Wasocaml: compiling OCaml to WebAssembly
Léo Andrès, Pierre Chambart, Jean-Christophe Filliâtre

To cite this version:
Léo Andrès, Pierre Chambart, Jean-Christophe Filliâtre. Wasocaml: compiling OCaml to WebAssembly. IFL 2023 - The 35th Symposium on Implementation and Application of Functional Languages, João Saraiva; João Fernandes, Aug 2023, Braga, Portugal. hal-04311345v2

HAL Id: hal-04311345
https://inria.hal.science/hal-04311345v2
Submitted on 13 Dec 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License
Wasocaml: compiling OCaml to WebAssembly

Léo Andrès*  
leo@ocamlpro.com  
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

ABSTRACT
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 FFI’s 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.

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 cater to the specific needs of other languages and uses. This version of Wasm can be seen as simplified C. Here is an example:

```wasm
(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:

```wasm
(func $f1 (func $f2 (result i32)))
(func $t1 (array i32))
(func $t2 (struct
  (field $a (ref $t1))
  (field $b (ref $f1))
  (field $c i32)))))
```

ACM Reference Format:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

IFL’23, August 29-August 31, 2023, Braga, Minho, Portugal
© 2023 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/XXXXXXX-XXXXXX
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.

2.2 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:

```ocaml
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:
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 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 $a$.

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

$$| a \quad \text{(address)}$$

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 $\mathcal{M}$ 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 $o^*$, a closure $\overline{x}$, or a set of closures $\mathcal{F}$. A closure is made of a set of closures $\mathcal{F}$ and of a name $x$ which is its identifier inside the set.

$$\mathcal{M} ::= a \mapsto n, 0^* \quad \text{(block)}$$

$$| \overline{x} \quad \text{(closure)}$$

$$| \mathcal{F} \quad \text{(set of closure)}$$

Set of closures. A set of closures $\mathcal{F}$ represents a set of mutually recursive functions.

$$\mathcal{F} ::= \mathcal{F'}, \mathcal{C}$$

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

$$\mathcal{F} ::= x \mapsto v$$

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

$$\mathcal{C} ::= x \mapsto \lambda x. x \cdot t$$

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).

2.3.2 Small-step semantics.

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

Reduction context.

$$E ::= \square$$

$$| \text{let } x = E \text{ in } t$$

$$| \text{while } E \text{ do } t$$

$$| \text{scatch } E \text{ with } x \cdot x^* t$$

$$| \text{pop } E$$

Reduction inside a context.

$$\frac{}{\mathcal{C}, \mathcal{M}, E(t_1) \mapsto \mathcal{C'}, \mathcal{M'}, E(t_2)}$$

Binary operator.

$$op ::= \text{eq} \quad \text{(equality)}$$

$$| \text{add} \quad \text{(addition)}$$

$$| \text{sub} \quad \text{(subtraction)}$$

Branches.

$$t ::= n \mapsto t \quad \text{(value)}$$

$$| x \quad \text{(identifier)}$$

$$| \text{let } x \equiv t \text{ in } t$$

$$| x \cdot x$$

$$| \text{if } x \text{ then } t \text{ else } t$$

$$| \text{switch } x \cdot \mathcal{R} \cdot \text{ (switch)}$$

$$| \text{sraise } x \cdot x^*$$

$$| \text{scatch } t \text{ with } x \cdot x^* t$$

$$| \text{while } t \text{ do } t$$

$$| \text{get_field } n \cdot x$$

$$| \text{set_field } n \cdot x$$

$$| \text{make_block } n \cdot x^*$$

$$| \text{project_closure } x \cdot x$$

$$| \text{project_var } x \cdot x$$

$$| \text{move_within_closure } x \cdot x$$

$$| \text{make_set_of_closures } \mathcal{F}$$

$$| \text{pop } t$$

$$| x \cdot \text{op } x$$

Note that the pop instruction is an administrative instruction that is never present in the concrete syntax coming from user code.
Head reductions. In order to reduce cluttering, $V^*$ and $M$ 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.

<table>
<thead>
<tr>
<th>Reduction</th>
<th>$\mathcal{Y}_0(x) = v$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifier</td>
<td>$\mathcal{Y}_0[V^*, x] \rightarrow v$</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Reduction</th>
<th>$\mathcal{Y}_0(x) = v$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Let-binding</td>
<td>$\mathcal{Y}_0[V^<em>, \text{let } x = v \text{ in } t] \rightarrow \mathcal{Y}_0[V^</em>, \text{let } x = v \text{ in } t]$</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Reduction</th>
<th>$\mathcal{Y}_0(x) = v$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>$\mathcal{Y}_0[V^<em>, \text{let } x = v \text{ in } t] \rightarrow \mathcal{Y}_0[V^</em>, \text{let } x = v \text{ in } t]$</td>
</tr>
</tbody>
</table>

An application $\mathcal{Y}_0[V^*, \text{let } x = v \text{ in } t]$ reduces to $t$ and we create a new environment containing two bindings: $x$ is mapped to the value of $x$ in the current environment and $x$ is mapped to $a$.

<table>
<thead>
<tr>
<th>Reduction</th>
<th>$\mathcal{Y}_0(x) = v$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pop</td>
<td>$\mathcal{Y}_0[V^<em>, \text{pop } v \text{ in } t] \rightarrow \mathcal{Y}_0[V^</em>, v]$</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Reduction</th>
<th>$\mathcal{Y}_0(x) = v$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutate</td>
<td>$\mathcal{Y}_0[V^<em>, x_1 \leftarrow x_2 \rightarrow \mathcal{Y}_0[V^</em>, x_1 \leftarrow v_2 \mathcal{Y}^<em>, 0] \rightarrow \mathcal{Y}_0[V^</em>, x_1 \leftarrow v_2 \mathcal{Y}^*, 0]$</td>
</tr>
</tbody>
</table>

A mutation $x_1 \leftarrow x_2$ 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 False in OCaml.

<table>
<thead>
<tr>
<th>Reduction</th>
<th>$\mathcal{Y}_0(x) = n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conditional 1</td>
<td>$\mathcal{Y}_0[V^*, \text{if } x \text{ then } t_1 \text{ else } t_2] \rightarrow t_1$</td>
</tr>
</tbody>
</table>

A conditional $\text{if } x \text{ then } t_1 \text{ else } t_2$ 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).

<table>
<thead>
<tr>
<th>Reduction</th>
<th>$\mathcal{Y}_0(x) = n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conditional 2</td>
<td>$\mathcal{Y}_0[V^*, \text{if } x \text{ then } t_1 \text{ else } t_2] \rightarrow t_2$</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Reduction</th>
<th>$\mathcal{Y}_0(x) = n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Switch 1</td>
<td>$\mathcal{Y}_0[V^*, \text{switch } x \text{ to } P_1 \text{ to } P_2] \rightarrow P_1$</td>
</tr>
</tbody>
</table>

When the value of $x$ in the current environment is an integer $n$, then $\text{switch } x \text{ to } P_1 \text{ to } P_2$ reduces to the image of $n$ in $P_1$.

<table>
<thead>
<tr>
<th>Reduction</th>
<th>$\mathcal{Y}_0(x) = a$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Switch 2</td>
<td>$\mathcal{Y}_0[V^*, \text{switch } x \text{ to } P_1 \text{ to } P_2] \rightarrow P_1$</td>
</tr>
</tbody>
</table>

Similarly, when $x$ is an address $a$ and $M(a)$ is a block with tag $n$, then $\text{switch } x \text{ to } P_1 \text{ to } P_2$ reduces to the image of $n$ in $P_2$.

<table>
<thead>
<tr>
<th>Reduction</th>
<th>$\text{raise } x_{\text{exn}} \neq x'_{\text{exn}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Switch 1</td>
<td>$\text{scatch (sraise } x_{\text{exn}} x^n \text{ with } x'<em>{\text{exn}} x'^n t] \rightarrow \text{sraise } x</em>{\text{exn}} x^n$</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Reduction</th>
<th>$\text{raise } x_{\text{exn}} = x'_{\text{exn}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Switch 1</td>
<td>$\text{scatch (sraise } x_{\text{exn}} x^n \text{ with } x'<em>{\text{exn}} x'^n t] \rightarrow \text{sraise } x</em>{\text{exn}} x^n$</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Reduction</th>
<th>$\text{raise } x_{\text{exn}} = x'_{\text{exn}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Switch 1</td>
<td>$\text{while } x_{\text{exn}} x^n \text{ do } t \rightarrow \text{sraise } x_{\text{exn}} x^n$</td>
</tr>
</tbody>
</table>

A while simply re-raises static exceptions.

<table>
<thead>
<tr>
<th>Reduction</th>
<th>$\text{raise } x_{\text{exn}} = x'_{\text{exn}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Switch 1</td>
<td>$\text{while } x_{\text{exn}} x^n \text{ do } t \rightarrow \text{sraise } x_{\text{exn}} x^n$</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Reduction</th>
<th>$\text{while } x_{\text{exn}} x^n \text{ do } t \rightarrow \text{sraise } x_{\text{exn}} x^n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Switch 1</td>
<td>$\text{while } x_{\text{exn}} x^n \text{ do } t \rightarrow \text{sraise } x_{\text{exn}} x^n$</td>
</tr>
</tbody>
</table>

A loop while $x_{\text{exn}} \neq 0$ reduces to $0$ when $x_{\text{exn}} = 0$. 

<table>
<thead>
<tr>
<th>Reduction</th>
<th>$\text{while } x_{\text{exn}} x^n \text{ do } t \rightarrow \text{sraise } x_{\text{exn}} x^n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Switch 1</td>
<td>$\text{while } x_{\text{exn}} x^n \text{ do } t \rightarrow \text{sraise } x_{\text{exn}} x^n$</td>
</tr>
</tbody>
</table>

A loop while $x_{\text{exn}} \neq 0$ reduces to $0$ when $x_{\text{exn}} = 0$. 

<table>
<thead>
<tr>
<th>Reduction</th>
<th>$\text{while } x_{\text{exn}} x^n \text{ do } t \rightarrow \text{sraise } x_{\text{exn}} x^n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Switch 1</td>
<td>$\text{while } x_{\text{exn}} x^n \text{ do } t \rightarrow \text{sraise } x_{\text{exn}} x^n$</td>
</tr>
</tbody>
</table>
On the contrary, when \( n \neq 0 \), it reduces to \( \text{let } _= t \text{ in } \text{while}_{p_r} \ t \text{ do } t \).

The term \( t \) corresponds to the original term that was used as a condition.

\[
\mathcal{V}_0(x) = a \\
\mathcal{M}(a) = \text{tag, } v^* \\
v^* = v_0, \ldots, v_m, \quad m > n \geq 0
\]

The term \( \text{get_field } n \times x \) reduces to the \( n \)-th value of the block stored at address \( a \) with \( a \) being the value of \( x \) in the current environment.

\[
\mathcal{V}_0(x_0) = v_0, \ldots, \mathcal{V}_0(x_{n-1}) = v_{n-1} \\
\text{set_field } n \times x_2 \rightarrow \mathcal{V}_0(x_2) = v
\]

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.

\[
\mathcal{V}_0(x_0) = v_0, \ldots, \mathcal{V}_0(x_{n-1}) = v_{n-1} \\
\text{block creation} \\
\mathcal{V}_0(x_2) = a_1 \\
\mathcal{M}(a_1) = \mathcal{V}_0(x_1) \\
\rightarrow \mathcal{V}_0(x_2) = a_1 \\
\mathcal{M}(a_2) = \mathcal{V}_1(x_1) \\
\rightarrow \mathcal{V}_0(x_2) = a_2
\]

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.

\[
\mathcal{V}_0(x_2) = a \\
\mathcal{M}(a) = \mathcal{V}_x \\
\mathcal{V}_x(x_1) = v
\]

This reduces to the value of \( x_1 \) in the environment of the set of closures that contains the closure at address \( a_0 \), with \( a_0 \) 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.
3 COMPILING FLAMBDATA 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.

Wasmocaml 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 Flambdata.

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)))
```

The type $block2$ is declared as a sub-type of $block1$ because in the OCaml IRs, the primitive for accessing a block field is un-typed: 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.

```
(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 $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:

```
(func $tag (param $x eqref) (result i32)
  (i32.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:

```ocaml
  (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 functions 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 Currification

In OCaml, functions take only one argument. However, in practice, 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 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 Wasmocaml. 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:

```ocaml
(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 $T$ 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.

$T(n) = i32.const n$

An integer literal is turned into an i31ref in order to match with the uniform representation.
 août 29-August 31, 2023, Braga, Minho, Portugal

Léo Andrès, Pierre Chambart, and Jean-Christophe Filliâtre

\[ T(x) = \text{local.get } \$x \]

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

\[ T(\text{let } x = t_1 \text{ in } t_2) = T(t_1) \]
\[ \text{local.set } \$x \]
\[ T(t_2) \]

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

\[ T(x_1, x_2) = \text{local.get } \$x_1 \]
\[ \text{ref.cast (ref } \$\text{closure}) \]
\[ \text{local.get } \$x_2 \]
\[ \text{ref.as_non_null} \]
\[ \text{local.get } \$x_1 \]
\[ \text{ref.cast (ref } \$\text{closure}) \]
\[ \text{struct.get } \$\text{closure} 0 \]
\[ \text{call_ref } \$\text{mlfun} \]

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 \( \text{eqref} \). Then we need to put the function of the stack. This is done by accessing the field 0 of the closure, which is a \( \text{funcref} \). Then we use \( \text{call_ref} \) to call the function with its two arguments. The type of the function reference is always \( \$\text{mlfun} \).

\[ T(x_1 \leftarrow x_2) = \text{local.get } \$x_2 \]
\[ \text{local.set } \$x_1 \]
\[ \text{i32.const } 0 \]
\[ \text{i31.new} \]

A mutation simply puts the content of \( x_2 \) into \( x_1 \) and leaves \( 0 \) on the stack (this is \( () : \text{unit} \) in Ocaml).

\[ T(\text{if } x_1 \text{ then } t_1 \text{ else } t_2) = \text{local.get } \$x_1 \]
\[ \text{ref.cast i31ref} \]
\[ \text{i31.get_s} \]
\[ \text{if (result (ref null eq)) (then } T(t_1)) \]
\[ \text{(else } T(t_2)) \]
\[ \text{ref.as_non_null} \]

A conditional converts the value of \( x_1 \) to an \( \text{i32} \) and then uses a Wasm \( \text{if} \). The two branches are simply the result of compiling \( t_1 \) and \( t_2 \).

\[ T(\text{switch } x \ B_1 B_2) = \]
\[ (\text{block } \$\text{switch.a (result (ref null eq))}) \]
\[ (\text{block } \$\text{switch.b (result (ref null eq))}) \]
\[ \text{local.get } \$x \]
\[ \text{br_on_cast } \$\text{switch.b (ref null eq) i31ref} \]
\[ ;; \text{handle block case} \]
\[ \text{ref.cast (ref } \$\text{block}) \]
\[ \text{struct.get } \$\text{block 0} \]
\[ \text{global.set } \$\text{switch.v} \]
\[ T(B_1) \]
\[ ;; \text{skip the scalar case} \]
\[ \text{br } \$\text{switch.a} \]
\[ ;; \text{handle scalar case} \]
\[ \text{ref.cast (ref i31)} \]
\[ \text{i31.get_s} \]
\[ \text{global.set } \$\text{switch.v} \]
\[ T(B_2) \]

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 \( T \) in order to compile the various cases of each branches. Let \( i_0, \ldots, i_{n-1} \in \text{dom}(B) \) be such that \( \exists j, j \notin \text{dom}(B) \wedge \forall k \in [0; n[, j \neq i_k \in \text{dom}(B) \) Then we define the auxiliary function as follows:

\[ T(B) = \]
\[ (\text{i32.eq (i32.const } i_{n-1}) (\text{global.get } \$\text{switch.v})) \]
\[ (\text{if (result (ref null eq)) (then } T(B(i_{n-1}))) \]
\[ (\text{else} \]
\[ ;; \ldots \]
\[ (\text{i32.eq (i32.const } i_0) (\text{global.get } \$\text{switch.v})) \]
\[ (\text{if (result (ref null eq)) (then } T(B(i_0)) \]
\[ (\text{else unreachable))) \]

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

\[ T(\text{raise } x_{\text{exn}} x^p) = \text{local.get } \$x_{n-1} \]
\[ \ldots \]
\[ \text{local.get } \$x_0 \]
\[ \text{br } \$\text{exn}_{x_{\text{exn}}} \]

A static raise puts all the value transported by the exception on the stack and branches to the block corresponding to the raise.
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 these values in the right locals.

\[
\mathcal{T}(\text{catch } t_1 \text{ with } x_{\text{exn}} \ x_{\text{exn}} \ t_2) = \\
(\text{block } \$\text{after_try_catch} \ (\text{result } (\text{ref null eq})) \\
(\text{block } \$\text{exn}_x_{\text{exn}} \\
\ (\text{result } (\text{ref null eq})) ; ; 0 \\
\ (\text{result } (\text{ref null eq})) ; ; \ldots \\
\ (\text{result } (\text{ref null eq})) ; ; n - 1 \\
\mathcal{T}(t_1) \\
\text{br } \$\text{after_try_catch}) \\
\text{local.set } \$x_0 \\
\ldots \\
\text{local.set } \$x_{n-1} \\
\mathcal{T}(t_2)) 
\]

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.

\[
\mathcal{T}(\text{set_field } n \ x_1 \ x_2) = \text{local.get } \$x_1 \\
\text{ref.cast } (\text{ref } \$\text{block}) \\
\text{struct.get } \$\text{block} = 1 \\
\text{ref.cast } (\text{ref } \$\text{data}) \\
i32.\text{const } x \\
\text{local.get } \$x_2 \\
\text{array.set } \$\text{data} \\
i32.\text{const } 0 \\
i31.\text{new}
\]

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.

\[
\mathcal{T}(\text{while } t_1 \ t_1 \ \text{do } t_2) = (\text{block } \$\text{exit_while_loop} \\
(\text{loop } \$\text{while_loop} \\
\mathcal{T}(t_1) \\
i31.\text{get}_s \\
i32.\text{eqz} \\
\text{br_if } \$\text{exit_while_loop} \\
\mathcal{T}(t_2) \\
\text{drop} \\
\text{br } \$\text{while_loop })) \\
i32.\text{const } 0 \\
i31.\text{new}
\]

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.

\[
\mathcal{T}(\text{get_field } n \ x) = \text{local.get } \$x \\
\text{ref.cast } (\text{ref } \$\text{block}) \\
\text{struct.get } \$\text{block} = 1 \\
\text{ref.as_non_null} \\
i32.\text{const } n \\
\text{array.get } \$\text{data}
\]

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.

\[
\mathcal{T}(\text{project_closure } x_1 \ x_2) = \\
\text{local.get } \$x_2 \\
\text{ref.cast } (\text{ref } \$\text{set_of_closures}) \\
\text{struct.get } \$\text{set_of_closures} = 0 \\
\text{global.get } \$\text{closure_offset_within_set}_x_1 \\
\text{array.get } \$\text{closures}
\]

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 $n$.

\[
\mathcal{T}(\text{set_field } n \ x_1 \ x_2) = \text{local.get } \$x_1 \\
\text{ref.cast } (\text{ref } \$\text{block}) \\
\text{struct.get } \$\text{block} = 1 \\
\text{ref.cast } (\text{ref } \$\text{data}) \\
i32.\text{const } x \\
\text{local.get } \$x_2 \\
\text{array.set } \$\text{data} \\
i32.\text{const } 0 \\
i31.\text{new}
\]

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.

\[ \mathcal{J}(\text{move\_within\_closure } x_1 \ x_2) = \]
local.get $x_2$
ref.cast (ref $\$closure$)
struct.get $\$closure$ 1
ref.as_non_null
struct.get $\$set\_of\_closures$ 0
global.get $\$closure\_offset\_within\_set\_x_1$
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 \( \mathcal{J} = \mathcal{F}, \mathcal{C} \) we define \( x_0, \ldots, x_{n-1} \)
to be the values that are captured in \( \mathcal{F} \) and \( f_0, \ldots, f_{m-1} \) to be the
names of the functions that are part of the set.

\[ \mathcal{J}(\text{make\_set\_of\_closures } \mathcal{J}) = \]
\[
\text{;; empty closures}
\text{ref.null $\$closures$}
\text{;; generating variables}
\text{(ref.as_non_null (local.get $x_0$))}
\text{;; ...}
\text{(ref.as_non_null (local.get $x_{n-1}$))}
\text{array.new_fixed $\$vars$ \ n}
\text{;; set of closures with empty closures and filled vars}
\text{struct.new $\$set\_of\_closures$}
\text{global.set $\$tmp\_set\_of\_closures$}
\text{global.get $\$tmp\_set\_of\_closures$}
\text{;; generating closures}
\text{ref.func $f_0$}
\text{global.get $\$tmp\_set\_of\_closures$}
\text{struct.new $\$closure$}
\text{;; ...}
\text{ref.func $f_{m-1}$}
\text{global.get $\$tmp\_set\_of\_closures$}
\text{struct.new $\$closure$}
\text{array.new_fixed $\$closures$ \ m}
\text{;; patching the set with closures}
\text{struct.set $\$set\_of\_closures$ 0}
\text{;; returning the set}
\text{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
\text{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 correctness 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 \text{cl_iter_f} from the example given in 2.4 would be compiled to the following Wasm code:

```wasm
(func $cl_iter_f (param $envtwo eqref) (param $l eqref)
  (result eqref) (local $x eqref) (local $e 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) i31
    ref.cast (ref $block$)
    struct.get $block$ 0
    global.set $switch_v$ $global.get $switch_v$
i32.const 0
i32.eq
(i32.switch_a)
  (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))
    (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 $mlfun$
    (if (result (ref null eq))
      (then
        i32.const 0
        i31.new)
```

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 Wasm 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 Wasmocaml 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 unpredictability 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 Wasmocaml 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 re-write 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 foresee 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 JsUnsafe.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 lead 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 target-
ing Wasm-GC is the Guile Scheme compiler. Scheme has many sim-
ilar constraints as OCaml and Guile uses many similar solutions de-
developed independently of Wasiocaml. 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 compila-
tion of OCaml to JavaScript is quite simple as it can almost be sum-
marized 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 se-
manetics. It compiles OCaml bytecode programs to JavaScript by
decompling it back to a simple untyped lambda calculus then per-
forms 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 writ-
ten in C so it is possible without any Wasm extension. In some
cases, it allows to start executing code faster than when compiling
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 Web-
Assembly, the main difference being that WASICaml does not in-
terpret 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 ver-
sions 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 be-
tween the two runtimes cannot be collected).

It compiles the bytecode to Wasm-GC. It was developed after Waso-
caml 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 repre-
sentation 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 ma-
jor 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 col-
lected functional language like OCaml. And we think that the ex-
perience 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 run-
time available. In order to test our compiler we are using V8 with
various flags to enable experimental features such as the GC sup-
port. The design of the various extensions required by Wasiocaml
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, sup-
port effect handlers and to move Wasiocaml to Flambda2. All of
these will allow easy deployment of multi-language software on
the browser while having good and predictable performances.

REFERENCES

ocamlpro/wasiocaml


