Compiling Linear Sax to WASM

CMU CS 15-817: HOT Compilation
Mikail Khan

Abstract

Sax is an intermediate representation (IR) that helps bridge the gap between high level, richly typed languages and lower level imperative execution. WebAssembly (WASM) is a bytecode instruction set focused on speed, security, and portability. It is an appealing compilation target because it can be run quickly and securely on many platforms, including embedded devices and web browsers, and is used in industry for accelerating web performance or running untrusted code. In this paper, I detail the structure and implementation of a Sax to WASM compiler, supporting purely linear types and 32-bit integers.

1 Introduction

The most interesting aspect of compiling linear Sax to WASM is memory management. WASM is lower-level than Sax and exposes details like stack and local layout, so when starting this project I spent a lot of time and effort trying to optimize these. However, WASM runtimes often include JIT compilers that make precise stack and local layout irrelevant to performance. Additionally, `wasm-opt` is a CLI tool and library that runs many optimization passes, including local allocation. So the main focus of this compiler is on exploiting linear typing's advantages for memory allocation, with an eye on maintaining extensibility for supporting adjoint Sax through WASM GC.

The variant of Sax implemented by this compiler is a modified version of the core linear, positive fragment introduced in Lab 1, with two additions. First, it supports 32-bit signed integers, leveraging WASM's i32 type. It also supports closures. Both of these types are always unrestricted, supporting all structural rules. Units are also unrestricted.

The performance of a compiler targeting WASM largely depends on the source language. Manually memory-managed languages like C or Rust are often only 20-40% slower through WASM, while more dynamic languages like OCaml or Scala can be 3-8x slower (no sources). Thus, the upper performance bound for linear Sax compiled to WASM should be similar to the performance of C compiled to WASM.

1.1 WASM Structure

The structure and interpretation of WASM programs is largely similar to that of well-known instruction sets like x86 or Java bytecode, with the major difference that all its control flow is structured. It is primarily a stack VM; instructions operate on some combination of bytecode immediates and stack operands. There are no stack manipulation instructions. Instead, there is a per-function set of local values which can be accessed through static indices, emulating registers. Function arguments are passed as locals

WASM also supports both a manually managed heap and a garbage collector. For adjoint Sax, we would like to use the heap for linear values, and the GC for others.

The relevant WASM instructions are:

- `i32.const $n`

  - pushes the i32 immediate n to the stack
- `local.get $n`
  - pushes the local at index n to the stack
- `local.set $n`
  - pops the stack, and sets local n to the popped value
- `i32.load offset=$n`
  - pops an address a from the stack, and loads address $a + n$
  - if the offset is omitted, it default to 0.
- `call $f`
  - calls the function f, popping all its arguments and pushing its result(s).

1.2 WASM Runtimes

There are many different WASM runtimes. The most common are browser engines like Chrome's V8 and Safari's JavaScript Core, however, there are a variety of WASM runtimes aimed at non-web usage. In particular, I chose a runtime called Wasmtime to evaluate the compiler.

WASM does not specify a system-level interface for I/O on its own. Instead, it supports a robust foreign-function interface, through which it requires WASM Runtimes to inject I/O functions. There is a proposal for a standardized WASM system interface called WASI which is gaining support in major runtimes. However, it is not well documented so I opted to embed custom I/O functions in the runtime. Thus, there is a small amount of code which is not runtime agnostic, for now.

Wasmtime is a popular and well-supported runtime focused on secure embedding. It is built as a JIT compiler on top of the Cranelift code generator, which is similar to LLVM but optimized for compilation speed rather than generating optimal code. It already supports WASI, and can be used through a Rust crate. The WASM files generated by this compiler must be run through a small Rust program which injects I/O functions and runs Wasmtime.

1.3 Integers

To achieve competetive performance on real programs, I supplement Sax with a native 32-bit integer type. The only changes to sax are the built-in `int` type and a few built-in metavariables used through Sax's existing `call` command. The supported builtins are as follows:

- `call _const_{n} dest`: parses the integer out of the metavariable name and writes it to `dest`
- `call _add_ dest l r`: writes `(i32.add l r)` to `dest`
- `call _sub_ dest l r`: writes `(i32.sub l r)` to `dest`
- `call _eqz_ dest n`: writes `(i32.eqz n)` to `dest`
  - In WASM, `(i32.eqz n)` pushes `1` when `n` is zero, and `0` otherwise.
  - I did not add a builtin boolean type to Sax. Instead, this just pushes a sum tag with a unit injection. It makes the most sense when used with a type like `+{ 'false : 1, 'true : 1 }`, but is not checked.

Example 1 shows a simple example of using integers in Sax.

```
proc sum_tailrec (d : int) (n : int) (acc : int) =
    cut tst : bool
        call _eqz_ tst n
    read tst {
    | 'true(u) =>
        id d acc
    | 'false(u) => cut n1 : int
                       cut one : int
                           call _const_1 one
                       call _sub_ n1 n one
                   cut nxt : int
                       call _add_ nxt acc n
                   call sum_tailrec d n1 nxt
    }
```

Example 1: `sum_tailrec(n, acc)` = $\sum(0..n) + \mathrm{acc}$

1.4 Compilation Stages and Execution

The compiler and runtime have several components. First, the compiler does a simple pass over the Sax, generating a simple stack-based sequential IR and doing a simple static analysis. Then, it does another pass over the stack-based IR to generate WASM instructions. Finally, the compiler expects the WASM module to be optimized by `wasm-opt`. It should not be strictly necessary, but there seem to be differences in how Wasmtime and `wasm-opt` validate modules, so output directly from the compiler often does not run in Wasmtime without using `wasm-opt` first.

Unlike the Sax compiler from class, this compiler requires a `main` proc, with a destination of any type. The runner will find the main proc and print its output.

1.5 Limitations

There are some limitations driven by time constraints, but they should be fixable without modifying the whole approach.

- The main proc is limited
  - Its return type must be a typename because of how printing works, as described below
  - It does not support Read, because of how the print call is injected
- Subtyping probably has bugs
  - the compiler assumes that all instances of a type have the same layout, but does not enforce this on downcasts
- Memory does not grow
  - the allocator does not ever grow the memory, so it is limited to one WASM page of 64 KiB.

2 Implementation

The translation from Sax to WASM is guided by a few principles. First, the destination of the current translated command is just the top of the stack. Thus, translating a metavariable writing to destination `d` results in a sequence of WASM instructions that result in `d` on the top of the stack.

The translation from the stack IR to WASM carefully tracks the stack and locals. In particular, it tries to track the relationship between different variables, in order to accurately locate them in cases where a stack value may represent both a variable and e.g. the left pi of a pair. I believe this approach is convoluted and could be simplified by taking advantage of properties of the initial translation from Sax to the stack IR. I designed it when I was trying to optimize exact stack and local layout.

Possible stack values are as follow:

- `Addr s`: an address bound to variable `s`
- `GcRef s`: a GC'd reference bound to variable `s`
- `InjTag s`: the tag of plus-type variable `s`
- `InjData s`: the injection of plus-type variable `s`
- `PairFst s`: the π_1 of pair-type variable `s`
- `PairSnd s`: the π_2 of pair-type variable `s`
- `Unit s`: a unit bound to variable `s`
  - These never actually materialize in WASM, and are instead used to pad with zero consts when needed
- `Int s`: an integer bound to variable `s`
  - Integers are passed by value

The basic translation judgement is:

$$S_I \vdash [\![c]\!] = (W; S_O)$$

meaning that translating Sax command c with abstract stack S_I yields a list of WASM instructions W and a stack representation S_O. Our rules will be used pretty informally, and leave out some info about state. It leaves skips stack IR step, so there are some slight differences from the code.

2.1 Allocation and Value Layout

The layout of values is the same on the stack and heap. Cells are always represented as two `i32`s. They may be a pair of primitives, or a primitive followed by a tag, where the primitives could represent an address, unit, or integer.

Since all cells are the same size, our allocator is very simple. We keep an implicit freelist, where the first value of a freed cell is the offset to the next cell. There are two functions:

- `alloc(v1: i32, v2: i32)` writes `v1` and `v2` to the cell on top of the freelist, bumps the freelist pointer by the offset previously written to the cell, and returns the address allocated.
- `free(addr: i32)` writes the current freelist head to the front of the cell, and sets the new freelist pointer to `addr`.

I made some strange choices in this design. First, I opted to start the freelist with every allocated cell by writing an offset of 8 to every cell on initialization. This saves storing an additional pointer to space that has yet to be allocated or freed, but is probably slower when memory is resized. Additionally, I wrote the allocator in Rust, embedding it into the runtime, instead of using WASM's built-in memory instructions. This was just done to save myself from writing WASM by hand. It makes the generated code less portable, but is likely a little bit faster, since the Rust code is not subject to the same safety checks that WASM would have.

2.2 Cuts and Locals

Using our invariants the cut translation is pretty straightforward. Since the destination of any command is the top of the stack, we know that the cut variable will be on top, either as components as a single address. With our stack representation, we can differentiate each case.

The binding created by cut should be randomly accessible by the second command, so we greedily make a local for each cut in case it is not on top of the stack when needed. This creates a lot of churn and local rewriting but `wasm-opt` handles it.

If the top of the stack after the first command is already an address, our translation just creates a local for it. If it is a decomposed cell of an injection tag and value or pair components, we allocate them and then push the address to our locals.

$$\frac{S \vdash [\![P]\!] = W_P; \text{Addr x} :: S_P \qquad S_P \vdash [\![Q]\!] = W_Q; S_Q}{S \vdash [\![\textbf{cut } (x : T)P(x)Q(x)]\!] = W_P :: \textbf{local.set } n :: W_Q; S_Q}\,\text{Cut}_{\text{addr}}$$

$$\frac{S \vdash [\![P]\!] = W_P; \diamond :: \diamond :: S_P \qquad S_P \vdash [\![Q]\!] = W_Q; S_Q}{S \vdash [\![\textbf{cut } (x : T)P(x)Q(x)]\!] = W_P :: \textbf{call } \text{alloc} :: \textbf{local.set } n :: W_Q; S_Q}\,\text{Cut}_{\text{pair}}$$

Note that Cut_{pair} applies to both pairs and injections, or even pairs of addresses where the top address is not the destination.

2.3 Read

Read commands dereference and free an address, put its components into locals, and then build a switch if there are multiple cases.

$$\frac{S \vdash [\![c]\!] = W_C; S_C}{S \vdash [\![\textbf{read } s \ (l, r) \ c]\!] = \text{DerefPair}(L_s) :: W_C; S_C}\,\text{Read}_{\text{Pair}}$$

$$\begin{aligned}\text{DerefPair}(L_s) = \textbf{local.get } L_s :: \textbf{i32.load} :: \textbf{local.set } L_{s_{\pi_1}} \\ :: \textbf{local.get } L_s :: \textbf{i32.load } \text{offset=4} :: \textbf{local.set } L_{s_{\pi_2}} \\ :: \textbf{local.get } L_s :: \textbf{call } \text{free}; S\end{aligned}$$

2.4 Write

Write commands move something onto the top of the stack.

$$\frac{}{S \vdash [\![\textbf{write } d \ ()]\!] = \textbf{i32.const } 0; () :: S}\,\text{Write}_{\text{Unit}}$$

$$\frac{}{S \vdash [\![\textbf{write } d \ (\pi_1, \pi_2)]\!] = \textbf{local.get } L_{\pi_1} :: \textbf{local.get } L_{\pi_2}; \text{PairFst d} :: \text{PairSnd d} :: S}\,\text{Write}_{\text{Pair}}$$

$$\frac{}{S \vdash [\![\textbf{write } d \ 'l(a)]\!] = \textbf{local.get } a :: \textbf{i32.const } T; \text{InjTag d} :: \text{InjData d} :: S}\,\text{Write}_{\text{Plus}}$$

2.5 Calls and Tail Calls

The call translation is straightforward.

$$\frac{}{S \vdash [\![\textbf{call } p \ d \ a_1 ... a_n]\!] = \textbf{local.get } L_{a_1} ... L_{a_n} :: \textbf{call } p; \text{Addr d} :: S}\,\text{Call}$$

Sax's only iteration construct is recursion. To avoid stack-overflows, we need to properly tail recurse on tail-recursive Sax functions. This is straightforward using WASM's tail call proposal, which has been merged and is supported by most runtimes. However, I found that it is still a bit slow in Wasmtime.

To detect when a tall is a tail-call, we just check if the destination of a call is the destination of the whole function. Then, it suffices to replace the WASM `call` instruction with `return_call`.

2.6 Id

Id is similar to Write. We just need to ensure that the required address is on top of the stack; there are two cases. If it is already on top, we continue, otherwise we must fetch it from our locals.

2.7 Closures

Closures are garbage collected. There are some considerations to make about adjoint Sax, since all closures are currently unrestricted. GC references can not be stored on the heap, so linear data cannot reference unrestricted closures, but we can still store heap references in the GC. This aligns with adjoint Sax's mode preorder restriction.

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere.

2.8 Printing

Printing is a little complicated. Since tags are indexed by type, we need the full type of an object to print it. It would be possible to generate a print function for each type, but I instead opted to write a single print function in Rust which accepts the type index as a parameter. To do so, we must serialize every type def so that it is accessible to the runtime. The compiler uses yojson to serialize the type defs to JSON, and then exports it to the runtime using WASM's data segments. This undoubtedly makes printing pretty slow.

2.9 Optimizations

We implement a few optimizations, mostly aimed at minimizing allocations.

2.9.1 Unboxed Cut/Read

If the variable instantiated by a cut is read in the same function, we skip allocating it. Instead, we put its components in locals. This modifies the translation rules for Cut and Read.

2.9.2 Cut/Id

The simple Cut/Id optimization is implicit through our translation of Id.

2.9.3 Vertical Reuse

Though we could easily implement vertical reuse in the compilation, it might be interesting to note that it the allocator design replicates this behavior, since the freelist is essentially a stack and the most recently freed cells are the next allocated ones. Implementing it through the compiler would save some freelist manipulation but the performance gain might be quite small.

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere.

3 Evaluation

Holistically, the generated code looks good. The optimizations catch many overaggressive allocations, and `wasm-opt` minimizes stack and local manipulation.

3.1 Cut/Id Example

The following example shows some optimizations in action. z is instantly id'd to x after being cut, so it could just be a substitution. This function ends up having no allocations except for potentially allocating `'zero(w)`.

```
proc pred (d : nat) (x : nat) =
    cut z : nat
        id z x
    read z {
    | 'zero(u) => cut w : 1
                    id w u
                  write d 'zero(w)
    | 'succ(y) => id d y
    }
```

Example 2: The cut/id example from class

```
  (func $pred (param i32) (result i32)
    (local i32 i32 i32 i32)
    (local.get $x)
    (local.set $z) ;; duplicate locals, because of id z x
    (local.get $z) ;; removed by `wasm-opt`

    ;; reads z's components into locals, and then frees it
    (i32.load 0) ;; read z
    (local.set $z_inj)
    (local.get $z)
    (i32.load 0 offset=4)
    (local.set $z_tag)
    (local.get $z)
    (call $free)

    ;; switch over z's tag
    (block
      (block (block (i32.const 0) (local.get $z_tag) (br_table 0 1 0)) (i32.const 0)
(i32.const 0) (return_call alloc))
      (local.get 2)
      (return)
    )
  )
```

Example 3: Compiler output for pred, before `wasm-opt`

3.2 Listrev Example

```
proc reverse (d : list) (l : list) (acc : list) =
    read l {
    | 'nil(u) =>
        read u ()
            id d acc
    | 'cons(ls) =>
        read ls (hd, tl)
            cut new : list
                cut new_p : bin * list
                    write new_p (hd, acc)
                write new 'cons(new_p)
            call reverse d tl new
    }
```

Example 4: `proc reverse` reverses a list tail-recursively

```
(func $reverse (param $l i32) (param $acc i32) (result i32)
 (local $2 $3 i32)
 (local.set $l_inj (i32.load (local.get $l)))
 (local.set $l_tag (i32.load offset=4 (local.get $0))
 )
 (call $free (local.get $l))
 ;; wasm_opt transforms a switch (`br_table`) over two cases into an `if`
 (if
  (i32.ne (local.get $l_tag) (i32.const 1))
  (then (return (local.get $acc))))
 ;; recycled the local for `l` for `hd`
 (local.set $l (i32.load (local.get $l_inj))) ;; read `ls`,
 ;; recycled to local for `l_inj` for `tl`
 (local.set $l_tag (i32.load offset=4 (local.get $l_inj)))
 (call free (local.get $l_inj))
 (return_call $reverse
  (local.get $l_tag) ;; tl
  (call alloc
   (call alloc (local.get $l) (local.get $l_tag)) ;; allocating (hd, acc)
   (i32.const 1) ;; 'cons tag
  ) ;; new
 )
)
```

Example 5: Compiled output for `reverse`, after `wasm-opt`

3.3 Benchmarks

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At.

3.4 Iteration and Tail Calls

Despite using WASM's native tail calls, iteration speed is pretty slow. For the `sum_tailrec` function in Example 1 the compiler generates pretty idiomatic recursive WASM below. After hand-translating it to a loop, it becomes around 10x faster for summing the first billion integers.

```
(func $sum_tailrec (param $0 i32) (param $1 i32) (result i32)
 (if (local.get $0)
  (then (return_call $8
        (i32.sub (local.get $0) (i32.const 1))
        (i32.add (local.get $0) (local.get $1)))))
 (local.get $1))
```

Example 6: compiled `sum_tailrec`, tail recursive

```
(func $sum_loop (param $0 i32) (param $1 i32) (result i32)
  (block $done
    (loop $continue
      (br_if $done (i32.eqz (local.get $0)))
      (local.set $1 (i32.add (local.get $0) (local.get $1)))
      (local.set $0 (i32.sub (local.get $0) (i32.const 1)))
      (br $continue)))
  (local.get $1))
```

Example 7: compiled `sum_tailrec`, loop

This is probably runtime dependent, and might be fixed by a Wasmtime update. However, a more robust compiler might want to turn some recursive functions into loops.

4 Related Work

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos irridente, statua est in quo a nobis philosophia defensa et collaudata est, cum id, quod maxime placeat, facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et aut officiis debitis aut rerum necessitatibus saepe eveniet, ut et voluptates repudiandae sint et molestiae non recusandae. Itaque earum rerum defuturum, quas natura non depravata desiderat. Et quem ad me accedis, saluto: 'chaere,' inquam, 'Tite!' lictores, turma omnis chorusque: 'chaere, Tite!' hinc hostis mi Albucius, hinc inimicus. Sed iure Mucius. Ego autem mirari satis non queo unde hoc sit tam insolens domesticarum rerum fastidium. Non est omnino hic docendi locus; sed ita prorsus existimo, neque eum Torquatum, qui hoc primus cognomen invenerit, aut torquem illum hosti detraxisse, ut aliquam ex eo est consecutus? – Laudem et caritatem, quae sunt vitae sine metu degendae praesidia firmissima. – Filium morte multavit. – Si sine causa, nollem me ab eo delectari, quod ista Platonis, Aristoteli, Theophrasti orationis ornamenta neglexerit. Nam illud quidem physici, credere aliquid esse minimum, quod profecto numquam putavisset, si a.

5 Conclusion

We have a compiler from a Sax variant to WASM, which takes advantage of linear types for a compact cell representation and inserted `free` calls. It uses a few optimizations to minimize allocations and the GC for unrestricted types. It also supports WASM's i32 type.

5.1 Future Work

Aside from fixing the limitations described in the introduction, there are quite a few ways to extend the compiler by either supporting more Sax features or adding further optimizations.

5.1.1 Lazy Records

I did not have time to implement lazy records, but they should be similar to closures.

5.1.2 Lazily allocating function returns

Currently, cuts within a function are not allocated if they are Read within the function. It might make sense to let all functions return without allocating first, so the caller can decide not to allocate if the return is Read.

5.1.3 Adjoint Types

Adjoint types should fit neatly into the existing framework. Nonlinear addresses would go on the GC, similarly to closures. One complication is that we cannot allocate linear closures or records with the current allocator that only supports one chunk size. Additionally, we cannot store GC references on the heap, so we would not be able to store linear values which refer to linear closures. However, this could be solved by updating the allocator to support multiple chunk sizes.

5.1.4 Inlining

`wasm-opt` actually inlines some functions, but currently all parameters and returns must be allocated. Lazily allocating returns could help, but inlining smaller procedures along with the unboxed cut/read optimization might reduce allocations further.

5.1.5 Transforming tail-calls to loops

WASM's `return-call` instruction may be slower than `loop` in runtimes other than Wasmtime. It should not be too difficult to transform them directly to WASM loops, and would provide a significant speedup for tail recursisve functions.