started, I just want to note that from what I’ve heard from the core team, there is currently no concrete plan for building WebAssembly into the Elm compiler. WebAssembly is still an MVP and won’t really be ready for Elm until it has Garbage Collection, and probably also access to the DOM and other Web APIs. That doesn’t look likely to happen in 2018.

But... it will get past MVP at some point, and this stuff is kind of fascinating, so let’s have a think about what it could look like!

So… how do you go about implementing first-class functions in a low-level language like WebAssembly? It’s a pretty high-level language feature, and WebAssembly is all just low-level machine instructions. Machine instructions aren’t something you can “pass around”! And what about partial function application? And isn’t there something about “closing over” values from outside the function scope?

Let’s break this down.

# Elm’s first-class functions

I think the best way to start is to look at some example Elm code and list the features of Elm functions that we’ll need to implement in WebAssembly.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | module ElmFunctionsDemo exposing (..)  outerFunc : Int -> (Int -> Int -> Int)  outerFunc closedOver =  let  innerFunc arg1 arg2 =  closedOver + arg1 + arg2  in  innerFunc  myClosure : Int -> Int -> Int  myClosure =  outerFunc 1  curried : Int -> Int  curried =  myClosure 2  higherOrder : (Int -> Int) -> Int -> Int  higherOrder function value =  function value  answer : Int  answer =  higherOrder curried 3 |

Running this code in the Elm REPL, we get the answer of 1+2+3=6

|  |
| --- |
| $ elm repl  ---- elm-repl 0.18.0 -----------------------------------------------------------  :help for help, :exit to exit, more at <https://github.com/elm-lang/elm-repl>  --------------------------------------------------------------------------------  > import ElmFunctionsDemo exposing (..)  > answer  6 : Int |

This is definitely not the simplest way to write this calculation! But it does illustrate all the most important features of Elm functions.

### Three key features

Firstly, Elm functions are first-class, meaning they are *values* that can be returned from other functions (like outerFunc) and passed into other functions (like higherOrder).

Secondly, they support *lexical closure*. innerFunc “captures” the value of closedOver, which is defined outside of its body. myClosure “remembers” the value of closedOver that it was created with, which in this case is 1.

Finally, Elm functions support *partial application*. myClosure is a function that takes two arguments, but on line 17 we only apply one argument to it. As a result, we get a new function that is waiting for one more argument before it can actually run. This new function “remembers” the value that was partially applied, as well as the closed-over value.

### Clues in the code

Note that we now have several Elm functions that will all will end up executing the *same line of code* when they actually get executed! That’s line 7. If somebody calls curried with one more argument, line 7 will do the work. Likewise, if somebody calls myClosure with two arguments, line 7 will do the work then too.

This gives us a clue to one of the things we’ll need to implement first-class functions in a language that doesn’t support them directly. We’ll need a WebAssembly function that implements the body expression on line 7. And somehow, all of the function values we’re passing around will need to have a *reference* to that WebAssembly function, so that they can eventually use it for execution.

In WebAssembly, we can’t pass functions around, only data. But maybe we can create a data structure that *represents* an Elm function value, keeping track of the curried arguments and closed-over values. The runtime can operate on that data structure and pass it around. When we finally have all the arguments ready and we’re ready to execute the body, we can call the WebAssembly function to evaluate the body expression and produce a return value. Let’s call the WebAssembly function an “evaluator function”.

There are still lots of details missing at this stage. In order to fill in the gaps, we’re going to need a bit of background knowledge on some of WebAssembly’s language features.

# Key WebAssembly concepts

### Linear memory

WebAssembly modules have access to a block of “linear memory” that they can use to store and load data. It’s a linear array of bytes, indexed by a 32-bit integer. WebAssembly has built-in instructions to store and load integers and floats, but anything more complex has to be built up from raw bytes.

The fact that everything is built up from raw bytes means that WebAssembly can be a compile target for lots of different languages. Different data structures will make sense for different languages, but they’re all just bytes in the end. It’s up to each compiler and runtime to define how those bytes are manipulated.

### Tables

WebAssembly has a feature called “tables” which it uses to implement “indirect calls”. Indirect calls are a feature of almost every high-level language, but what are they?

When a machine executes a function call, it obviously needs some reference to know which function to invoke. In a *direct call*, that function reference is simply hardcoded, so it invokes the same function every time. In an *indirect call*, however, the function reference is provided by a runtime value instead. This is a very handy thing to be able to do, because it means the caller doesn’t need to know in advance the full list of functions it might have to call. Because of this, most languages have some version of this. C and C++ have function pointers, Java has class-based polymorphism, and Elm has first-class functions.

A WebAssembly *table* is an array of functions, each indexed by a 32-bit integer. There’s a special call\_indirect instruction that takes the index of the function to be called, with a list of arguments, and executes it. The program statically declares which functions are *elements* of the table, and call\_indirect only works on those functions. (Incidentally, there’s also a call instruction for direct calls, but we won’t be focusing on that.)

By the way, WebAssembly has this design for safety reasons. If functions were stored in linear memory, it would be possible for code to inspect or corrupt other code, which is not good for web security. But with an indexed function table, that’s impossible. The only instruction that can even access the table is call\_indirect, which is safe.

If you’re interested in some further reading, I recommend Mozilla’s article on [Understanding the Text Format](https://developer.mozilla.org/en-US/docs/WebAssembly/Understanding_the_text_format), and the design document on [WebAssembly Semantics](https://github.com/WebAssembly/design/blob/master/Semantics.md).

But for now, we have enough knowledge to discuss how to implement first-class functions.

# Representing closures as bytes

As mentioned earlier, to represent an Elm function in WebAssembly we’ll need a function and a data structure. We’ll use the term “closure” to refer to the data structure*,* and “evaluator function” to refer to the WebAssembly function that will evaluate the body expression and produce a return value.

One way of representing a closure in binary could be the following, where each box represents an integer (4 bytes).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| fn\_index | arity | mem\_ptr0 | mem\_ptr1 | mem\_ptr2 | ... |

**fn\_index** is an integer index into the function table where the evaluator function for this closure can be found. At runtime, once all of the arguments have been applied to the closure, we can invoke the call\_indirect instruction to look up the table, call the evaluator function, and return a result.

**arity** is the *remaining* number of parameters to be applied to the closure. Every time we apply another argument, we insert a pointer to that argument, and decrement the arity. When the arity reaches zero, we’re ready to call the evaluator function.

**mem\_ptr\*** are pointers representing the addresses in linear memory of the arguments and closed-over values. They all start off “empty” (zero), and are filled in reverse order as arguments are applied. So if the closure has an arity of 2, then mem\_ptr0 and mem\_ptr1 will be “empty”. When we apply the next argument, the mem\_ptr1 will be filled with the address of the argument value, and arity will be decremented from 2 to 1, with mem\_ptr0 still being empty.

# Function application

We’ve already mentioned some of the things that need to happen when a closure is applied to some arguments, but let’s look at it in more detail.

Here’s some pseudo-code to describe the algorithm for function application.

|  |
| --- |
| closure = copy(original\_closure)  for each applied argument  closure.mem\_ptr[closure.arity - 1] = *argument address*  closure.arity--  if closure.arity > 0 then  return closure  else  return call\_indirect(closure.func\_index, closure) |

Let’s go through this from the top.

Before applying the closure, we need to create a new copy of it so, that the old closure is still available for other code to use. All Elm values are immutable, and this is no exception.

Next, we insert the applied arguments into the closure. The closure structure has one pointer “slot” for each argument. We need to make sure we put each argument in the right slot, and there’s a neat little trick we can use here to make this easy. Since the arity has to go down by 1 every time we apply an argument, we can actually use it to tell us which slot is next. All we have to do is fill them in reverse!

For example if the arity is 2, we’ll insert an argument into the closure at mem\_ptr1, and if the arity is 1, we’ll insert the argument into the closure at mem\_ptr0.

Finally, we check the remaining arity after applying all the arguments in this call. If the remaining arity is non-zero, this must be a partial application, and we just return the closure. If it’s zero, that means all arguments have been applied. We need to call the evaluator function, and return the value it gives us.

Note that we are passing the closure data structure *into* the evaluator function as its only argument. The closure structure contains all of the data we need to evaluate the body of our Elm function, because that’s exactly what it was designed for. So it’s the only argument the evaluator function needs.

Inside the evaluator function, we’ll need to do some destructuring to get the individual arguments out of the closure structure. The compiler will have to generate the appropriate code for that.

# Lexical closure

Let’s look again at our example of closing over values from an outer scope.

|  |
| --- |
| outerFunc : Int -> (Int -> Int -> Int)  outerFunc closedOver =  let  innerFunc arg1 arg2 =  closedOver + arg1 + arg2  in  innerFunc |

To help us think about how to generate WebAssembly for innerFunc, let’s first refactor the source code to the equivalent version below.

|  |
| --- |
| outerFunc : Int -> (Int -> Int -> Int)  outerFunc closedOver =  let  -- Replace inner function definition with partial application  innerFunc =  transformedInnerFunc closedOver  in  innerFunc  -- Move definition to top level, inserting a new first argument  transformedInnerFunc closedOver arg1 arg2 =  closedOver + arg1 + arg2 |

Here we’ve moved the definition of the inner function to the top level, and inserted closedOver as a new first argument, instead of actually closing over it. This doesn’t make any difference to anyone who calls outerFunc - it still creates an innerFunc that remembers the value of closedOver it was created with.

The big win here is that we no longer have nested function definitions. Instead, they’re all defined at top level. This is useful because we need to put all of our evaluator functions into one global WebAssembly function table. Remember, the table is WebAssembly’s way of supporting indirect function calls. So we’ll need the compiler to do this transformation on all nested function definitions.

# Code generation

We’re now ready to look at the steps the compiler needs to take to generate code for an Elm function.

1. Generate the body expression, keeping track of all of the *local names* referenced in the body (we can ignore top-level names).
2. From the set of local names, remove the argument names and any names defined ‘let’ subexpressions. Only the closed-over names will remain.
3. Prepend the list of the closed-over names to the list of function arguments, to get the argument list for the evaluator function.
4. Generate the evaluator function
5. Declare the evaluator function as an element of the function table
6. Insert code into the parent scope that does the following
   1. Create a new closure structure in memory
   2. Partially apply the closed-over values from the parent scope

# Summary

One of the interesting challenges in compiling Elm to WebAssembly is how to implement first-class functions.

Elm functions have a lot of advanced features that are not directly available in WebAssembly. They behave like values, they can be partially applied, and they can capture values from outer scopes.

Although WebAssembly doesn’t have these features natively, it does provide the foundations to build them. WebAssembly supports indirect function calls using a function table, allowing us to pass around *references* to WebAssembly functions in the form of a table index.

We can represent an Elm function using a WebAssembly function and a data structure. We saw what the byte level representation of the data structure could look like. The data structure is what gets passed around the program, keeping track of partially-applied arguments and closed-over values. It also contains the table index of the evaluator function, which is what will eventually produce a return value.

We discussed a way to implement lexical closure. It involves automatically transforming Elm code, flattening nested function definitions so that they can be inserted into the WebAssembly function table. This transformation turns lexical closure into partial function application.

Finally we outlined some of the steps the compiler’s code generator needs to take, and looked at the runtime algorithm for function application.

# What’s next?

I’m working on a prototype code generator to prove out these ideas. I’m making reasonable progress, and there don’t appear to be any major blockers, but it needs some more work to get it working. I’ll probably share something more if/when I get that far!

I’ve also got some ideas for more blog posts around this topic. Let me know in the comments if you’d like to see any of these!

* Byte-level representations of the other Elm data structures
  + Extensible records, union types, lists, tuples
  + Numbers, comparables and appendables
* Code generation architecture
  + WebAssembly AST and code gen structure
  + Can we generate Wasm from Haskell? Should we use Rust?
* The Elm runtime in WebAssembly
  + Platform, Scheduler, Task, Process, Effect Managers
* DOM, HTTP, and ports. Differences between Wasm MVP and post-MVP.
* Strings and Unicode
* Tail-Call Elimination with trampolines

Thanks for reading! Leave a comment!