



HAL
open science

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 (v2), last revised 1 Sep 2024 (v3)

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 reimplement 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.

KEYWORDS

OCaml, Wasm, compilation, value representation, memory, FFI

ACM Reference Format:

Léo Andrès, Pierre Chambart, and Jean-Christophe Filliâtre. 2023. Wasocaml: compiling OCaml to WebAssembly. In *Proceedings of Symposium on Implementation and Application of Functional Languages (IFL '23)*. ACM, New York, NY, USA, 13 pages. <https://doi.org/XXXXXXX.XXXXXXX>

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

*Also with OCamlPro SAS.

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 are not made or distributed for profit or commercial advantage and 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.XXXXXXX>

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:

```
(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)
```

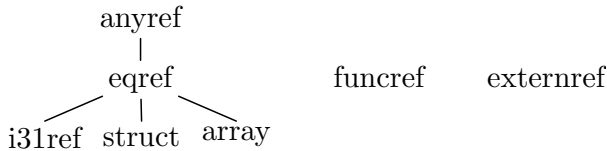
```

117 (i31.get_u
118   (array.get $t1
119     (i32.const 0)
120     (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. Up-casts 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.

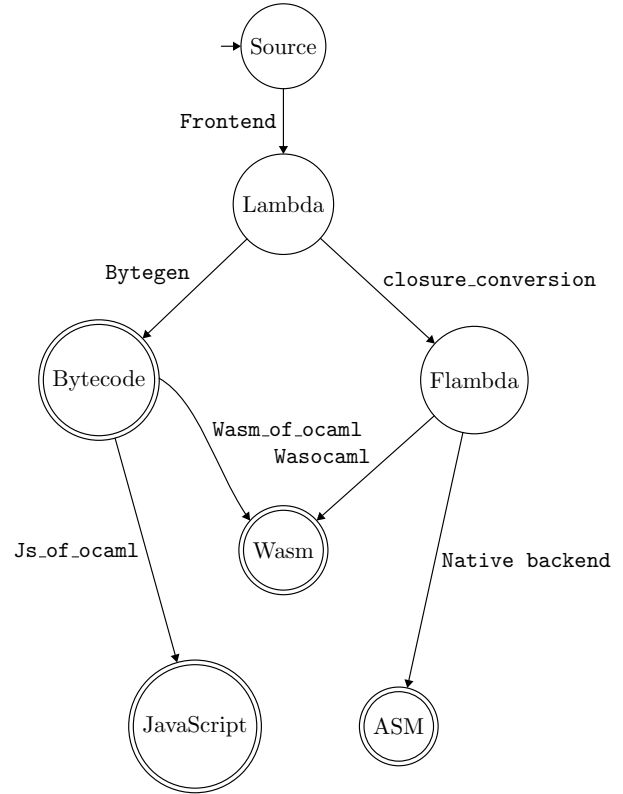
2 SOURCE LANGUAGE

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 chose 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.

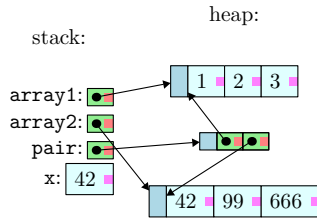
Small scalars are used to represent booleans, unboxed integers, characters or constant constructors of a sum type. Heap-allocated blocks are used to represent arrays, non-constant constructors of a sum type or pairs. For instance, consider the following OCaml code:

```

226 let array1 = [| 1; 2; 3 |]
227 let array2 = [| 42; 99; 666 |]
228 let pair = (array1, array2)
229 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 .

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

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

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

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

$$| \mathcal{S}.x \quad (\text{closure})$$

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

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

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

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

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

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

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

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.

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

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

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

Branches.

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

Term.

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

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

$$| \text{let } x = t \text{ in } t \quad (\text{let-binding})$$

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

$$| x <- x \quad (\text{mutation})$$

$$| \text{if } x \text{ then } t \text{ else } t \quad (\text{conditional})$$

$$| \text{switch } x \mathcal{B} \mathcal{B} \quad (\text{switch})$$

$$| \text{sraise } x x^* \quad (\text{static raise})$$

$$| \text{scatch } t \text{ with } x x^* t \quad (\text{static catch})$$

$$| \text{while}_t t \text{ do } t \quad (\text{while loop})$$

$$| \text{get_field } n x \quad (\text{block field read})$$

$$| \text{set_field } n x x \quad (\text{block field write})$$

$$| \text{make_block } n x^* \quad (\text{block creation})$$

$$| \text{project_closure } x x \quad (\text{closure projection})$$

$$| \text{project_var } x x \quad (\text{closure environment})$$

$$| \text{move_within_closure } x x \quad (\text{closure movement})$$

$$| \text{make_set_of_closures } \mathcal{S} \quad (\text{closures allocation})$$

$$| \text{pop } t \quad (\text{pop environment})$$

$$| x op x \quad (\text{binary operator})$$

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 $\mathcal{V}^*, \mathcal{M}, t$ where \mathcal{V}^* is a stack of environments 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.

$$E ::= \square$$

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

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

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

$$| \text{pop } E$$

Reduction inside a context.

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

Head reductions. In order to reduce cluttering, \mathcal{V}^* and \mathcal{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.

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

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

$$\text{let-binding} \frac{}{\mathcal{V}_0 \mathcal{V}^*, \quad \text{let } x = v \text{ in } t \rightarrow \mathcal{V}_0[x \leftarrow v] \mathcal{V}^*, t}$$

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.

$$\text{application} \frac{\mathcal{V}_0(x_1) = a \quad \mathcal{M}(a) = \mathcal{S}.x \quad \mathcal{S} = _ , \mathcal{C} \quad \mathcal{C}(x) = \lambda x_3 x_4 . t \quad \mathcal{V}_0(x_2) = v}{\mathcal{V}_0 \mathcal{V}^*, \quad \mathcal{M}, \quad x_1 x_2 \rightarrow [x_3 \leftarrow v; x_4 \leftarrow a] \mathcal{V}_0 \mathcal{V}^*, \quad \text{pop } t}$$

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 $\mathcal{S}.x$ where the set of closures $\mathcal{S} = _ , \mathcal{C}$. Let the value of x in the closure map \mathcal{C} be $\lambda x_3 x_4 . t$. The term reduces to $\text{pop } t$ 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 .

$$\text{pop environment} \frac{}{\mathcal{V}_0 \mathcal{V}^*, \text{pop } v \rightarrow \mathcal{V}^*, v}$$

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.

$$\text{mutate} \frac{\mathcal{V}_0(x_1) = v_1 \quad \mathcal{V}_0(x_2) = v_2}{\mathcal{V}_0 \mathcal{V}^*, x_1 \leftarrow x_2 \rightarrow \mathcal{V}_0[x_1 \leftarrow v_2] \mathcal{V}^*, 0}$$

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 $()$: **unit** in OCaml.

$$\text{conditional 1} \frac{\mathcal{V}_0(x) = n \quad n \neq 0}{\mathcal{V}_0 \mathcal{V}^*, \text{ if } x \text{ then } t_1 \text{ else } t_2 \rightarrow t_1}$$

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).

$$\text{conditional 2} \frac{\mathcal{V}_0(x) = n \quad n = 0}{\mathcal{V}_0 \mathcal{V}^*, \text{ if } x \text{ then } t_1 \text{ else } t_2 \rightarrow t_2}$$

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

$$\text{switch 1} \frac{\mathcal{V}_0(x) = n \quad \mathcal{B}_1(n) = t}{\mathcal{V}_0 \mathcal{V}^*, \text{ switch } x \mathcal{B}_1 \mathcal{B}_2 \rightarrow t}$$

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

$$\text{switch 2} \frac{\mathcal{V}_0(x) = a \quad \mathcal{M}(a) = n, v^* \quad \mathcal{B}_2(n) = t}{\mathcal{V}_0 \mathcal{V}^*, \mathcal{M}, \text{ switch } x \mathcal{B}_1 \mathcal{B}_2 \rightarrow t}$$

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

$$\text{raise 1} \frac{x_{\text{exn}} \neq x'_{\text{exn}}}{\text{scatch (sraise } x_{\text{exn}} x^n) \text{ with } x'_{\text{exn}} x'^n t \rightarrow \text{sraise } x_{\text{exn}} x^n}$$

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.

$$\text{raise 2} \frac{x_{\text{exn}} = x'_{\text{exn}}, \quad x^n = x_0, \dots, x_{n-1} \quad x'^n = x'_0, \dots, x'_{n-1} \quad \forall i \in [0; n], \mathcal{V}_0(x_i) = v_i}{\mathcal{V}_0 \mathcal{V}^*, \text{ scatch (sraise } x_{\text{exn}} x^n) \text{ with } x'_{\text{exn}} x'^n t \rightarrow \mathcal{V}_0[x'_0 \leftarrow v_0, \dots, x'_{n-1} \leftarrow v_{n-1}] \mathcal{V}^*, t}$$

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.

$$\text{raise 3} \frac{}{\text{while}_t, (\text{sraise } x_{\text{exn}} x^*) \text{ do } t \rightarrow \text{sraise } x_{\text{exn}} x^*}$$

A while simply re-raises static exceptions.

$$\text{raise 4} \frac{}{\text{let } x = (\text{sraise } x_{\text{exn}} x^*) \text{ in } t \rightarrow \text{sraise } x_{\text{exn}} x^*}$$

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

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

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

465
466
467 while loop 2 $\frac{n \neq 0}{\text{while}_{t'} n \text{ do } t}$
468 $\rightarrow \text{let } _ = t \text{ in while}_{t'} t' \text{ do } t$
469

470 On the contrary, when $n \neq 0$, it reduces to $\text{let } _ = t \text{ in while}_{t'} t' \text{ do } t$.
471 The term t' corresponds to the original term that was used as a condition.
472

473
474 $\mathcal{V}_0(x) = a \quad \mathcal{M}(a) = \text{tag}, v^*$
475 $v^* = v_0, \dots, v_{m-1} \quad m > n \geq 0$
476 field read $\frac{\mathcal{V}_0 \mathcal{V}^*, \mathcal{M}, \text{get_field } n \ x}{\mathcal{V}_0 \mathcal{V}^*, \mathcal{M}, \text{get_field } n \ x \rightarrow v_n}$
477

478 The term $\text{get_field } n \ x$ reduces to the n^{th} value of the block
479 stored at address a with a being the value of x in the current envi-
480 ronment.
481

482 $\mathcal{V}_0(x_1) = a \quad \mathcal{M}(a) = \text{tag}, v^* \quad v^* = v_0, \dots, v_{m-1}$
483 $m > n \geq 0 \quad \mathcal{V}_0(x_2) = v$
484 field write $\frac{\mathcal{V}_0 \mathcal{V}^*, \mathcal{M}, \text{set_field } n \ x_1 \ x_2}{\mathcal{M}[a \leftarrow \text{tag}, v_0, \dots, v_{n-1}, v, v_{n+1}, \dots, v_{m-1}], 0}$
485
486
487
488
489

490 If x_1 is the address a in the current environment, then the term
491 $\text{set_field } n \ x_1 \ x_2$ sets the n^{th} value of the block stored at address
492 a to the value of x_2 in the current environment.
493

494
495 block creation $\frac{\mathcal{V}_0(x_0) = v_0, \dots, \mathcal{V}_0(x_{n-1}) = v_{n-1} \quad a \notin \text{dom}(\mathcal{M})}{\mathcal{V}_0 \mathcal{V}^*, \mathcal{M}, \text{make_block } n_{\text{tag}} \ x_0, \dots, x_{n-1}}$
496 $\rightarrow \mathcal{M}[a \leftarrow n_{\text{tag}}, v_0, \dots, v_{n-1}], 0$
497
498

499 This creates a new block in the store at a fresh address. The tag is
500 given as a value while the others values of the block are read from
501 the current environment.
502

503
504 closure projection $\frac{\mathcal{V}_0(x_2) = a_1 \quad \mathcal{M}(a_1) = \mathcal{S} \quad \mathcal{M}(a_2) = \mathcal{S}.x_1}{\mathcal{V}_0 \mathcal{V}^*, \mathcal{M}, \text{project_closure } x_1 \ x_2}$
505 $\rightarrow a_2$
506
507

508 This reduces to the address a_2 of the closure which has the identi-
509 fier x_1 in the set of closures at address a_1 , with a_1 being the value
510 of x_2 in the current environment. In short, it allows one to get a
511 particular function from a previously allocated set of closures.
512

513
514 variable projection $\frac{\mathcal{V}_0(x_2) = a \quad \mathcal{M}(a) = \mathcal{S}.x \quad \mathcal{S} = \mathcal{V}, _ \quad \mathcal{V}(x_1) = v}{\mathcal{V}_0 \mathcal{V}^*, \mathcal{M}, \text{project_var } x_1 \ x_2 \rightarrow v}$
515
516

517 This reduces to the value of x_1 in the environment of the set of
518 closures that contains the closure at address a_a , with a_a being the
519 value of x_2 in the current environment. In short, when inside a
520 closure, it allows to read a variable from the set of closures the
521 closure belongs to.
522

523
524 closure movement $\frac{\mathcal{V}_0(x_2) = a_1 \quad \mathcal{M}(a_1) = \mathcal{S}.x \quad \mathcal{M}(a_2) = \mathcal{S}.x_1}{\mathcal{V}_0 \mathcal{V}^*, \mathcal{M}, \text{move_within_closure } x_1 \ x_2}$
525 $\rightarrow a_2$
526
527
528

529 This reduces to the address a_2 of the closure x_1 in the set of clo-
530 sures to which the closure at address a_1 belongs to. In short, when
531 inside a closure, it allows to get another closure in the current set
532 of closures.
533

534 $\mathcal{S} = _, \mathcal{C} \quad x_0 \in \text{dom}(\mathcal{C}), \dots, x_{n-1} \in \text{dom}(\mathcal{C})$
535 $\forall a \in [a_0, \dots, a_n], a \notin \text{dom}(\mathcal{M})$
536 set alloc $\frac{\mathcal{M}, \text{make_set_of_closures } \mathcal{S}}{\mathcal{M}[a_n \leftarrow \mathcal{S}; a_0 \leftarrow \mathcal{S}.x_0; \dots; a_{n-1} \leftarrow \mathcal{S}.x_{n-1}], a_n}$
537
538
539

540 This maps the set of closures \mathcal{S} and all its closures to fresh ad-
541 dresses in the store. The term reduces to the address of the set.
542

543 2.4 Example

544 Consider the following OCaml code:

```
545 let f = print_int
546 let rec iter f l =
547   match l with
548   | [] -> ()
549   | hd :: tl ->
550     f hd;
551     iter f tl
```

552 In Flambda, it would be represented as follows:

```
553 let f = (* ... *)
554 let iter =
555   let set =
556     make_closures
557     | cl_iter { f } env ->
558       let otherset =
559         make_set_of_closures
560         (* this is the closure map *)
561         | cl_iter_f { l } envtwo ->
562           switch l
563           | int 0 -> 0
564           | tag 0 ->
565             let x = get_field 0 l in
566             let f = project_var f envtwo in
567             let dummy = f x in
568             let tl = get_field 1 l in
569             envtwo tl
570             (* this is the environment *)
571           | f -> f
572         end
573       in
574         project_closure cl_iter_f otherset
575     in
576       project_closure cl_iter set
577   in
578     (* ... *)
```

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 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 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 \mathcal{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.

$$\mathcal{T}(n) = \text{i32.const } n \\ \text{i31.new}$$

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

813 $\mathcal{T}(x) = \text{local.get } \x

814
815
816 To access the value of an identifier, we simply access the value
817 stored in the local variable of the current function.
818

819
820 $\mathcal{T}(\text{let } x = t_1 \text{ in } t_2) = \mathcal{T}(t_1)$
821 $\text{local.set } \$x$
822 $\mathcal{T}(t_2)$
823

824 A let-binding is compiled by evaluating t_1 , storing its result in the
825 local x and evaluating t_2 .
826

827
828 $\mathcal{T}(x_1 \ x_2) = \text{local.get } \x_1
829 $\text{ref.cast (ref } \$\text{closure)}$
830 $\text{local.get } \$x_2$
831 ref.as_non_null
832 $\text{local.get } \$x_1$
833 $\text{ref.cast (ref } \$\text{closure)}$
834 $\text{struct.get } \$\text{closure } 0$
835 $\text{call_ref } \$\text{mlfun}$
836
837
838

839 To apply a function x_1 to a value x_2 we first put the two param-
840 eters of the function on the stack. The first one is the closure itself,
841 which is needed to access the environment and other functions in
842 the set of closures. The second one can be any value of type **eqref**.
843 Then we need to put the function of the stack. This is done by ac-
844 cessing the field 0 of the closure, which is a **funcref**. Then we use
845 **call_ref** to call the function with its two arguments. The type of
846 the function reference is always **\$mlfun**.
847

848
849 $\mathcal{T}(x_1 <- x_2) = \text{local.get } \x_2
850 $\text{local.set } \$x_1$
851 $\text{i32.const } 0$
852 i31.new
853

854 A mutation simply puts the content of x_2 into x_1 and leaves 0 on
855 the stack (this is **() : unit** in OCaml).
856

857
858 $\mathcal{T}(\text{if } x_1 \text{ then } t_1 \text{ else } t_2) = \text{local.get } \x_1
859 $\text{ref.cast } \text{i31ref}$
860 i31.get_s
861 $(\text{if (result (ref null eq))$
862 $\text{(then } \mathcal{T}(t_1))$
863 $\text{(else } \mathcal{T}(t_2))$
864 ref.as_non_null
865
866

867 A conditional converts the value of x_1 to an **i32** and then uses a
868 Wasm **if**. The two branches are simply the result of compiling t_1
869 and t_2 .
870

871
872 $\mathcal{T}(\text{switch } x \ \mathcal{B}_1 \ \mathcal{B}_2) =$
873 $(\text{block } \$\text{switch_a (result (ref null eq))}$
874 $(\text{block } \$\text{switch_b (result (ref null eq))}$
875 $\text{local.get } \$x$
876 $\text{br_on_cast } \$\text{switch_b (ref null eq) } \text{i31ref}$
877 $;; \text{ handle block case}$
878 $\text{ref.cast (ref } \$\text{block)}$
879 $\text{struct.get } \$\text{block } 0$
880 $\text{global.set } \$\text{switch_v}$
881 $\mathcal{F}(\mathcal{B}_2)$
882 $;; \text{ skip the scalar case}$
883 $\text{br } \$\text{switch_a}$
884 $;; \text{ handle scalar case}$
885 $\text{ref.cast (ref } \text{i31})$
886 i31.get_s
887 $\text{global.set } \$\text{switch_v}$
888 $\mathcal{F}(\mathcal{B}_1))$
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911

912 To compile a switch, we start by testing if x is a block or a scalar
913 value. If it's a block, we store its tag, otherwise we store the value
914 directly. Then, we use an auxiliary function \mathcal{F} in order to compile
915 the various cases of each branches. Let $i_0, \dots, i_{n-1} \in \text{dom}(\mathcal{B})$ be
916 such that $\nexists j, j \notin \text{dom}(\mathcal{B}) \wedge \forall k \in [0; n[, j \neq i_k$. Then we define the
917 auxiliary function as follows:
918

919
920 $\mathcal{F}(\mathcal{B}) =$
921 $(\text{i32.eq (i32.const } i_{n-1}) (\text{global.get } \$\text{switch_v}))$
922 $(\text{if (result (ref null eq)) (then } \mathcal{F}(\mathcal{B}(i_{n-1})))$
923 $(\text{else}$
924 $;; \dots$
925 $(\text{i32.eq (i32.const } i_0) (\text{global.get } \$\text{switch_v}))$
926 $(\text{if (result (ref null eq)) (then } \mathcal{F}(\mathcal{B}(i_0)))$
927 $(\text{else unreachable}))$
928

929 We have a sequence of nested **if** to find which of the binding in
930 \mathcal{B} we should evaluate. If none of them correspond to the value
931 stored in $\$switch_v$ (which has been set by the code generated
932 when compiling the switch) then we have to put an **unreachable**
933 in order to generate Wasm code that typechecks. In practice, the
934 **unreachable** corresponds to a match failure, which should not
935 happen in a valid source program, thus it is indeed unreachable.
936

937
938 $\mathcal{T}(\text{sraise } x_{\text{exn}} \ x^n) = \text{local.get } \x_{n-1}
939 \dots
940 $\text{local.get } \$x_0$
941 $\text{br } \$\text{exn_}x_{\text{exn}}$
942
943
944

945 A static raise puts all the value transported by the exception on the
946 stack and branches to the block corresponding to the raise.
947

```

929
930
931  $\mathcal{T}(\text{scatch } t_1 \text{ with } x_{\text{exn}} x^n t_2) =$ 
932 (block $after_try_catch (result (ref null eq))
933   (block $exn_xexn
934     (result (ref null eq)) ;; 0
935     (result (ref null eq)) ;; ...
936     (result (ref null eq)) ;; n - 1
937      $\mathcal{T}(t_1)$ 
938     br $after_try_catch)
939   local.set $x_0
940   ...
941   local.set $x_{n-1}
942    $\mathcal{T}(t_2)$  )
943
944
945
946

```

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.

```

947
948  $\mathcal{T}(\text{while}_{t_1} t_1 \text{ do } t_2) =$  (block $exit_while_loop
949   (loop $while_loop
950      $\mathcal{T}(t_1)$ 
951     i31.get_s
952     i32.eqz
953     br_if $exit_while_loop
954      $\mathcal{T}(t_2)$ 
955     drop
956     br $while_loop ))
957   i32.const 0
958   i31.new
959
960
961
962
963
964
965
966
967
968
969

```

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.

```

970
971  $\mathcal{T}(\text{get\_field } n \ x) =$  local.get $x
972   ref.cast (ref $block)
973   struct.get $block 1
974   ref.as_non_null
975   i32.const n
976   array.get $data
977
978
979
980
981
982

```

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 .

```

983
984  $\mathcal{T}(\text{set\_field } n \ x_1 \ x_2) =$  local.get $x_1
985   ref.cast (ref $block)
986   struct.get $block 1
987   ref.cast (ref $data)
988   i32.const x
989   local.get $x_2
990   array.set $data
991   i32.const 0
992   i31.new
993
994
995
996
997
998
999

```

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.

```

1000
1001  $\mathcal{T}(\text{make\_block } n \ x_0, \dots, x_{m-1}) =$  i32.const n
1002   local.get $x_0
1003   ...
1004   local.get $x_{m-1}
1005   array.new_fixed $data m
1006   struct.new $block
1007
1008
1009
1010
1011
1012

```

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`.

```

1013
1014  $\mathcal{T}(\text{project\_closure } x_1 \ x_2) =$ 
1015 local.get $x_2
1016 ref.cast (ref $set_of_closures)
1017 struct.get $set_of_closures 0
1018 global.get $closure_offset_within_set_x1
1019 array.get $closures
1020
1021
1022
1023
1024
1025

```

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.

```

1026
1027  $\mathcal{T}(\text{project\_var } x_1 \ x_2) =$ 
1028 local.get $x_2
1029 ref.cast (ref $closure)
1030 struct.get $closure 1
1031 ref.as_non_null
1032 struct.get $set_of_closures 1
1033 global.get $var_offset_within_set_x1
1034 array.get $vars
1035
1036
1037
1038
1039
1040

```

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.

```

1045  $\mathcal{T}(\text{move\_within\_closure } x_1 \ x_2) =$ 
1046
1047
1048
1049     local.get $x_2
1050
1051     ref.cast (ref $closure)
1052
1053     struct.get $closure 1
1054
1055     ref.as_non_null
1056
1057     struct.get $set_of_closures 0
1058
1059     global.get $closure_offset_within_set_x1
1060
1061     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 $\mathcal{S} = \mathcal{V}, \mathcal{C}$ we define x_0, \dots, x_{n-1} to be the values that are captured in \mathcal{V} and f_0, \dots, f_{m-1} to be the names of the functions that are part of the set.

```

1066  $\mathcal{T}(\text{make\_set\_of\_closures } \mathcal{S}) =$ 
1067
1068     ;; empty closures
1069     ref.null $closures
1070
1071     ;; generating variables
1072     (ref.as_non_null (local.get $x_0))
1073
1074     ;; ...
1075     (ref.as_non_null (local.get $x_{n-1}))
1076
1077     array.new_fixed $vars n
1078
1079     ;; set of closures with empty closures and filled vars
1080     struct.new $set_of_closures
1081
1082     global.set $tmp_set_of_closures
1083
1084     global.get $tmp_set_of_closures
1085
1086     ;; generating closures
1087     ref.func $f_0
1088
1089     global.get $tmp_set_of_closures
1090
1091     struct.new $closure
1092
1093     ;; ...
1094     ref.func $f_{m-1}
1095
1096     global.get $tmp_set_of_closures
1097
1098     struct.new $closure
1099
1100     array.new_fixed $closures m
1101
1102     ;; patching the set with closures
1103     struct.set $set_of_closures 0
1104
1105     ;; returning the set
1106     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 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 `cl_iter_f` from the example given in 2.4 would be compiled to the following Wasm code:

```

1103 (func $cl_iter_f (param $envtwo eqref) (param $l eqref)
1104 (result eqref) (local $x eqref) (local $f eqref)
1105 (local $dummy eqref) (local $tl eqref)
1106   (block $switch_a (result (ref null eq))
1107     (block $switch_b (result (ref null eq))
1108       local.get $l
1109       br_on_cast $switch_b (ref null eq) i31ref
1110       ref.cast (ref $block)
1111       struct.get $block 0
1112       global.set $switch_v
1113       global.get $switch_v
1114       i32.const 0
1115       i32.eq
1116       (if (result (ref null eq))
1117         (then
1118           local.get $l
1119           ref.cast (ref $block)
1120           struct.get $block 1
1121           ;; ...
1122           local.get $envtwo
1123           ref.cast (ref $closure)
1124           struct.get $closure 0
1125           call_ref $mlfun)
1126         (else unreachable))
1127       br $switch_a)
1128   ref.cast (ref i31)
1129   i31.get_s
1130   global.set $switch_v
1131   global.get $switch_v
1132   i32.const 0
1133   i32.eq
1134   (if (result (ref null eq))
1135     (then
1136       i32.const 0
1137       i31.new)

```

```

1161   (else
1162     unreachable))))
1163

```

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 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 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 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 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 `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 use `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 were 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 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 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

1393	web up to speed with WebAssembly. In <i>Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation</i> . 185–200.	1451
1394	[4] Daniel Hillerström, Sam Lindley, Robert Atkey, and KC Sivaramakrishnan. 2017. Continuation passing style for effect handlers. (2017).	1452
1395	[5] Donald E Knuth and Peter B Bendix. 1983. Simple word problems in universal algebras. <i>Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970</i> (1983), 342–376.	1453
1396	[6] Chris Lattner. 2008. LLVM and Clang: Next generation compiler technology. In <i>The BSD conference</i> , Vol. 5. 1–20.	1454
1397	[7] Sebastian Markbåge. 2017. <i>OCamlrun WebAssembly</i> . https://github.com/sebmarkbage/ocamlrun-wasm	1455
1398	[8] Antonio Nuno Monteiro. 2022. <i>Melange</i> . https://github.com/melange-re/melange	1456
1399	[9] Richard Musiol. 2018. WebAssembly architecture for Go. https://docs.google.com/document/d/131vjr4DH6JFnB-blm_uRdaC0_Nv3OUwjEY5qVCxCup4 .	1457
1400	[10] Gordon D Plotkin and Matija Pretnar. 2013. Handling algebraic effects. <i>Logical methods in computer science</i> 9 (2013).	1458
1401	[11] Cheng Shao. 2022. WebAssembly backend merged into GHC. https://www.tweag.io/blog/2022-11-22-wasm-backend-merged-in-ghc .	1459
1402	[12] Gerd Stolpmann. 2021. <i>WASICaml</i> . https://github.com/remixlabs/wasicaml	1460
1403	[13] The Dart Project Authors. 2022. <i>dart2wasm</i> . https://github.com/dart-lang/sdk/tree/main/pkg/dart2wasm	1461
1404	[14] Jérôme Vouillon. 2023. <i>Wasm_of_ocaml</i> . https://github.com/ocaml-wasm/wasm_of_ocaml	1462
1405	[15] Jérôme Vouillon and Vincent Balat. 2014. From bytecode to JavaScript: the Js_of_ocaml compiler. <i>Software: Practice and Experience</i> 44, 8 (2014), 951–972.	1463
1406	[16] WebAssembly Community Group participants. 2015. <i>Binaryen</i> . https://github.com/WebAssembly/binaryen	1464
1407	[17] WebAssembly Community Group participants. 2017. <i>Exception Handling Proposal for WebAssembly</i> . https://github.com/WebAssembly/exception-handling	1465
1408	[18] WebAssembly Community Group participants. 2017. <i>GC Proposal for WebAssembly</i> . https://github.com/WebAssembly/gc	1466
1409	[19] WebAssembly Community Group participants. 2017. <i>Tail Call Proposal for WebAssembly</i> . https://github.com/WebAssembly/tail-call	1467
1410	[20] WebAssembly Community Group participants. 2018. <i>Reference Types Proposal for WebAssembly</i> . https://github.com/WebAssembly/reference-types	1468
1411	[21] WebAssembly Community Group participants. 2020. <i>Typed Function References Proposal for WebAssembly</i> . https://github.com/WebAssembly/function-references	1469
1412	[22] WebAssembly Community Group participants. 2021. <i>JavaScript-Promise Integration Proposal for WebAssembly</i> . https://github.com/WebAssembly/js-promise-integration	1470
1413	[23] WebAssembly Community Group participants. 2021. <i>Stack Switching Proposal for WebAssembly</i> . https://github.com/WebAssembly/stack-switching	1471
1414	[24] WebAssembly Community Group participants. 2022. <i>Reference-Typed Strings Proposal for WebAssembly</i> . https://github.com/WebAssembly/stringref	1472
1415	[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 .	1473
1416	[26] Andy Wingo. 2023. Compiling Scheme to WasmGC. https://github.com/WebAssembly/meetings/blob/main/gc/2023/GC-04-18.md .	1474
1417	[27] Andy Wingo. 2023. <i>Hoot</i> . https://gitlab.com/spritely/guile-hoot/-/blob/main/design/ABI.md	1475
1418		1476
1419		1477
1420		1478
1421		1479
1422		1480
1423		1481
1424		1482
1425		1483
1426		1484
1427		1485
1428		1486
1429		1487
1430		1488
1431		1489
1432		1490
1433		1491
1434		1492
1435		1493
1436		1494
1437		1495
1438		1496
1439		1497
1440		1498
1441		1499
1442		1500
1443		1501
1444		1502
1445		1503
1446		1504
1447		1505
1448		1506
1449		1507
1450		1508