Wasocaml: compiling OCaml to WebAssembly

Léo Andrès, Pierre Chambart, Jean-Christophe Filliâtre

Wasocaml: compiling OCaml to WebAssembly

LÉO ANDRÈS*, Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, Laboratoire Méthodes Formelles, France
PIERRE CHAMBART, OCamlPro SAS, France
JEAN-CHRISTOPHE FILLIÂTRE, Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, Laboratoire Méthodes Formelles, France

The limitations of JavaScript as the default language of the web led to the development of Wasm, a secure, efficient and modular language. However, compiling garbage-collected languages to Wasm presents challenges, including the need to compile or re-implement the runtime. Some Wasm extensions such as Wasm-GC are developed by the Wasm working groups to facilitate the compilation of garbage-collected languages. We present Wasocaml, an OCaml to Wasm-GC compiler. It is the first compiler for a real-world functional programming language targeting Wasm-GC. Wasocaml confirms the adequacy of the Wasm-GC proposal for a functional language and had an impact on the design of the proposal. Moreover, the compilation strategies developed within Wasocaml are applicable to other compilers and languages. Indeed, two compilers already used a design similar to ours. Finally, we describe how we plan to handle the C/JavaScript FFIs and effect handlers, in order to allow developers to easily deploy programs mixing C, JavaScript and OCaml code to the web, while maintaining good performances.

CCS Concepts: • **Software and its engineering → Compilers**.

Additional Key Words and Phrases: OCaml, Wasm, compilation, value representation, memory, FFI

ACM Reference Format:
Léo Andrès, Pierre Chambart, and Jean-Christophe Filliâtre. 2024. Wasocaml: compiling OCaml to WebAssembly. 1, 1 (September 2024), 29 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Context

JavaScript is the de facto language of the web. However, it does show its limitations when it comes to performance, security and safety. In order to remedy this, WebAssembly (Wasm for short) [3] has been developed as a secure modular language of predictable performance. Its usage is expanding beyond the web, e.g. finding applications in the cloud (Fastly, Cloudflare) and in the creation of portable binaries.

The first version of Wasm was meant to compile libraries in languages such as C, C++ or Rust to expose them for consumption by a JavaScript program. It was designed with the explicit aim that future versions would catter to the specific needs of other languages and uses. This version of Wasm can be seen as simplified C. Here is an example:

*Also with OCamlPro SAS.

Authors' Contact Information: Léo Andrès, contact@ndrs.fr, Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, Laboratoire Méthodes Formelles, Gif-sur-Yvette, France; Pierre Chambart, pierre.chambart@ocamlpro.com, OCamlPro SAS, Paris, France; Jean-Christophe Filliâtre, jean-christophe.filliatre@cnrs.fr, Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, Laboratoire Méthodes Formelles, Gif-sur-Yvette, France.

Manuscript submitted to ACM

```
(func $fact (param $x i32) (result i32)
  (if (i32.eq (local.get $x) (i32.const 0))
    (then (i32.const 1))
  (else
    (i32.mul
      (local.get $x)
      (call $fact
        (i32.sub (local.get $x) (i32.const 1)))))))
```

A Wasm module is executed inside what is called an *embedder* or a *host*. In the context of the web, it is a browser and the host language is JavaScript. A Wasm program only manipulates scalar values (integers and floats) but not garbage collected values coming from the host language, such as JavaScript objects. This would be useful in a browser context, but requires some interaction with the GC of the embedder.

Various extensions have been in development for Wasm, some of them being part of the standard already. Our work is based on some of them, that is to say the reference types [20], typed function references [21], GC [18], exception handling [17] and tail calls [19] extensions.

References and GC. The aim of these proposals is to allow the program to manipulate references to different kinds of values such as opaque objects, functions, and garbage-collected values. Since Wasm has to be memory safe, those values are not exposed as raw pointers. Instead, the type system guarantees the proper manipulation of such values.

For instance, in the following example, the `struct` and the `array` can only be manipulated through some primitives such as `struct.get` and `array.get`, there's no way to access their memory address:

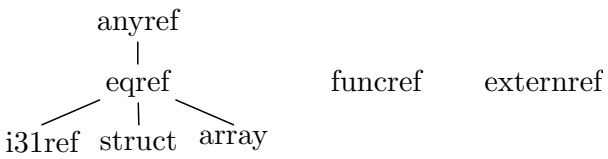
```
(type $f1 (func (param i32) (result i32)))
(type $t1 (array i31ref))
(type $t2 (struct
  ((field $a (ref $t1))
   (field $b (ref $f1))
   (field $c i32))))

(func (param $x (ref $t2)) (result i32)
  (i31.get_u
    (array.get $t1
      (i32.const 0)
      (struct.get $t2 $a (local.get $x)))))
```

However, Wasm is rather a compilation target than a programming language. It needs to be able to represent the values of any kind of source language. It was not deemed possible to have a type system powerful enough to do that. Instead, the decision was made to have a simple type system, but to rely on dynamic casts to fill the gaps, and guarantee that those casts are efficient.

There is a sub-typing relation that controls which casts are allowed. Some types are predefined by Wasm: `anyref`, `eqref`, `i31ref`, **`funcref`** and `externref`. Others are user defined: `array` and `struct`. There is a structural sub-typing

relation between user types. Upcasts are implicit, downcasts are explicit and incorrect ones lead to runtime errors. It is possible to dynamically test for type compatibility. The following figure depicts the sub-typing relation:



Contribution to the proposals. The general rule for the Wasm committee is to only include features with a demonstrated use case. As there are currently very few compilers targeting the GC-proposal, some features were lacking conclusive evidence of their usefulness. An example is the `i31ref` type that is not required by the Dart compiler (the only one targeting the GC-proposal at the time). Wasocaml demonstrates the usefulness of `i31ref`. It also validates the GC-proposal on a functional language. We presented Wasocaml to the Wasm-GC working group [2]. It helped in convincing the working group to keep `i31ref` in the proposal.

1.0.1 Exceptions. Another serious limitation of Wasm1 is the absence of an exception mechanism. This is not a problem for compiling either C or Rust, but C++ does require them. It is of course possible to encode exception, but this has a significant runtime cost and limits the interactions with JavaScript (that has exceptions).

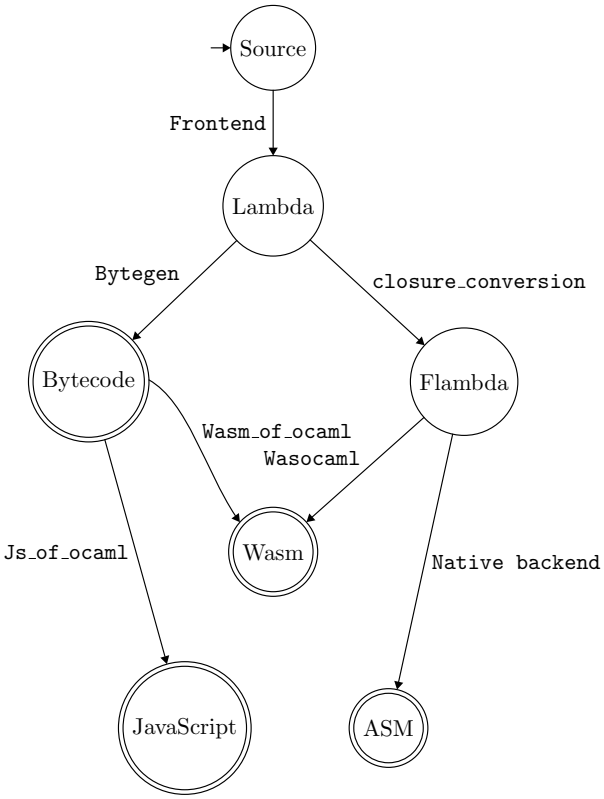
1.0.2 Tail Calls. Wasm1 does not allow tail calls to be optimized. This is a show-stopper when compiling functional languages. In the tail-call proposal, new instructions such as `return_call` guarantee that a tail-call will be optimized.

2 Source Language

What we are interested in is compiling OCaml to Wasm. Before being able to compile, we have to choose where to start from in the OCaml compiler. In the following subsection we will describe the possible choices and then we will give a proper small-step semantics of the chosen language.

2.1 OCaml Intermediate Representation Choice

OCaml has many intermediate representations (IR for short). In the *Lambda* IR, closures are still implicit and the code has not been optimized. The *Bytecode* representation is not well optimized. The *Clambda* one is too low-level for Wasm because code pointers and values are mixed inside closures. *Cmm* is even more low-level: it has pointer arithmetic. The *Flambda* IR is simpler than *Flambda2*, thus it is the one we choosed for our first prototype. We will use Flambda2 in the future, as it provides better optimisations. The following figure depicts a simplified version of the OCaml compilation chain:



2.2 Native OCaml Value Representation

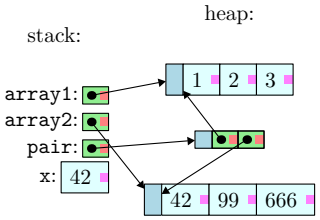
The various OCaml IRs are working on a uniform representation of values. There are two kinds of values: small scalars and heap-allocated blocks.

At runtime, it is possible to test if a value is a scalar or a block. In the native backend, this is done through a technique known as *pointer tagging*, but in the Flambda IR, the precise representation is not imposed yet.

Small scalars are used to represent booleans, unboxed integers, characters or constant constructors of a sum type. Heap-allocated blocks are used to represent arrays, non-constant constructors of a sum type or pairs. For instance, consider the following OCaml code:

```
let array1 = [| 1; 2; 3 |]
let array2 = [| 42; 99; 666 |]
let pair = (array1, array2)
let x = 42
```

In memory, it would be represented as follows:



Note that a given type can have values of both kinds. For instance the empty list is a scalar value but a list with at least one element will be a heap-allocated block.

2.3 Flambda semantics

We give a formal semantics for a subset of Flambda. Compared to the full Flambda it has the following limitations: the objects related primitives have been removed, functions only have one argument, exceptions have been removed but we still have static exceptions that are described later.

2.3.1 *Abstract syntax.*

Value. A value v is either an integer n or an address a.

$$v ::= n \quad \text{(integer)}$$
$$\mid a \quad \text{(address)}$$

An integer represents a value of many types in the OCaml source language such as an `int`, a `bool`, a `char` or the constant constructors of a sum type. An address points to a heap-allocated chunk.

Store. A store $\mathscr{M}$ is a map from addresses to heap-allocated chunks. It is used to represent the state of the memory. Heap-allocated chunks can either be a block made of a tag n and a list of values v^*, a closure $\mathscr{S}.x$, or a set of closures $\mathscr{S}$. A closure is made of a set of closures $\mathscr{S}$ and of a name x which is its identifier inside the set.

$$\mathscr{M} ::= a \mapsto n, v^* \quad \text{(block)}$$
$$\mid \mathscr{S}.x \quad \text{(closure)}$$
$$\mid \mathscr{S} \quad \text{(set of closure)}$$

Set of closures. A set of closures $\mathscr{S}$ represents a set of mutually recursive functions.

$$\mathscr{S} ::= \mathscr{V}, \mathscr{C}$$

It is made of two maps $\mathscr{V}$ and $\mathscr{C}$. The map $\mathscr{V}$ is the environment (the values captured by the closures). An environment is a map from names to values.

$$\mathscr{V} ::= x \mapsto v$$

The map $\mathscr{C}$ is the closure map (the code of each function). It is a map from names to codes.

$$\mathscr{C} ::= x \mapsto \lambda\, x\, x\, .\, t$$

The code is a function that takes two arguments. The first argument is the real argument of the function. The second one is the address of the closure being called. It is used to access the environment or to call other functions from the same set of mutually recursive functions. The body t of the function is a term (see below).

Binary operator.

$$
\begin{array}{rll}
op & ::= & eq \quad \text{(equality)} \\
& | & add \quad \text{(addition)} \\
& | & sub \quad \text{(subtraction)}
\end{array}
$$

Branches.

$$\mathscr{B} ::= n \ \mapsto \ t$$

Term.

$$
\begin{array}{rll}
t & ::= & v \quad \text{(value)} \\
& | & x \quad \text{(identifier)} \\
& | & \texttt{let}\ x = t\ \texttt{in}\ t \quad \text{(let-binding)} \\
& | & x\ x \quad \text{(application)} \\
& | & x\ \texttt{<-}\ x \quad \text{(mutation)} \\
& | & \texttt{if}\ x\ \texttt{then}\ t\ \texttt{else}\ t \quad \text{(conditional)} \\
& | & \texttt{switch}\ x\ \mathscr{B}\ \mathscr{B} \quad \text{(switch)} \\
& | & \texttt{sraise}\ x\ x^* \quad \text{(static raise)} \\
& | & \texttt{scatch}\ t\ \texttt{with}\ x\ x^*\ t \quad \text{(static catch)} \\
& | & \texttt{while}_t\ t\ \texttt{do}\ t \quad \text{(while loop)} \\
& | & \texttt{get\_field}\ n\ x \quad \text{(block field read)} \\
& | & \texttt{set\_field}\ n\ x\ x \quad \text{(block field write)} \\
& | & \texttt{make\_block}\ n\ x^* \quad \text{(block creation)} \\
& | & \texttt{project\_closure}\ x\ x \quad \text{(closure projection)} \\
& | & \texttt{project\_var}\ x\ x \quad \text{(closure environment)} \\
& | & \texttt{move\_within\_closure}\ x\ x \quad \text{(closure movement)} \\
& | & \texttt{make\_set\_of\_closures}\ \mathscr{S} \quad \text{(closures allocation)} \\
& | & \texttt{pop}\ t \quad \text{(pop environment)} \\
& | & x\ op\ x \quad \text{(binary operator)}
\end{array}
$$

Note that the pop instruction is an *administrative instruction* that is never present in the concrete syntax coming from user code.

2.3.2 *Small-step semantics.*

Configuration. Our semantics is operating on a *configuration*. A configuration is of the form $\mathscr{V}^*, \mathscr{M}, t$ where $\mathscr{V}^*$ is a stack of environments representing the call stack ; $\mathscr{M}$ is a store representing the global memory of the program ; t is the current term being evaluated.

Reduction context.

$$
\begin{aligned}
E ::= & \square \\
& \mid \texttt{let}\ x = E\ \texttt{in}\ t \\
& \mid \texttt{while}_t\ E\ \texttt{do}\ t \\
& \mid \texttt{scatch}\ E\ \texttt{with}\ x\ x^*\ t \\
& \mid \texttt{pop}\ E
\end{aligned}
$$

Reduction inside a context.

$$
\frac{\mathscr{V}^*, \mathscr{M}, t_1 \rightarrow \mathscr{V}'^*, \mathscr{M}', t_2}{\mathscr{V}^*, \mathscr{M}, E(t_1) \rightarrow \mathscr{V}'^*, \mathscr{M}', E(t_2)}
$$

Head reductions. In order to reduce cluttering, $\mathscr{V}^*$ and $\mathscr{M}$ are removed from the configuration in the left-hand side of the reduction rules when they are not read and not modified but also from the right-hand side when they are not modified.

$$
\text{identifier}\ \frac{\mathscr{V}_0(x) = v}{\mathscr{V}_0 \mathscr{V}^*, x \rightarrow v}
$$

An identifier simply reduces to its image in the current environment.

$$
\text{let-binding}\ \frac{}{\begin{array}{lll} & \mathscr{V}_0 \mathscr{V}^*, & \texttt{let}\ x = v\ \texttt{in}\ t \\ \rightarrow & \mathscr{V}_0[x \leftarrow v] \mathscr{V}^*, & t \end{array}}
$$

A let-binding $\texttt{let}\ x = v\ \texttt{in}\ t$ reduces to t and adds a binding from x to v in the current environment.

$$
\text{application}\ \frac{\begin{array}{c} \mathscr{V}_0(x_1) = a \qquad \mathscr{M}(a) = \mathscr{S}.x \\ \mathscr{S} = \_, \mathscr{C} \qquad \mathscr{C}(x) = \lambda\ x_3\ x_4\ .\ t \\ \mathscr{V}_0(x_2) = v \end{array}}{\begin{array}{lll} & \mathscr{V}_0 \mathscr{V}^*, & \mathscr{M}, \quad x_1\ x_2 \\ \rightarrow & [x_3 \leftarrow v; x_4 \leftarrow a] \mathscr{V}_0 \mathscr{V}^*, & \texttt{pop}\ t \end{array}}
$$

An application $x_1\ x_2$ looks for an address a in the current environment. The image of a in the store must be a closure $\mathscr{S}.x$ where the set of closures $\mathscr{S} = \_, \mathscr{C}$. Let the value of x in the closure map $\mathscr{C}$ be $\lambda\ x_3\ x_4\ .\ t$. The term reduces to $\texttt{pop}\ t$ and we create a new environment containing two bindings: x_3 is mapped to the value of x_2 in the current environment and x_4 is mapped to a.

$$\text{pop environment}\ \frac{}{\mathscr{V}_0\mathscr{V}^*, \mathsf{pop}\ v \longrightarrow \mathscr{V}^*, v}$$

This simply throws away the current environment. This is used to model the call stack. A pop is inserted around each function application and removed once the function is fully evaluated to a value.

$$\text{mutate}\ \frac{\mathscr{V}_0(x_1) = v_1 \qquad \mathscr{V}_0(x_2) = v_2}{\mathscr{V}_0\mathscr{V}^*, x_1 \texttt{ <- } x_2 \longrightarrow \mathscr{V}_0[x_1 \leftarrow v_2]\mathscr{V}^*, 0}$$

A mutation x_1 `<-` x_2 simply updates the value of x_1 to be the value of x_2 in the current environment. It reduces to 0 which corresponds to the value `()` : **`unit`** in OCaml.

$$\text{conditional 1}\ \frac{\mathscr{V}_0(x) = n \qquad n \neq 0}{\begin{array}{rll} & \mathscr{V}_0\mathscr{V}^*, & \texttt{if}\ x\ \texttt{then}\ t_1\ \texttt{else}\ t_2 \\ \longrightarrow & & t_1 \end{array}}$$

A conditional `if` x `then` t_1 `else` t_2 reduces to t_1 when the value of x in the current environment is not 0 (i.e. when it is `true` : **`bool`** in OCaml).

$$\text{conditional 2}\ \frac{\mathscr{V}_0(x) = n \qquad n = 0}{\begin{array}{rll} & \mathscr{V}_0\mathscr{V}^*, & \texttt{if}\ x\ \texttt{then}\ t_1\ \texttt{else}\ t_2 \\ \longrightarrow & & t_2 \end{array}}$$

Similarly, it reduces to t_2 when the value of x is 0 (i.e. when it is `false` : **`bool`** in OCaml).

$$\text{switch 1}\ \frac{\mathscr{V}_0(x) = n \qquad \mathscr{B}_1(n) = t}{\mathscr{V}_0\mathscr{V}^*, \texttt{switch}\ x\ \mathscr{B}_1\ \mathscr{B}_2 \longrightarrow t}$$

When the value of x in the current environment is an integer n, then `switch` x $\mathscr{B}_1$ $\mathscr{B}_2$ reduces to the image of n in $\mathscr{B}_1$.

$$\text{switch 2}\ \frac{\mathscr{V}_0(x) = a \qquad \mathscr{M}(a) = n, v^* \qquad \mathscr{B}_2(n) = t}{\mathscr{V}_0\mathscr{V}^*, \mathscr{M}, \texttt{switch}\ x\ \mathscr{B}_1\ \mathscr{B}_2 \longrightarrow t}$$

Similarly, when x is an address a and $\mathscr{M}(a)$ is a block with tag n, then `switch` x $\mathscr{B}_1$ $\mathscr{B}_2$ reduces to the image of n in $\mathscr{B}_2$.

$$\text{raise 1}\ \frac{x_{exn} \neq x'_{exn}}{\begin{array}{rl} & \texttt{scatch}\ (\texttt{sraise}\ x_{exn}\ x^n)\ \texttt{with}\ x'_{exn}\ x'^n\ t \\ \longrightarrow & \texttt{sraise}\ x_{exn}\ x^n \end{array}}$$

If a raised exception does not match the exception expected by a static catch, then the catch is discarded and we re-raise the exception.

$$\text{raise 2}\ \frac{\begin{array}{c} x_{exn} = x'_{exn} \qquad x^n = x_0, \ldots, x_{n-1} \\ x'^n = x'_0, \ldots, x'_{n-1} \qquad \forall i \in [0; n[, \mathscr{V}_0(x_i) = v_i \end{array}}{\begin{array}{l} \mathscr{V}_0 \mathscr{V}^*, \\ \quad \texttt{scatch}\ (\texttt{sraise}\ x_{exn}\ x^n)\ \texttt{with}\ x'_{exn}\ x'^n\ t \\ \rightarrow \quad \mathscr{V}_0[x'_0 \leftarrow v_0, \ldots, x'_{n-1} \leftarrow v_{n-1}]\mathscr{V}^*, \\ \quad t \end{array}}$$

On the contrary, when the two exceptions match, the identifiers specified in the catch are binded to the values of the identifiers transported by the exception. The values are taken from the current environment. We have the guarantee that if the transported identifiers were in the environment when we raised, then they are also in the environment when we catch. This is because static exceptions never cross a function application. Finally, the reduced term is the one attached to the catch.

$$\text{raise 3}\ \frac{}{\begin{array}{l} \texttt{while}_{t,}\ (\texttt{sraise}\ x_{exn}\ x^*)\ \texttt{do}\ t \\ \rightarrow \quad \texttt{sraise}\ x_{exn}\ x^* \end{array}}$$

A while simply re-raises static exceptions.

$$\text{raise 4}\ \frac{}{\begin{array}{l} \texttt{let}\ x = (\texttt{sraise}\ x_{exn}\ x^*)\ \texttt{in}\ t \\ \rightarrow \quad \texttt{sraise}\ x_{exn}\ x^* \end{array}}$$

Similarly, a let-binding also re-raises static exceptions.

$$\text{while loop 1}\ \frac{n = 0}{\texttt{while}_{t,}\ n\ \texttt{do}\ t \rightarrow 0}$$

A loop $\texttt{while}_{t,}\ n\ \texttt{do}\ t$ reduces to 0 when $n = 0$.

$$\text{while loop 2}\ \frac{n \neq 0}{\begin{array}{l} \texttt{while}_{t,}\ n\ \texttt{do}\ t \\ \rightarrow \texttt{let}\ \_ = t\ \texttt{in}\ \texttt{while}_{t,}\ t_{,}\ \texttt{do}\ t \end{array}}$$

On the contrary, when $n \neq 0$, it reduces to $\texttt{let}\ \_ = t\ \texttt{in}\ \texttt{while}_{t,}\ t_{,}\ \texttt{do}\ t$. The term $t_{,}$ corresponds to the original term that was used as a condition.

$$\text{field read}\ \frac{\begin{array}{c} \mathscr{V}_0(x) = a \qquad \mathscr{M}(a) = tag, v^* \\ v^* = v_0, \ldots, v_{m-1} \qquad m > n \geq 0 \end{array}}{\mathscr{V}_0 \mathscr{V}^*, \mathscr{M}, \texttt{get\_field}\ n\ x \rightarrow v_n}$$

The term $\texttt{get\_field}\ n\ x$ reduces to the n^{th} value of the block stored at address a with a being the value of x in the current environment.

$$\text{field write}\ \frac{\begin{array}{ll}\mathscr{V}_0(x_1) = a & \mathscr{M}(a) = tag, v^* \qquad v^* = v_0, \ldots, v_{m-1} \\ m > n \geq 0 & \mathscr{V}_0(x_2) = v\end{array}}{\begin{array}{l}\mathscr{V}_0\mathscr{V}^*, \\ \mathscr{M}, \\ \texttt{set\_field}\ n\ x_1\ x_2 \\ \rightarrow\quad \mathscr{M}[a \leftarrow tag, v_0, \ldots, v_{n-1}, v, v_{n+1}, \ldots, v_{m-1}], \\ 0\end{array}}$$

If x_1 is the address a in the current environment, then the term `set_field` n x_1 x_2 sets the n^{th} value of the block stored at address a to the value of x_2 in the current environment.

$$\text{block creation}\ \frac{\mathscr{V}_0(x_0) = v_0, \ldots, \mathscr{V}_0(x_{n-1}) = v_{n-1} \qquad a \notin \text{dom}(\mathscr{M})}{\begin{array}{l}\mathscr{V}_0\mathscr{V}^*, \mathscr{M}, \texttt{make\_block}\ n_{tag}\ x_0, \ldots, x_{n-1} \\ \rightarrow\quad \mathscr{M}[a \leftarrow n_{tag}, v_0, \ldots, v_{n-1}], 0\end{array}}$$

This creates a new block in the store at a fresh address. The tag is given as a value while the others values of the block are read from the current environment.

$$\text{closure projection}\ \frac{\begin{array}{c}\mathscr{V}_0(x_2) = a_1 \\ \mathscr{M}(a_1) = \mathscr{S} \qquad \mathscr{M}(a_2) = \mathscr{S}.x_1\end{array}}{\begin{array}{l}\mathscr{V}_0\mathscr{V}^*, \mathscr{M}, \texttt{project\_closure}\ x_1\ x_2 \\ \rightarrow\quad a_2\end{array}}$$

This reduces to the address a_2 of the closure which has the identifier x_1 in the set of closures at address a_1, with a_1 being the value of x_2 in the current environment. In short, it allows one to get a particular function from a previously allocated set of closures.

$$\text{variable projection}\ \frac{\begin{array}{ll}\mathscr{V}_0(x_2) = a & \mathscr{M}(a) = \mathscr{S}.x \\ \mathscr{S} = \mathscr{V}, \_ & \mathscr{V}(x_1) = v\end{array}}{\mathscr{V}_0\mathscr{V}^*, \mathscr{M}, \texttt{project\_var}\ x_1\ x_2 \rightarrow v}$$

This reduces to the value of x_1 in the environment of the set of closures that contains the closure at address a_a, with a_a being the value of x_2 in the current environment. In short, when inside a closure, it allows to read a variable from the set of closures the closure belongs to.

$$\text{closure movement}\ \frac{\begin{array}{c}\mathscr{V}_0(x_2) = a_1 \\ \mathscr{M}(a_1) = \mathscr{S}.x \qquad \mathscr{M}(a_2) = \mathscr{S}.x_1\end{array}}{\begin{array}{l}\mathscr{V}_0\mathscr{V}^*, \mathscr{M}, \texttt{move\_within\_closure}\ x_1\ x_2 \\ \rightarrow\quad a_2\end{array}}$$

This reduces to the address a_2 of the closure x_1 in the set of closures to which the closure at address a_1 belongs to. In short, when inside a closure, it allows to get another closure in the current set of closures.

$$\text{set alloc}\ \frac{\begin{array}{c}\mathscr{S} = \_\,, \mathscr{C} \qquad x_0 \in \mathrm{dom}(\mathscr{C}), \ldots, x_{n-1} \in \mathrm{dom}(\mathscr{C}) \\ \forall a \in [a_0, \ldots, a_n], a \notin \mathrm{dom}(\mathscr{M})\end{array}}{\begin{array}{rl} & \mathscr{M}\,, \texttt{make\_set\_of\_closures}\ \mathscr{S} \\ \rightarrow & \mathscr{M}\,[a_n \leftarrow \mathscr{S}; a_0 \leftarrow \mathscr{S}.x_0; \ldots; a_{n-1} \leftarrow \mathscr{S}.x_{n-1}], \\ & a_n\end{array}}$$

This maps the set of closures $\mathscr{S}$ and all its closures to fresh addresses in the store. The term reduces to the address of the set.

2.4 Example

Consider the following OCaml code:

```
let f = print_int
let rec iter f l =
  match l with
  | [] -> ()
  | hd :: tl ->
    f hd;
    iter f tl
```

In Flambda, it would be represented as follows:

```
let f = (* ... *)
let iter =
  let set =
    make_closures
    | cl_iter { f } env ->
      let otherset =
        make_set_of_closures
        (* this is the closure map *)
        | cl_iter_f { l } envtwo ->
          switch l
          | int 0 -> 0
          | tag 0 ->
            let x = get_field 0 l in
            let f = project_var f envtwo in
            let dummy = f x in
            let tl = get_field 1 l in
            envtwo tl
        (* this is the environment *)
        | f -> f
        end
      in
```

```
      project_closure cl_iter_f otherset
  in
  project_closure cl_iter set
in
(* ... *)
```

3 Compiling Flambda to Wasm

Wasm code runs inside an *embedder*. In the case of the web it is the browser and the host language is JavaScript, which has a GC. As OCaml also has a GC, it means having two different runtimes and two different garbage collectors. For this reason, it is hard to avoid memory leaks when compiling OCaml to Wasm1. Indeed, cycles between the two GCs can not be collected without adding a third GC. it is hard to avoid memory leaks when compiling OCaml to Wasm1, because it means having two different runtimes and two different garbage collectors.

To properly interact with the embedder, we need to leverage the reference and the GC extensions. This extension adds new types to the language, e.g. `externref` which is an opaque type representing a value from the embedder. References cannot be stored in the linear memory of Wasm thus they cannot appear inside OCaml values when using the previously described compilation scheme.

In order to use references, we require a completely different compilation strategy. We do not use the linear memory but rely entirely on WasmGC. Our strategy is close to the native OCaml one, which we describe now.

We use the Flambda IR of the OCaml compiler as input for the Wasm generation. This is a step of the compilation chain where most of the high-level OCaml-specific optimisations are already applied. Also in this IR, the closure conversion pass is already performed. Most of the constructions of this IR maps quite directly to Wasm ones.

Wasocaml is a compiler for the full Flambda IR. We describe the full compilation scheme but for the sake of simplicity we only formalize a subset of Flambda.

3.1 OCaml Value Representation in Wasm

As in native OCaml, we use a uniform representation. We cannot use integers, as they cannot be used as pointers. We need to use a reference type. The most general one is `anyref`. We can be more precise and use `eqref`. This is the type of all values that can be tested for physical equality. It is a super type of OCaml values. This also allows us to test for equality without requiring an additional cast.

Small scalars. All OCaml values that are represented as integers in the native OCaml are represented as `i31ref`. This includes the `int` and `char` types, but also all constant constructors of sum types.

Arrays. The OCaml array type is directly represented as a Wasm array.

Blocks. For other kinds of blocks, there are two possible choices: we can either use a struct with a field for the tag and one field per OCaml value field, or use arrays of `eqref` with the tag stored at position 0.

3.1.1 *Blocks as Structs.*

A block of size one is represented using the type `$block1` while for size two it is `$block2`:

```
(type $block1 (struct
  (field $tag i8)
  (field $field0 eqref)))
```

```
(type $block2 (sub $block1) (struct
  (field $tag i8)
  (field $field0 eqref)
  (field $field1 eqref)))
```

The type `$block2` is declared as a sub-type of `$block1` because in the OCaml IRs, the primitive for accessing a block field is untyped: when accessing the n-th field of the block the only available information is that the block has size at least $n+1$. It could be possible to propagate size metadata in the IRs but that wouldn't be sufficient because of untyped primitives such as `Obj.field` that we need to support in order to be compatible with existing code.

```
(func $snd (param $x eqref) (result eqref)
  (struct.get $block2 1
    (ref.cast $block2 (local.get $x))))
```

3.1.2 Blocks as Arrays. We represent blocks with an array of eqref:

```
(type $block (array eqref))
```

The tag is stored in the cell at index 0 of the array. Reading its value is implemented by getting the cell and casting it to an integer:

```
(func $tag (param $x eqref) (result i32)
  (i31.get_u (ref.cast i31ref
    (array.get $block
      (i32.const 0)
      (ref.cast $block (local.get $x))))))
```

Thus accessing the field n of the OCaml block amounts to accessing the field $n+1$ of the array:

```
(func $snd (param $x eqref) (result eqref)
  (array.get $block
    (i32.const 2)
    (ref.cast $block (local.get $x))))
```

3.1.3 Block Representation Tradeoffs. The array representation is simpler but requires (implicit) bound checks at each field access and a cast to read the tag.

On the other hand, the struct representation requires a more complex cast for the Wasm runtime (a sub-typing test). A compiler propagating more types could use finer Wasm type information, providing a precise type for each struct field. This would allow fewer casts.

OCaml blocks can be arbitrarily large, and very large ones do occur in practice. In particular, modules are compiled as blocks and tend to be on the larger side. We have seen in the wild some examples containing thousands of fields. Wasm only allows a sub-typing chain of length 64, which is far from sufficient for the struct encoding.

For this reason we use the *blocks as arrays* representation.

3.1.4 Boxed Numbers. Raw Wasm scalars such as `i64` are not sub-types of `eqref` thus they cannot be used directly to represent OCaml boxed numbers. We need to box them inside a struct in order to make them compatible with our representation:

```
(type $boxed_float (struct (field $v f64)))
(type $boxed_int64 (struct (field $v i64)))
```

3.1.5 Closures. Wasm `funcref` are functions, not closures, hence we need to produce values containing both the function and its environment. The only Wasm type construction that can contain both `funcref` and other values are structs. Thus, a closure is a struct containing a `funcref` and the captured variables. As an example, here is the type of a closure with two captured variables:

```
(type $closure1 (struct
  (field funcref)
  (field $v1 eqref)
  (field $v2 eqref)))
```

3.2 Control flow

Control flow has a direct equivalent with Wasm `block`, `loop`, `br_table`, and `if` instructions. Low level OCaml primitives to handle exceptions are quite similar to Wasm ones. In OCaml, it is possible to generate new exceptions at runtime by using *e.g.* the `let exception` syntax or functors and first-class modules. This is not possible in the Wasm exception proposal. Thus, we use the same Wasm exception everywhere and manage the rest on the side by ourselves, using an identifier to discriminate between different exceptions.

3.3 Curryfication

In OCaml, functions take only one argument. However, in pratice, functions look like they have more than one. Without any special management this would mean that most of the code would be functions producing closures that would be immediately applied. To handle that, internally, OCaml does have functions taking multiple arguments, with some special handling for times when they are partially applied. This information is explicit at the Flambda level. In the native OCaml compiler, the transformation handling that occurs in a later step called `cmmgen`. Hence, we have to duplicate this in Wasocaml. Compiling this requires some kind of structural sub-typing on closures such that, closures for functions of arity one are supertypes of all the other closures. Thankfully there are easy encodings for that in Wasm.

3.4 Stack Representation

Most of the remaining work revolves around translating from let bound expressions to a stack based language without producing overly naive code. Also, we do not need to care too much about low level Wasm specific optimisations as we rely on Binaryen [16] (a quite efficient Wasm to Wasm optimizer) for those.

3.5 Unboxing

The main optimisation available in native OCaml that we are missing is number unboxing. As OCaml values have a uniform representation, only small scalars can fit in a true OCaml value. This means that the types `nativeint`, `int32`, `int64` and `float` have to be boxed. In numerical code, lots of intermediate values of type float are created, and in that

case, the allocation time to box numbers completely dominates the actual computation time. To limit that problem, there is an optimisation called unboxing performed during the cmmgen pass that tends to eliminate most of the useless allocations. As this pass is performed after Flambda, and was not required to produce a complete working compiler, this was left for future work. Note that the end plan is to use the next version of Flambda, which does a much better job at unboxing. For the time being, we can expect Binaryen to perform local unboxing in some cases.

4 Formalized compilation scheme

When compiling a program, we start with the following prelude:

```
(module $wasocaml_generated_modul

  (type $mlfun (sub final
    (func (param eqref) (param eqref) (result eqref))))

  (type $data (sub final
    (array (mut (ref null eq)))))

  (type $block (sub final (struct
    (field i32)
    (field (ref null $data)))))

  (type $vars (sub final (array (mut eqref))))

  (rec
    (type $set_of_closures (sub final (struct
      (field (mut (ref null $closures)))
      (field (ref null $vars)))))

    (type $closures (sub final
      (array (mut (ref $closure)))))

    (type $closure (sub final (struct
      (field (ref $mlfun))
      (field (mut (ref null $set_of_closures)))))))

  (global $tmp_set_of_closures
    (mut (ref null $set_of_closures))
    ref.null $set_of_closures)
  (global $switch_v
    (mut i32) i32.const 0)
```

```
(elem declare (ref $mlfun)
  ;; to allow forward references of functions
  ;; ...
)

(func $start
  ;; ...
  drop
)

(start $start))
```

To compile a term we define a function $\mathscr{T}$ from Flambda terms to Wasm expressions. We define it case by case. Here we do not only use the S-expression notation for Wasm but also use the stack notation which makes these rules clearer.

$$\mathscr{T}(n) = \texttt{i32.const}\ n$$
$$\texttt{i31.new}$$

An integer literal is turned into an `i31ref` in order to match with the uniform representation.

$$\mathscr{T}(x) = \texttt{local.get}\ \$x$$

To access the value of an identifier, we simply access the value stored in the local variable of the current function.

$$\mathscr{T}(\texttt{let}\ x = t_1\ \texttt{in}\ t_2) = \mathscr{T}(t_1)$$
$$\texttt{local.set}\ \$x$$
$$\mathscr{T}(t_2)$$

A let-binding is compiled by evaluating t_1, storing its result in the local x and evaluating t_2.

$$\mathscr{T}(x_1\ x_2) = \texttt{local.get}\ \$x_1$$
$$\texttt{ref.cast}\ (\texttt{ref}\ \$\texttt{closure})$$
$$\texttt{local.get}\ \$x_2$$
$$\texttt{ref.as\_non\_null}$$
$$\texttt{local.get}\ \$x_1$$
$$\texttt{ref.cast}\ (\texttt{ref}\ \$\texttt{closure})$$
$$\texttt{struct.get}\ \$\texttt{closure}\ 0$$
$$\texttt{call\_ref}\ \$\texttt{mlfun}$$

To apply a function x_1 to a value x_2 we first put the two parameters of the function on the stack. The first one is the closure itself, which is needed to access the environment and other functions in the set of closures. The second one can be any value of type `eqref`. Then we need to put the function of the stack. This is done by accessing the field 0 of the closure, which is a **`funcref`**. Then we use `call_ref` to call the function with its two arguments. The type of the function reference is always `$mlfun`.

$$\mathscr{T}(x_1 \texttt{<-}\ x_2) = \texttt{local.get}\ \$x_2$$
$$\texttt{local.set}\ \$x_1$$
$$\texttt{i32.const}\ 0$$
$$\texttt{i31.new}$$

A mutation simply puts the content of x_2 into x_1 and leaves 0 on the stack (this is `() : unit` in OCaml).

$$\mathscr{T}(\texttt{if}\ x_1\ \texttt{then}\ t_1\ \texttt{else}\ t_2) = \texttt{local.get}\ \$x_1$$
$$\texttt{ref.cast}\ \texttt{i31ref}$$
$$\texttt{i31.get\_s}$$
$$(\textbf{if}\ (\textbf{result}\ (\texttt{ref}\ \texttt{null}\ \texttt{eq}))$$
$$(\textbf{then}\ \mathscr{T}(t_1))$$
$$(\textbf{else}\ \mathscr{T}(t_2)))$$
$$\texttt{ref.as\_non\_null}$$

A conditional converts the value of x_1 to an **`i32`** and then uses a Wasm **`if`**. The two branches are simply the result of compiling t_1 and t_2.

$\mathscr{T}(\texttt{switch}\ x\ \mathscr{B}_1\ \mathscr{B}_2) =$

```
(block $switch_a (result (ref null eq))
    (block $switch_b (result (ref null eq))
        local.get $x
        br_on_cast $switch_b (ref null eq) i31ref
        ;; handle block case
        ref.cast (ref $block)
        struct.get $block 0
        global.set $switch_v
        𝒯(ℬ2)
        ;; skip the scalar case
        br $switch_a)
    ;; handle scalar case
    ref.cast (ref i31)
    i31.get_s
    global.set $switch_v
    ℱ(ℬ1))
```

To compile a switch, we start by testing if x is a block or a scalar value. If it's a block, we store its tag, otherwise we store the value directly. Then, we use an auxiliary function $\mathscr{F}$ in order to compile the various cases of each branches. Let $i_0, \ldots, i_{n-1} \in \mathrm{dom}(\mathscr{B})$ be such that $\nexists j, j \notin \mathrm{dom}(\mathscr{B}) \wedge \forall k \in [0; n[, j \neq i_k$. Then we define the auxiliary function as follows:

$\mathscr{F}(\mathscr{B}) =$

```
(i32.eq (i32.const in−1) (global.get $switch_v))
(if (result (ref null eq)) (then 𝒯(ℬ(in−1)))
(else
    ;; ...
    (i32.eq (i32.const i0) (global.get $switch_v))
    (if (result (ref null eq)) (then 𝒯(ℬ(i0)))
    (else unreachable))))
```

We have a sequence of nested **if** to find which of the binding in $\mathscr{B}$ we should evaluate. If none of them correspond to the value stored in `$switch_v` (which has been set by the code generated when compiling the switch) then we have to put an `unreachable` in order to generate Wasm code that typechecks. In pratice, the `unreachable` corresponds to a match failure, which should not happen in a valid source program, thus it is indeed unreachable.

$$\begin{aligned}\mathscr{T}(\texttt{sraise}\ x_{exn}\ x^n) = {}& \texttt{local.get}\ \$x_{n-1}\\ & \ldots\\ & \texttt{local.get}\ \$x_0\\ & \texttt{br}\ \$\texttt{exn\_}x_{exn}\end{aligned}$$

A static raise puts all the value transported by the exception on the stack and branches to the block corresponding to the raise.

$$\begin{aligned}&\mathscr{T}(\texttt{scatch}\ t_1\ \texttt{with}\ x_{exn}\ x^n\ t_2) =\\ &\texttt{(block \$after\_try\_catch (result (ref null eq))}\\ &\quad\texttt{(block \$exn\_}x_{exn}\\ &\qquad\texttt{(result (ref null eq)) ;; 0}\\ &\qquad\texttt{(result (ref null eq)) ;; ...}\\ &\qquad\texttt{(result (ref null eq)) ;; n - 1}\\ &\qquad\mathscr{T}(t_1)\\ &\qquad\texttt{br \$after\_try\_catch)}\\ &\quad\texttt{local.set \$}x_0\\ &\quad\ldots\\ &\quad\texttt{local.set \$}x_{n-1}\\ &\quad\mathscr{T}(t_2)\ \texttt{)}\end{aligned}$$

A catch has two nested blocks. The first one is used in case no exception is raised and directly returns the value of t_1. The second one corresponds to the case where an exception is raised. It needs to have many results corresponding to all the values transported by the exception. It will put all theses values in the right locals.

$$\mathscr{T}(\texttt{while}_{t_1}\ t_1\ \texttt{do}\ t_2) = \texttt{(block \$exit\_while\_loop}$$
$$\texttt{(loop \$while\_loop}$$
$$\mathscr{T}(t_1)$$
$$\texttt{i31.get\_s}$$
$$\texttt{i32.eqz}$$
$$\texttt{br\_if \$exit\_while\_loop}$$
$$\mathscr{T}(t_2)$$
$$\texttt{drop}$$
$$\texttt{br \$while\_loop))}$$
$$\texttt{i32.const 0}$$
$$\texttt{i31.new}$$

A while loop maps quite directly to a Wasm **loop**. We also need an other block in order to exit the loop when the condition is false.

$$\mathscr{T}(\texttt{get\_field}\ n\ x) = \texttt{local.get \$}x$$
$$\texttt{ref.cast (ref \$block)}$$
$$\texttt{struct.get \$block 1}$$
$$\texttt{ref.as\_non\_null}$$
$$\texttt{i32.const}\ n$$
$$\texttt{array.get \$data}$$

To read a field of a block, we need to cast it to a block, access its second field which contains the values of the block (the first field being the tag). Then we simply read the element at the index n.

$$\begin{aligned}\mathscr{T}(\texttt{set\_field}\ n\ x_1\ x_2) = {}& \texttt{local.get}\ \$x_1\\ & \texttt{ref.cast}\ (\texttt{ref}\ \$\texttt{block})\\ & \texttt{struct.get}\ \$\texttt{block}\ 1\\ & \texttt{ref.cast}\ (\texttt{ref}\ \$\texttt{data})\\ & \texttt{i32.const}\ x\\ & \texttt{local.get}\ \$x_2\\ & \texttt{array.set}\ \$\texttt{data}\\ & \texttt{i32.const}\ 0\\ & \texttt{i31.new}\end{aligned}$$

Similarly, to set a field of a block, we cast it, access its data field and then set the element at index n to the value of x_2. The array is mutable and there is no need to update the structure after the element has been set.

$$\begin{aligned}\mathscr{T}(\texttt{make\_block}\ n\ x_0, \ldots, x_{m-1}) = {}& \texttt{i32.const}\ n\\ & \texttt{local.get}\ \$x_0\\ & \ldots\\ & \texttt{local.get}\ \$x_{m-1}\\ & \texttt{array.new\_fixed}\ \$\texttt{data}\ m\\ & \texttt{struct.new}\ \$\texttt{block}\end{aligned}$$

To create a new block, we put its tag on the stack. Then we put all its data and make a new array from it. With the tag and the array we then make a new structure of type $block.

$$\begin{aligned}&\mathscr{T}(\texttt{project\_closure}\ x_1\ x_2) =\\ &\texttt{local.get}\ \$x_2\\ &\texttt{ref.cast}\ (\texttt{ref}\ \$\texttt{set\_of\_closures})\\ &\texttt{struct.get}\ \$\texttt{set\_of\_closures}\ 0\\ &\texttt{global.get}\ \$\texttt{closure\_offset\_within\_set\_}x_1\\ &\texttt{array.get}\ \$\texttt{closures}\end{aligned}$$

To read a closure from a set of closures, we cast x_2 to a set. Then we access its first field, which is an array of closures. Then we look for the offset of x_1 and access it.

$\mathscr{T}$(project_var x_1 x_2) =

```
local.get $x2
ref.cast (ref $closure)
struct.get $closure 1
ref.as_non_null
struct.get $set_of_closures 1
global.get $var_offset_within_set_x1
array.get $vars
```

This is similar to the closure case but the difference here is that x_2 is a closure and not a set. We need to access its parent first. Then we access the second field, which is the array of the captured variables. Then we need to look for the offset of the variable again before reading the array.

$\mathscr{T}$(move_within_closure x_1 x_2) =

```
local.get $x2
ref.cast (ref $closure)
struct.get $closure 1
ref.as_non_null
struct.get $set_of_closures 0
global.get $closure_offset_within_set_x1
array.get $closures
```

This is the same as the previous case: x_2 is a closure and not a set. The only difference being that we look in the first field instead of the second one.

For allocating a set of closures $\mathscr{S} = \mathscr{V}, \mathscr{C}$ we define $x_0, ..., x_{n-1}$ to be the values that are captured in $\mathscr{V}$ and $f_0, ..., f_{m-1}$ to be the names of the functions that are part of the set.

$\mathscr{T}$(make_set_of_closures $\mathscr{S}$) =

```
;; empty closures
ref.null $closures
;; generating variables
(ref.as_non_null (local.get $x_0))
;; ...
(ref.as_non_null (local.get $x_{n-1}))
array.new_fixed $vars n
;; set of closures with empty closures and filled vars
struct.new $set_of_closures
global.set $tmp_set_of_closures
global.get $tmp_set_of_closures
;; generating closures
ref.func $f_0
global.get $tmp_set_of_closures
struct.new $closure
;; ...
ref.func $f_{m-1}
global.get $tmp_set_of_closures
struct.new $closure
array.new_fixed $closures m
;; patching the set with closures
struct.set $set_of_closures 0
;; returning the set
global.get $tmp_set_of_closures
```

We start by generating an empty array of closures. We build the array made of the captured variables. Then we generate an empty set of closures from our empty array of closures and our array of variables.. Each closure needs a reference to its set, hence we have to store the set in a global before constructing them. Then, we build the array made of the various functions in the set and patch the set.

The code of the compiled Flambda program is inserted as the code of the `$start` function. But we also need to analyse the term in order to do a few more things. Each function defined in a set of closures is also added as a new Wasm function and marked as a declarative element (in order to be referenced by the `ref.func` instruction). Moreover, each function and each variable in a set of closures has its own offset stored as a global variable (for instance `global $closure_offset_within_set_myfunc`). Finally, each function is analysed in order to find which locals it requires to be declared.

For now, we don't have a proof of the correcteness of our compilation scheme. The formal semantics of the various Wasm extensions we are needing is still an ongoing work by the various working groups. We would like to use these semantics in the future.

4.1 Example of generated code

The code of `cl_iter_f` from the example given in 2.4 would be compiled to the following Wasm code:

```
(func $cl_iter_f (param $envtwo eqref) (param $l eqref)
(result eqref) (local $x eqref) (local $f eqref)
(local $dummy eqref) (local $tl eqref)
  (block $switch_a  (result (ref null eq))
    (block $switch_b  (result (ref null eq))
      local.get $l
      br_on_cast $switch_b (ref null eq) i31ref
      ref.cast (ref  $block)
      struct.get $block 0
      global.set $switch_v
      global.get $switch_v
      i32.const 0
      i32.eq
      (if (result (ref null eq))
        (then
          local.get $l
          ref.cast (ref  $block)
          struct.get $block 1
          ;; ...
          local.get $envtwo
          ref.cast (ref  $closure)
          struct.get $closure 0
          call_ref $mlfun)
        (else unreachable))
      br $switch_a)
    ref.cast (ref  i31)
    i31.get_s
    global.set $switch_v
```

```
    global.get $switch_v
    i32.const 0
    i32.eq
    (if (result (ref null eq))
      (then
        i32.const 0
        i31.new)
      (else
        unreachable))))
```

5 Performances

We are able to run the classical functional microbenchmarks using the experimental branch of V8 and the results are quite convincing. While we do not expect to be competitive with the native code compiler, the performance degradation is almost constant (around twice times slower).

We are also able to compile an OCaml implementation of the Knuth-Bendix algorithm [5]. On V8 the Wasm exception handling proposal is really slow compared to native OCaml: a raise is around a hundred times slower. In SpiderMonkey (the Firefox engine) they are much faster but other proposals are missing and we can't run code we generate. We have to discuss with the V8 team if this is expected and if this can be improved. We implemented an alternative compilation strategy for exceptions where we do not use Wams exceptions. Instead we make every function return a pair containing its return value along with a boolean indicating if an exception was raised or not. When using this alternative strategy, we are around two times slower than native OCaml.

The V8 runtime allows to consider casts as no-ops (only for test purposes). The speedup we measured is around 10%. That gives us an upper bound on the actual runtime cost of casts.

We are currently working on producing benchmarks on real sized programs, it is not easy as Wasocaml is still a prototype and is not yet integrated into existing build systems.

Compared to a JavaScript VM, a Wasm compiler is a much simpler beast that can compile ahead of time. For this reason, various Wasm engines are expected to behave quite similarly. They do not show any of the wild impredictability that browsers tend to demonstrate with JavaScript. Indeed, compiling OCaml to JS using js_of_ocaml leads to results that are usually also twice as slow as native code in the best cases, but can sometimes be much slower in an unpredictable fashion.

Currently there is no other Wasm runtime supporting all the extensions we require. SpiderMonkey does not have tail calls. The reference interpreter implementation of the various extensions are split in separate repositories and merging them requires some work.

6 Perspectives

As the first version of the implementation was intended as a demonstration, it remains a bit rough around the edges. In particular only a fraction of the externals from the standard library externals are provided as handwritten Wasm. The only unsupported part of the language is the object-oriented fragment, by lack of time rather than due to any specific complexity: in Flambda this fragment is reduced to a single primitive. The source code of Wasocaml is publicly available [1].

6.1 FFI

A lot of OCaml code in the wild interacts with C or JavaScript code through bindings written using dedicated FFI mechanisms. We plan to allow re-use of existing bindings when compiling to Wasm. When targeting the browser through JavaScript, it is not possible to re-use C bindings: people either have to rewrite their code in pure OCaml, or to use JavaScript bindings. Compiling C bindings directly to Wasm allows to re-use them as-is.

C bindings. Some extensions recently added to Clang [6] allow to compile C code to Wasm in a way that makes reusing existing OCaml bindings to C code possible with almost no change. We would only have to provide a modified version of the OCaml native FFI headers files, replacing the usual macros with hand written Wasm functions. The only limitation that we forsee is that the `Field` macro won't be an l-value anymore; a new `Set_field` macro will be needed instead.

JavaScript bindings. To re-use existing bindings of OCaml to JavaScript code, we need one more extension: the reference-typed strings proposal [24]. Almost all external calls in the JavaScript FFI go through the `Js.Unsafe.meth_call` function which has the type `'a -> string -> any array -> 'b`. It can be exposed to the Wasm module by the embedder as a function of type:

```
(func $meth_call
  (param $obj externref)
  (param $method stringref)
  (param $args $anyarray)
  (result externref))
```

This calls a method of name `$method` on the object `$obj` with the arguments `$args`. The JavaScript side manages all the dynamic typing.

6.2 Effect Handlers Support

Our compiler is based on OCaml 4.14. The OCaml 5 compiler introduces effect handlers [10], a mechanism that can be seen as a generalisation of exceptions, allowing them to be resumed. They were out of scope for our compiler, yet we describe three strategies to handle them in the future.

CPS Compilation. It is possible to represent effect handlers as continuations, using whole-program CPS transformation [4]. However it is not ideal for two reasons. It requires the program to contain OCaml code only. Moreover, it prevents the use of the stack of the target language and instead stores that stack in the heap, with a non-negligible cost. It can be mitigated by only applying the transformation to code that cannot be proved to not use effect handlers but then it is no longer compatible with separate compilation and the optimisation needs to be done as a whole program transformation. This is the choice made by Js_of_ocaml.

Wasm Stack Switching. There is an ongoing proposal, called stack switching [23], that exactly matches the needs for OCaml effect handlers. With it the translation would be quite straightforward.

JavaScript Promise Integration. Another strategy is available for runtimes providing both Wasm and JavaScript. There is another ongoing proposal called JavaScript promise integration [22]. With this proposal, effects handlers

can be implemented with a JavaScript device. This proposal is likely to land before the proper stack switching one. It might be a temporary solution.

7 Related Work

7.1 Garbage Collected Languages to Wasm

7.1.1 Targeting Wasm1.

Go. Go compiles to Wasm1 [9]. In order to be able to re-use the Go GC, it must be able to inspect the stack. This is not possible within Wasm. Thus it has to maintain the stack by itself using the memory, which is much slower than using the Wasm stack.

Haskell. Haskell also compiles to Wasm [11]. It has constraints similar to Go and uses similar solutions. In particular, it also uses the memory to manage the stack.

7.1.2 Targeting Wasm-GC.

Dart. Dart [13] compiles to Wasm-GC. It does not uses `i31ref` and boxes scalars in a `struct`.

Scheme. The last addition to the small family of compiler targeting Wasm-GC is the Guile Scheme compiler. Scheme has many similar constraints as OCaml and Guile uses many similar solutions developed independently of Wasocaml. The compiler was presented to the Wasm-GC working group [26]. A more detailed explanation is published [25]. The implementation and its in-depth technical description are available [27].

7.2 OCaml Web Compilers

The history of OCaml compilers targeting web languages is quite crowded. This is surely thanks to the great pleasure that is writing compilers in OCaml.

7.2.1 Targeting JavaScript. There are multiple OCaml compilers targeting JavaScript. Many approaches where experimented, the main two live ones are js_of_ocaml and melange. Naive compilation of OCaml to JavaScript is quite simple as it can almost be summarized as "type erasure". There are some limitations in JavaScript that prevent that to be a complete compilation strategy. Moreover, getting to a proper compiler producing efficient and small JavaScript code is more complex.

Jsoo. Jsoo [15] tries to be as close as possible to the native semantics. It compiles OCaml bytecode programs to JavaScript by decompiling it back to a simple untyped lambda calculus then performs many optimisation and minimisation passes.

Melange. Melange [8] tries to produce readable JavaScript, with a closer integration in the JavaScript module system. Melange starts from a modified version of the lambda IR which provides more type information. This allows the use of JavaScript features that match the uses of source features at the cost of some small semantical differences.

7.2.2 Targeting Wasm.

OCamlrun WebAssembly. OCamlrun WebAssembly [7] compiles the OCaml runtime and interpreter to Wasm. They are both written in C so it is possible without any Wasm extension. In some cases, it allows to start executing code faster

than when compilin OCaml to Wasm. This is because there is less WebAssembly code to compile for the embedder. On the other hand, the execution is slower as it is interpreting OCaml bytecode inside Wasm instead of running a compiled version.

WASICaml. WASICaml [12] is quite similar to OCamlrun WebAssembly, the main difference being that WASICaml does not interpret the bytecode directly but translates it partially to Wasm. It leads to a faster execution.

These approaches suffer from the same problem as C programs running on Wasm: they cannot natively manipulate external values, such as DOM objects in the browser. Indeed, in the first versions of Wasm, the only types are scalars (`i32`, `i64`, `f32` and `f64`). There is no way to directly manipulate values from the embedder (*e.g.* JavaScript objects). It could be possible to identify objects with an integer and map them to their corresponding objects on the embedder side. This approach comes with the usual limitations of having two runtimes manipulating (indirectly) garbage-collected values: it is tedious and easily leads to memory leaks (cycles between the two runtimes cannot be collected).

Wasm_of_ocaml. Wasm_of_ocaml [14] is a fork of Js_of_ocaml. It compiles the bytecode to Wasm-GC. It was developed after Wasocaml and is based on the techniques we developed and that are described in the article. It does not benefit from the optimisations provided by Flambda. It is also quite restricted in its value representation choice and must use a representation based on arrays.

8 Conclusion

The original goal of this implementation was to validate the Wasm-GC proposal and discuss its limits. This part was successful. No major changes were required for OCaml (mainly keeping the `i31ref` type and changes to the maximal length of sub-typing chains). And we are quite happy to claim that Wasm-GC is, to our opinion a very good design, that is a perfect compilation target for a garbage collected functional language like OCaml. And we think that the experience for most other garbage-collected languages will probably be similar.

As a side-effect, we now have an OCaml to Wasm-GC compiler. It is not yet usable because there are no user-side Wasm-GC runtime available. In order to test our compiler we are using V8 with various flags to enable experimental features such as the GC support. The design of the various extensions required by Wasocaml is stable and quite close to completion, but some details are still in flux. For this reason, we cannot expect them to be widely available soon. On the other hand, it means that our compiler will be ready when browsers start deploying new Wasm extensions.

Our future plans are to complete the Wasm implementations of OCaml externals, to implement the various FFI mechanisms, support effect handlers and to move Wasocaml to Flambda2. All of these will allow easy deployment of multi-language software on the browser while having good and predictable performances.

References

[1] Léo Andrès and Pierre Chambart. 2022. *Wasocaml.* https://github.com/ocamlpro/wasocaml
[2] Léo Andrès and Pierre Chambart. 2023. OCaml on WasmGC. https://github.com/WebAssembly/meetings/blob/main/gc/2023/GC-01-10.md.
[3] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 185–200.
[4] Daniel Hillerström, Sam Lindley, Robert Atkey, and KC Sivaramakrishnan. 2017. Continuation passing style for effect handlers. (2017).
[5] Donald E Knuth and Peter B Bendix. 1983. Simple word problems in universal algebras. *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970* (1983), 342–376.
[6] Chris Lattner. 2008. LLVM and Clang: Next generation compiler technology. In *The BSD conference*, Vol. 5. 1–20.
[7] Sebastian Markbåge. 2017. *OCamlrun WebAssembly.* https://github.com/sebmarkbage/ocamlrun-wasm
[8] Antonio Nuno Monteiro. 2022. *Melange.* https://github.com/melange-re/melange

[9] Richard Musiol. 2018. WebAssembly architecture for Go. https://docs.google.com/document/d/131vjr4DH6JFnb-blm_uRdaC0_Nv3OUwjEY5qVCxCup4.
[10] Gordon D Plotkin and Matija Pretnar. 2013. Handling algebraic effects. *Logical methods in computer science* 9 (2013).
[11] Cheng Shao. 2022. WebAssembly backend merged into GHC. https://www.tweag.io/blog/2022-11-22-wasm-backend-merged-in-ghc.
[12] Gerd Stolpmann. 2021. *WASICaml.* https://github.com/remixlabs/wasicaml
[13] The Dart Project Authors. 2022. *dart2wasm.* https://github.com/dart-lang/sdk/tree/main/pkg/dart2wasm
[14] Jérôme Vouillon. 2023. *Wasm_of_ocaml.* https://github.com/ocaml-wasm/wasm_of_ocaml
[15] Jérôme Vouillon and Vincent Balat. 2014. From bytecode to JavaScript: the Js_of_ocaml compiler. *Software: Practice and Experience* 44, 8 (2014), 951–972.
[16] WebAssembly Community Group participants. 2015. *Binaryen.* https://github.com/WebAssembly/binaryen
[17] WebAssembly Community Group participants. 2017. *Exception Handling Proposal for WebAssembly.* https://github.com/WebAssembly/exception-handling
[18] WebAssembly Community Group participants. 2017. *GC Proposal for WebAssembly.* https://github.com/WebAssembly/gc
[19] WebAssembly Community Group participants. 2017. *Tail Call Proposal for WebAssembly.* https://github.com/WebAssembly/tail-call
[20] WebAssembly Community Group participants. 2018. *Reference Types Proposal for WebAssembly.* https://github.com/WebAssembly/reference-types
[21] WebAssembly Community Group participants. 2020. *Typed Function References Proposal for WebAssembly.* https://github.com/WebAssembly/function-references
[22] WebAssembly Community Group participants. 2021. *JavaScript-Promise Integration Proposal for WebAssembly.* https://github.com/WebAssembly/js-promise-integration
[23] WebAssembly Community Group participants. 2021. *Stack Switching Proposal for WebAssembly.* https://github.com/WebAssembly/stack-switching
[24] WebAssembly Community Group participants. 2022. *Reference-Typed Strings Proposal for WebAssembly.* https://github.com/WebAssembly/stringref
[25] Andy Wingo. 2023. a world to win: webassembly for the rest of us. https://www.wingolog.org/archives/2023/03/20/a-world-to-win-webassembly-for-the-rest-of-us.
[26] Andy Wingo. 2023. Compiling Scheme to WasmGC. https://github.com/WebAssembly/meetings/blob/main/gc/2023/GC-04-18.md.
[27] Andy Wingo. 2023. *Hoot.* https://gitlab.com/spritely/guile-hoot/-/blob/main/design/ABI.md