

# Reconstructing Big-Step Continuation-Passing Semantics for WebAssembly

Guannan Wei<sup>1,2</sup>, Alexander Y. Bai<sup>2</sup>, Dinghong Zhong<sup>3</sup>, and Jiatai Zhang<sup>2</sup>

<sup>1</sup> INRIA/ENS-PSL, Paris, France [guannan.wei@inria.fr](mailto:guannan.wei@inria.fr)

<sup>2</sup> Tufts University, Medford, USA

{[guannan.wei](mailto:guannan.wei@tufts.edu), [alexander.bai](mailto:alexander.bai@tufts.edu), [jiatai.zhang](mailto:jiatai.zhang@tufts.edu)}@tufts.edu

<sup>3</sup> Unaffiliated [azhong.934@gmail.com](mailto:azhong.934@gmail.com)

**Abstract.** WebAssembly is now a popular low-level intermediate representation (IR) and compilation target. The official specification of WebAssembly provides a small-step reduction semantics. Unlike other common low-level IRs, WebAssembly provides structured control-flow constructs, whose reduction rules are complicated by additional administrative instructions in the reference semantics. This paper provides an alternative big-step semantics in continuation-passing style (CPS) that is concise, compositional, and tail-recursive. Using continuations from the meta-language eliminates the need to introduce administrative instructions. We further demonstrate that the CPS semantics can be extended to support other forms of control abstractions, such as effect handlers.

**Keywords:** WebAssembly · Continuation-passing style · Big-step semantics.

## 1 Introduction

*Motivation* WebAssembly (Wasm) is a low-level intermediate representation aiming to be portable, compact, and efficient for the web. Unlike similar languages (e.g. LLVM’s IR), Wasm provides structured control-flow constructs whose behaviors are regulated by a static type system. Another notable advancement with Wasm is that its semantics has been formally specified [11] from the outset and have evolved [32] in tandem with the language’s development.

The co-developed formal specification facilitates not only the development of language tools and compilers, but also the design and prototyping of new extensions for Wasm. For example, there are a number of proposals adding richer control abstractions to Wasm, such as exception handling [28], continuations [30], effect handlers [17], and more, all building on top of the reference semantics.

The specification of Wasm describes a small-step reduction semantics [18], which is straightforward to implement as an interpreter. However, the reduction semantics [11, 32] is complicated by the additional *administrative instructions*, which are extensions to handle control constructs (e.g., loops and breaks) but do not appear in source programs. These administrative instructions are inserted on-the-fly during the evaluation of the program, essentially serving as the

representation of evaluation contexts. Another choice of the design is that the reference semantics conflates the control stack and the value stack [27], arguably adding cognitive overhead to understand the semantics. The redundancy also leads to inefficiency when implementing the reference interpreter.

Moreover, the reference small-step semantics is *not compositional*, making it unsuitable to apply mechanical program transformations (e.g. partial evaluation, staging, or unfolding) to the interpreter [14], which are useful to derive other tools or optimizations based on the semantics. The lack of compositionality adds complication in both reasoning about the semantics and in developing tools that adhere to a semantics-first approach.

***This Work*** Following Reynolds’ seminal work of definitional interpreters [21], an alternative is to leverage the control mechanisms from the meta-language to define the semantics of the object language, rather than using first-order representations of control. We can give meanings to the control constructs of the object language by using continuations from the higher-order meta-language.

This paper reconstructs a continuation-passing style (CPS) semantics for Wasm. The key contribution is a big-step, compositional, tail-recursive semantics in CPS that captures the essence of Wasm’s control-flow semantics. Such a CPS semantics is high-level and can be straightforwardly implemented in a functional language. By employing CPS in the meta-language, our approach eliminates the need for administrative instructions, disentangles the control stack and the operand value stack, makes the semantics more concise and streamlined compared to the reference reduction semantics [11, 32].

Unlike a typical CPS interpreter with a single continuation, the interpreter uses a list (or, a trail) of continuations to handle lexically induced breaks, inspired by double-barrelled CPS [23] and the CPS semantics of dynamic continuations [2]. We further demonstrate that the CPS semantics serves as a flexible foundation that can be extended to define a wide range of control abstractions, which are not (yet) in the current WebAssembly standard, such as high-level for-loop [22], tail calls [29], exceptions [22, 28], and effect handlers [17, 19, 20].

***Contributions*** This work makes the following contributions:

- We clarify WebAssembly’s control-flow semantics by developing big-step, CPS-based definitional interpreters, marking the first compositional and tail-recursive CPS semantics for WebAssembly.
- We extend the CPS semantics to support control abstractions such as structured loops, tail calls, try/catch, and effect handlers.
- We implement an WebAssembly interpreter based on the proposed semantics, validating its consistency with the reference interpreter.

The paper is organized as follows: Section 2 introduces  $\mu$ Wasm, a minimal language following the essence of Wasm. Section 3 presents the core CPS semantics for  $\mu$ Wasm. Section 4 discusses several control extensions and their CPS semantics. Section 5 describes the implementation of our CPS interpreter. Section 6 discusses related work and Section 7 concludes with remarks on future work.

## 2 $\mu$ Wasm: A Minimal Language à la WASM

In this section, we introduce a minimal language  $\mu$ Wasm, capturing interesting control flow constructs as a proper subset of WebAssembly. We first present the abstract syntax, followed by an example informally explaining the control-flow semantics.

### 2.1 Syntax

$$\begin{array}{ll}
 \ell \in \text{Label} & = \mathbb{N} \\
 x \in \text{Identifier} & = \mathbb{N} \\
 t \in \text{ValueType} & ::= i32 \mid i64 \mid \dots \\
 ft \in \text{FunctionType} & ::= t^* \rightarrow t^* \\
 e \in \text{Instruction} & ::= \text{nop} \mid t.\text{const } c \mid t.\{\text{add, sub, eq, } \dots\} \\
 & \quad \mid \text{local.get } x \mid \text{local.set } x \\
 & \quad \mid \text{block } ft \text{ } es \mid \text{loop } ft \text{ } es \mid \text{if } ft \text{ } es \text{ } es \\
 & \quad \mid \text{br } \ell \mid \text{call } x \mid \text{return} \\
 es \in \text{Instructions} & = \text{List}[\text{Instruction}] \\
 f \in \text{Function} & ::= \text{func } x \{ \text{type} : ft, \text{locals} : t^*, \text{body} : es \} \\
 m \in \text{Module} & ::= \text{module } f^*
 \end{array}$$

Fig. 1: The abstract syntax of  $\mu$ Wasm.

Figure 1 presents the abstract syntax of  $\mu$ Wasm. A  $\mu$ Wasm module consists of a sequence of function definitions. Each function defines its type, identifier, a list of types of local variables, and a block of instructions.

WebAssembly assumes a stack-based computation model (instead of using named registers), thus most instructions do not take explicit operands. For example, `i32.const 42` pushes the constant 42 onto the stack, and a numeric instruction such as `i32.add` consumes two values from the stack and pushes the result back. Instructions for local variables, e.g., `local.get` and `local.set`, accesses and updates local variables with an implicit stack too, respectively.

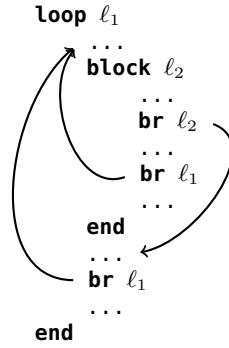
Control flow constructs in Wasm are structured and can be nested. For example, block-like constructs (`block`, `loop`, `if`) contain sequences of instructions. They also take a function type regulating the shape of the stack before and after the block. Break instruction (`br`) takes a label, which is a numeric value referring to the jump targets in the control flow. Block-like constructs introduce an implicit label à la de Bruijn indices, which can be referred by `br` within the block.

Compared to the full-fledged Wasm language, we have omitted many features, e.g., memory operations, globals, imports/exports, tables, etc. However, these

features are largely orthogonal to the control flow semantics of Wasm, and can be added back in a straightforward manner. Labels (for blocks) and identifiers (for local variables and functions) are restricted to numerics compared to standard Wasm, in which can be symbolics. Symbolic labels/identifiers can be mapped to their numeric correspondences as a semantics-preserving preprocessing step. In this work, we are interested in the dynamic semantics of Wasm, therefore we assume the same validation semantics for  $\mu$ Wasm as in standard WebAssembly.

## 2.2 Example

Although WebAssembly looks like a standard stack-based low-level language, one of the uncommon feature of WebAssembly is that the semantics of breaks contextually depends on the enclosing target block. For example, the following code snippet defines a loop block containing a regular block, labeled  $\ell_1$  and  $\ell_2$  respectively:



There are three `br` instructions in the code snippet. Within the inner block, the first `br` instruction jumps to the end of the block, while the second `br` instruction jumps to the beginning of the enclosing loop block. Similarly, the `br` instruction at the end of the loop block jumps back to the beginning of the loop block. If there is no `br` back to the loop head, the loop finishes. Our CPS semantics will capture this contextual control-flow structure.

## 3 A CPS Semantics for $\mu$ Wasm

We now present a continuation-passing semantics for  $\mu$ Wasm. The semantics  $\llbracket \cdot \rrbracket$  is defined as a recursive function mapping a *list of instructions* to functions with continuations:

$$\llbracket \cdot \rrbracket : \text{List}[\text{Inst}] \rightarrow (\text{Stack} \times \text{Env} \times \text{Cont} \times \text{Trail}) \rightarrow \text{Ans}$$

In the following, we first explain the domain definitions and notations, then present the definition of  $\llbracket \cdot \rrbracket$  in Figure 3. We also discuss bridging the gap between our core CPS semantics and the full Wasm language.

$$\begin{aligned}
v \in \text{Value} &= \mathbb{Z} \\
\sigma \in \text{Stack} &= \text{List}[\text{Value}] \\
\rho \in \text{Env} &= \text{List}[\text{Value}] \\
\kappa \in \text{Cont} &= \text{Stack} \times \text{Env} \rightarrow \text{Ans} \\
\theta \in \text{Trail} &= \text{List}[\text{Cont}]
\end{aligned}$$

Fig. 2: Semantic domains and auxiliary functions.

**Evaluation function:**  $\llbracket \cdot \rrbracket : \text{List}[\text{Inst}] \rightarrow (\text{Stack} \times \text{Env} \times \text{Cont} \times \text{Trail}) \rightarrow \text{Ans}$

$$\begin{aligned}
\llbracket \text{nil} \rrbracket(\sigma, \rho, \kappa, \theta) &= \kappa(\sigma, \rho) \\
\llbracket \text{nop} :: \text{rest} \rrbracket(\sigma, \rho, \kappa, \theta) &= \llbracket \text{rest} \rrbracket(\sigma, \rho, \kappa, \theta) \\
\llbracket t.\text{const } c :: \text{rest} \rrbracket(\sigma, \rho, \kappa, \theta) &= \llbracket \text{rest} \rrbracket(c :: \sigma, \rho, \kappa, \theta) \\
\llbracket t.\text{add} :: \text{rest} \rrbracket(v_1 :: v_2 :: \sigma, \rho, \kappa, \theta) &= \llbracket \text{rest} \rrbracket(v_1 + v_2 :: \sigma, \rho, \kappa, \theta) \\
\llbracket \text{local.get } x :: \text{rest} \rrbracket(\sigma, \rho, \kappa, \theta) &= \llbracket \text{rest} \rrbracket(\rho(x) :: \sigma, \rho, \kappa, \theta) \\
\llbracket \text{local.set } x :: \text{rest} \rrbracket(v :: \sigma, \rho, \kappa, \theta) &= \llbracket \text{rest} \rrbracket(\sigma, \rho[x \mapsto v], \kappa, \theta) \\
\llbracket \text{block } (t^m \rightarrow t^n) \text{ es} :: \text{rest} \rrbracket(\sigma_{\text{arg } m} \uparrow \sigma, \rho, \kappa, \theta) &= \\
\quad \text{let } \kappa_1 := \lambda(\sigma_1, \rho_1). \llbracket \text{rest} \rrbracket(\llbracket \sigma_1 \rrbracket_n \uparrow \sigma, \rho_1, \kappa, \theta) \text{ in} & \\
\quad \llbracket \text{es} \rrbracket(\sigma_{\text{arg}}, \rho, \kappa_1, \kappa_1 :: \theta) & \\
\llbracket \text{loop } (t^m \rightarrow t^n) \text{ es} :: \text{rest} \rrbracket(\sigma_{\text{arg } m} \uparrow \sigma, \rho, \kappa, \theta) &= \\
\quad \text{let } \kappa_1 := \lambda(\sigma_1, \rho_1). \llbracket \text{rest} \rrbracket(\llbracket \sigma_1 \rrbracket_n \uparrow \sigma, \rho_1, \kappa, \theta) \text{ in} & \\
\quad \text{fix } \kappa_2 := \lambda(\sigma_2, \rho_2). \llbracket \text{es} \rrbracket(\llbracket \sigma_2 \rrbracket_m, \rho_2, \kappa_1, \kappa_2 :: \theta) \text{ in} & \\
\quad \kappa_2(\sigma_{\text{arg}}, \rho) & \\
\llbracket \text{if } (t^m \rightarrow t^n) \text{ es}_1 \text{ es}_2 :: \text{rest} \rrbracket(v :: \sigma_{\text{arg } m} \uparrow \sigma, \rho, \kappa, \theta) &= \\
\quad \text{let } \text{es} := \text{if } v \equiv 0 \text{ then } \text{es}_2 \text{ else } \text{es}_1 \text{ in} & \\
\quad \text{let } \kappa_1 := \lambda(\sigma_1, \rho_1). \llbracket \text{rest} \rrbracket(\llbracket \sigma_1 \rrbracket_n \uparrow \sigma, \rho_1, \kappa, \theta) \text{ in} & \\
\quad \llbracket \text{es} \rrbracket(\sigma_{\text{arg}}, \rho, \kappa_1, \kappa_1 :: \theta) & \\
\llbracket \text{br } \ell :: \text{rest} \rrbracket(\sigma, \rho, \kappa, \theta) &= \theta(\ell)(\sigma, \rho) \\
\llbracket \text{call } x :: \text{rest} \rrbracket(\sigma_{\text{arg } m} \uparrow \sigma, \rho, \kappa, \theta) &= \\
\quad \text{let } \{\text{type} : t^m \rightarrow t^n, \text{locals} : ts, \text{body} : \text{es}\} := \text{lookupFunc}(x) \text{ in} & \\
\quad \text{let } \rho_1 := \text{buildEnv}(\sigma_{\text{arg}}, ts) & \\
\quad \text{let } \kappa_1 := \lambda(\sigma_1, \rho_1). \llbracket \text{rest} \rrbracket(\llbracket \sigma_1 \rrbracket_n \uparrow \sigma, \rho, \kappa, \theta) \text{ in} & \\
\quad \llbracket \text{es} \rrbracket(\llbracket \cdot \rrbracket, \rho_1, \kappa_1, [\kappa_1]) & \\
\llbracket \text{return} :: \text{rest} \rrbracket(\sigma, \rho, \kappa, \theta) &= \theta.\text{last}(\sigma, \rho)
\end{aligned}$$

Fig. 3: The continuation-passing style semantics of  $\mu\text{Wasm}$ .

### 3.1 Preliminaries

**Domains** Figure 2 shows the definitions of semantic domains. We represent values as integers, and the stack as a list of values. An environment (i.e. frames in Wasm reference semantics)  $\rho$  is represented as a list of values, mapping numeric-represented local variables as indices to values. We represent continuations  $\kappa$  as functions that take a stack and an environment, and return an answer. We deliberately leave the definition of the answer type `Ans` abstract, as it depends on the specific instantiation of the semantics. One possible choice is to define it as the `Stack` type, representing the side effects of instructions. A trail  $\theta$  is a list (or stack) of continuations, which is used to represent lexical control structures.

**Notations** Before explaining the details of the semantics, we introduce some notations used in Figure 3.

For a list  $\theta$  (representing `Stack`, `Env`, or `Trail`),  $\theta(\ell)$  accesses the  $\ell$ -th element in the list  $\theta$ .  $\sigma_1 \uparrow\uparrow \sigma_2$  concatenates two stacks  $\sigma_1$  and  $\sigma_2$ .  $\rho[x \mapsto v]$  updates the environment  $\rho$  with  $x$  mapping to  $v$ , producing a new environment.

We write a function type  $t^m \rightarrow t^n$  to denote that there are  $m$  argument types, and  $n$  result types. Given a type  $t^m$  and a stack  $\sigma$ ,  $[\sigma_1]_m$  takes the top  $m$  elements matching the types  $t^m$  from the stack  $\sigma$ . This is useful, e.g., to truncate the stack only taking the necessary arguments.

To the left of the equations in Figure 3, we use a few notations to match the program syntax and the shape of the operand stack. For example, `::` is the usual cons operator used to destruct the stack into the top element and the rest of the stack. We also use  $\sigma_1 \uparrow_m \uparrow\uparrow \sigma_2$  to split the stack into two parts, where the first part  $\sigma_1$  has  $m$  values, and the second part  $\sigma_2$  contains the rest of the stack.

### 3.2 CPS Semantics

Figure 3 presents the CPS semantics of  $\mu$ Wasm, which structurally recurs over the list of instructions. The interpretation of instructions manipulating the stack and environment is straightforward to define. For example, `const` pushes a constant onto the stack, `add` pops two values from the stack, adds them, and pushes the result back. Then, they recursively evaluate the *rest* of the instructions with the new stack. Similarly, `local.get` and `local.set` access and update local variables with the environment.

Besides storage arguments such as the stack and environment to the interpreter, our semantics takes additional control arguments. In WebAssembly, there are three different ways to leave a block-like scope, namely by (1) falling through to the immediately enclosing block, (2) breaking out of the block, and (3) returning from the function. Therefore, to accommodate such control flow,  $\llbracket \cdot \rrbracket$  takes additional control arguments:

- a continuation function  $\kappa$  representing the control-flow after consuming the current list of instructions (i.e. the case for  $\llbracket nil \rrbracket$ , yielding to the enclosing block), and

- a trail of continuations  $\theta$  representing the control-flow for jumping (informed by the lexical structure of the program) or returning.

In the following, we explain how they play different roles for different control constructs.

**Block, Loop, and Break** For control-flow instructions,  $\mu$ Wasm models three kinds of block-like structures (block, loop, and if) which introduce a labeled scope, along with the branch instruction `br`. Within a block-like construct, a break instruction `br` consisting of a label, which indicates the number of blocks to skip. For example, `br 0` targets the immediately enclosing block, and `br 1` targets the next outer block, and so on. However, it is important to note that `br`s to a loop and to a block behave differently: the former jumps back to the beginning of the loop (as `continue` in C), while the latter escapes to the end of the block (recall the illustration in Section 2.2). In other words, the meaning of `br` depends on its enclosing constructs. Our semantics uses a trail  $\theta$  to keep track of the lexical control-flow structure, then labels of `br`s are interpreted as the index of the trail to invoke the corresponding continuation.

Now we explain the rules defined in Figure 3. For a block instruction, we prepare a new continuation  $\kappa_1$  that evaluates the rest of the instructions after the block. This continuation is shared by falling through the block and breaking out of the block (appended to the trail  $\theta$ ). Similarly, for a if instruction, we prepare the same kind of continuation, but evaluate the inner instructions based on the condition on the top of the stack.

However, for a loop instruction, we define two different continuations: (1)  $\kappa_1$  represents the fall-through continuation that evaluates the rest of the instructions syntactically after the loop, and (2)  $\kappa_2$  represents the break-out continuation that jumps back to the beginning of the loop. Note that  $\kappa_2$  is defined as a fixed-point, so that it can be recursively pushed onto the trail  $\theta$  within its definition. Lastly, we initiate the evaluation of the loop body by invoking  $\kappa_2$ .

Block-like structures are also annotated with a function type, indicating the stack values consumed and produced by the block. In Figure 3, we use  $\sigma_{arg\ m} \# \sigma$  to split the input stack for the argument values, and use  $[\sigma_1]_n$  to truncate the output stack for the return values.

**Call and Return** For function calls, we use an auxiliary function `lookupFunc` to find the definition (i.e., its function type, local variable types, and body) by the function index. Our presentation omits the function environment and the definition of `lookupFunc`, as it is straightforward to implement and not the focus of this paper. Another auxiliary function `buildEnv` constructs a new environment  $\rho_1$  containing the arguments and default values of local variables (Appendix A shows its definition). The continuation  $\kappa_1$  evaluates the rest of the instructions after the call, with the returned values appended with the previous stack (without the arguments to the callee), the caller’s environment, continuation, trail, and return index.

When we enter the function body `es`, we use an empty stack, the new environment  $\rho_1$ , and the return continuation  $\kappa_1$ . Since a function introduces a *fresh*

block scope too, the trail  $\theta$  is a just singleton list containing the return continuation  $\kappa_1$ , discarding all other jump targets that are non-local to the function. Within a function body, this continuation  $\kappa_1$  remains to be the last one in the trail, and it is invoked when encountering a `return` instruction, since it is the maximal number of blocks to skip.

**Remarks** The presented semantics (Figure 3) as a definitional interpreter is compositional. We apply the  $\llbracket \cdot \rrbracket$  function only to the syntactic sub-constructs of the current term from the left-hand side. The interpreter is also tail-recursive – every call of  $\llbracket \cdot \rrbracket$  and the continuations are in tail position. In a meta-language with proper tail-call optimization, the interpreter executes without stack overflow. In Section 4.2, we discuss the extension to support tail calls in the object language  $\mu\text{Wasm}$ .

Notice that our evaluation function takes a list of instructions as argument. Alternatively, we could define it as only taking a single instruction, instead a list of instructions. In this way, we need to define an auxiliary function (e.g. `foldl`) that iterates the evaluation over the list of instructions, so that the evaluation of *rest* becomes part of the continuation  $\kappa$  of the current instruction. We choose the former approach that works better with the low-level syntactic structure of WebAssembly; our evaluation function can also be considered as the fused version of the single-instruction evaluation and the iterating driver.

**Scaling to Full Wasm** Many features of WebAssembly, such as memory, global variables, and tables for function references, are orthogonal to the control flow semantics. They are omitted in our semantics but nevertheless can be added on top of it, following the reference interpreter. Our presented semantics have not dealt with errors, which are represented by the dedicated administrative instruction `trap` in WebAssembly. In our semantics, the interpretation of `trap` can be represented by errors in the host language.

## 4 Extensions

We have shown the CPS semantics for  $\mu\text{Wasm}$ . Now, we discuss several control extensions to  $\mu\text{Wasm}$  and show that the CPS semantics can be extended to support these extensions. Some of the simple constructs can be directly added to the language without global changes, while others such as exception/effect handling require additional facilities. We start from the simple constructs and gradually move to more complex ones.

### 4.1 Structured Loops

We first show how to add structured `for-loop` in  $\mu\text{Wasm}$ . The `for-loop` resembles the behavior of similar looping constructs in other high-level languages, such as C. This extension is inspired by one of the assignment problems in Stanford’s CS242 course [22].



**Syntax and Statics** The syntax of  $\mu\text{Wasm}$  is extended with the following construct:

$$e \in \text{Instruction} ::= \dots \mid \text{for } (es_{\text{init}}; es_{\text{cond}}; es_{\text{post}}) es$$

The initialization instructions  $es_{\text{init}}$  run only once before the loop begins. The condition  $es_{\text{cond}}$  produces a single boolean (represented as `i32`) value on the stack. If the condition is true, then the loop body  $es$  is executed otherwise the loop terminates. The post-instructions  $es_{\text{post}}$  are executed (every time) after the loop body  $es$ . As for typing, only the condition  $es_{\text{cond}}$  has type  $[] \rightarrow [\text{i32}]$ ; all other constructs in the for-loop take no arguments and produce no results on the stack.

**CPS Semantics** We formalize the CPS semantics for for-loop, following the same style presented in Figure 3:

$$\begin{aligned} \llbracket \text{for } (es_{\text{init}}; es_{\text{cond}}; es_{\text{post}}) es :: rest \rrbracket (\sigma, \rho, \kappa, \theta) = \\ \text{fix } \kappa_1 := \lambda(\sigma_1, \rho_1). \llbracket es_{\text{cond}} \rrbracket ([], \rho_1, \lambda(v :: \sigma_2, \rho_2). \\ \quad \text{if } v \equiv 0 \text{ then } \llbracket rest \rrbracket (\sigma, \rho_2, \kappa, \theta) \\ \quad \text{else } \llbracket es \rrbracket ([], \rho_2, \lambda(\sigma_3, \rho_3). \llbracket es_{\text{post}} \rrbracket ([], \rho_3, \kappa_1, \theta), \theta)) \text{ in} \\ \llbracket es_{\text{init}} \rrbracket ([], \rho, \kappa_1, \theta) \end{aligned}$$

Similar to the semantics of `loop`-block, we use a fixed-point to define continuation  $\kappa_1$  that evaluates the condition and possibly the loop body followed by the post-instructions. This continuation  $\kappa_1$  is used as the continuation when recursively applying the semantic function to  $es_{\text{post}}$ .

Note that in the above definition, we have deliberately left for-loop no branch semantics, i.e. a `br0` instruction in the loop body would target the outer enclosing block, not the loop itself. However, the use of continuations in our semantics allows us to flexibly recover behaviors of `br` either as `continue` or `break` in C. For example, we can append  $\kappa_1$  to the trail continuation  $\theta$  when evaluating the loop body  $es$ , and use it as the target for a `br0`. Similarly, to implement a `break` behavior, we can append the evaluation of  $rest$  to the trail  $\theta$ . Appendix B shows the full semantics of the for-loop with both variants of break semantics.

## 4.2 Tail Calls

Standard WebAssembly's `call` instruction prohibits tail-call optimization, which is useful particularly for recursive functions since it ensures constant stack space consumption for tail calls. A recent proposal [29] extends WebAssembly with tail calls, and has been experimentally supported by major implementations. Although our CPS semantics does not operationally describe every detail of a low-level virtual machine or implementation, it can help clarify the behavior of tail calls, especially from the perspective of continuations. Essentially, it becomes unnecessary to create a new frame/continuation for a tail call, as we can directly reuse the current one that returns.

**Syntax and Statics** The proposal [29] adds several new call instructions that are the tail version of the regular ones. For brevity, we only demonstrate the `return_call` instruction with the following syntax:

$$e \in \text{Instruction} ::= \dots \mid \text{return\_call } x$$

Here, `return_call x` represents a tail call to the function at index  $x$ . We omit the typing, which follows the proposal [29].

**CPS Semantics** As its name suggested, the instruction `return_call` combines the semantics of `return` and `call`, performing them in a single step. In the CPS semantics, we do not need to return explicitly; instead, to return is to call the last continuation in the trail. And the caller prepares a new continuation for the callee to return. Combining these two steps into one eliminates the need to prepare a new continuation for the callee, as shown in the following definition:

$$\begin{aligned} \llbracket \text{return\_call } x :: \text{rest} \rrbracket (\sigma_{\text{arg } m} \uplus \sigma, \rho, \kappa, \theta) = \\ \text{let } \{\text{type} : t^m \rightarrow t^n, \text{locals} : ts, \text{body} : es\} := \text{lookupFunc}(x) \text{ in} \\ \text{let } \rho_1 := \text{buildEnv}(\sigma_{\text{arg}}, ts) \\ \llbracket es \rrbracket ([], \rho_1, \kappa, [\kappa]) \end{aligned}$$

The definition is nearly identical to the semantics of the regular function call, except that we do not need to create a new continuation for the rest of instructions after the call. Since the tail call is the last operation inside a function, we can directly reuse the current continuation  $\kappa$  for the callee.

### 4.3 Exceptions

In standard WebAssembly and  $\mu\text{Wasm}$ , the jump target  $\ell$  in a break instruction `br  $\ell$`  is entirely static and local within a function. It is not possible to jump out of the function via `br` or similar break instructions. Now we demonstrate our semantics can be extended to support exception handling, a simple form of dynamic, non-local control flow.

There is already a work-in-progress proposal for exception handling in WebAssembly [28]. Without introducing heavy mechanisms (e.g. a table describing multiple handlers as in the proposal), we choose to demonstrate a higher-level `try/catch` construct, which is close to the description in one of the assignment problems in Stanford’s CS242 course [22]. In the next section, we will further generalize it to more flexible effect handlers.

**Syntax and Statics** The following shows the syntax for the new instructions:

$$e \in \text{Instruction} ::= \dots \mid \text{try } es_1 \text{ catch } es_2 \mid \text{throw}$$

Instructions  $es_1$  is the body that warrants to throw an exception, and  $es_2$  is the handler that catches the exception. As for typing their stack behavior,  $es_1$  takes

no stack argument and returns no argument. The `throw` instruction expects an error code on the top of the stack, which becomes the input to  $es_2$ . Unlike a full-blown exception handling mechanism, we omit “exception tags” that can be used to differentiate the types of exceptions.

**CPS Semantics** When we encounter a try-catch block, the instructions in  $es_1$  are evaluated first. If an exception is thrown during the dynamic extent of  $es_1$ , control is transferred to  $es_2$  (provided no other handler is installed within). Otherwise, execution continues with the instruction following the try-catch block. To model such a behavior, we extend the semantics  $\llbracket \cdot \rrbracket$  with an additional failure continuation [23] (or, handler), denoted as  $\gamma$  argument of type `Cont`:

$$\begin{aligned} \llbracket \text{try } es_1 \text{ catch } es_2 :: rest \rrbracket(\sigma, \rho, \kappa, \theta, \gamma) = & \\ \text{let } \kappa_1 := \lambda(\sigma_1, \rho_1). \llbracket rest \rrbracket(\sigma, \rho_1, \kappa, \theta, \gamma) \text{ in} & \\ \text{let } \gamma_1 := \lambda(\sigma_2, \rho_2). \llbracket es_2 \rrbracket(\sigma_2, \rho_2, \kappa_1, \theta, \gamma) \text{ in} & \\ \llbracket es_1 \rrbracket(\sigma, \rho, \kappa_1, \theta, \gamma_1) & \\ \llbracket \text{throw } :: rest \rrbracket(v :: \sigma, \rho, \kappa, \theta, \gamma) & = \gamma([v], \rho) \end{aligned}$$

The success continuation  $\kappa_1$  is prepared to evaluate the rest of the instructions after the try-catch block. The failure continuation  $\gamma_1$  is prepared as the handler for  $es_2$ . When  $es_1$  is evaluated, both the success and failure continuation are installed. If an exception is thrown, the control is transferred to the failure continuation by invoking  $\gamma$  with the error code on the stack. Similar to the base semantics of `for-loop`, we deliberately leave no interaction with breaks. It is also possible to recover a normal break behavior within  $es_1$  or  $es_2$  by appending the continuation  $\kappa_1$  to the trail  $\theta$ .

The interpretation of other constructs retains the failure continuation  $\gamma$ . For example, in contrast to local trail continuations, which are discarded for function calls, the failure continuation is preserved across function calls, enabling non-local control transfers. Note that the  $es_2$  does not have the ability to resume execution at the point where the exception was thrown, which we will generalize in the next section.

#### 4.4 Resumable Exceptions

Effect handlers [19, 20] are known to be a generalization of exception handling, which allows the handlers to access the (delimited) continuation at the point where the effectful operation was invoked. In this section, we demonstrate a flavor of effect handlers as an extension to WebAssembly, which is higher-level than the existing WasmFX proposal [17]. Based on the development of the previous section, we further introduce a new instruction `resume`, which allows the handler to invoke the resumption. To informally explain the semantics, let us consider the following code snippet:

```

1  try
2    i32.const 1
3    call print
4    i32.const -1 ;; error code
5    throw
6    i32.const 2
7    call print
8  catch
9    ;; stack: [-1, resumption]
10   call print
11   resume ;; back to line 6
12 end

```

The program outputs `1` first, followed by a `throw` instruction, which transfer the control to the `catch` block. When entering the `catch` block (i.e., the handler), the stack contains the error code `-1` and the resumption. The `catch` block prints the error code `-1`, and then resumes the execution back into the `try` block (line 6), printing `2`. Therefore, the whole output of this program is `1 -1 2`.

In the following, we discuss the extension and its CPS semantics under our framework.

**Syntax and Statics** On top of Section 4.3, we add the new `resume` instruction:

$$e \in \text{Instruction} ::= \dots \mid \text{try } es_1 \text{ catch } es_2 \mid \text{throw} \mid \text{resume}$$

The `resume` instruction assumes that the top element of the stack is a resumable continuation. The full treatment of typing is omitted here, but one can follow ideas from from WasmFX [17]. In essence, our extension is equivalent to unlabelled effect handlers (e.g. as in [4]), where handlers only handle a single kind of effect. One can readily read `try  $es_1$  catch  $es_2$`  as `handle  $es_1$  with  $\{x, k \mapsto es_2\}$` , where  $x$  is the error code and  $k$  is the resumption.

**CPS Semantics** Following Danvy and Filinski [6], we extend our semantics with a meta-continuation, which conceptually is the continuation of continuations `Cont`. The notion of meta-continuations is useful to delimit the context when evaluating the `try` block, since when suspended by a `throw` instruction only a portion of the continuation should be captured within the `try` block. Below we show the changes in the semantic domain definitions:

$$\begin{aligned}
\kappa \in \text{Cont} &= \text{Stack} \times \text{Env} \times \text{MCont} \rightarrow \text{Ans} \\
m \in \text{MCont} &= \text{Stack} \times \text{Env} \rightarrow \text{Ans} \\
\gamma \in \text{Handler} &= \text{Stack} \times \text{Env} \rightarrow \text{Ans} \\
v, r \in \text{Value} &::= \dots \mid \text{Stack} \times \text{Env} \times \text{Cont} \times \text{MCont} \rightarrow \text{Ans}
\end{aligned}$$

We also extend the value domain to include resumable continuation values (denoted by  $r$ ), in the sense that they are first-class values<sup>4</sup> that can be stored on the stack or in local variables. The handler expects such a continuation value

<sup>4</sup> Supporting first-class functions is already an ongoing proposal to WebAssembly [31].

$$\begin{aligned}
\llbracket \text{try } es_1 \text{ catch } es_2 \text{ :: } rest \rrbracket(\sigma, \rho, \kappa, \theta, m, \gamma) &= \\
\text{let } m_j &:= \lambda(\sigma_1, \rho_1). \llbracket rest \rrbracket(\sigma, \rho_1, \kappa, \theta, m, \gamma) \text{ in} \\
\text{let } \kappa_1 &:= \lambda(\sigma_1, \rho_1, m_1). m_1(\sigma_1, \rho_1) \text{ in} \\
\text{let } \gamma_1 &:= \lambda(\sigma_2, \rho_2). \llbracket es_2 \rrbracket(\sigma_2, \rho_2, \kappa_1, \theta, m_j, \gamma) \text{ in} \\
&\llbracket es_1 \rrbracket(\llbracket \cdot \rrbracket, \rho, \kappa_1, \theta, m_j, \gamma_1) \\
\llbracket \text{throw :: } rest \rrbracket(v :: \sigma, \rho, \kappa, \theta, m, \gamma) &= \\
\text{let } r &:= \lambda(\sigma_1, \rho_1, \kappa_1, m_1). \llbracket rest \rrbracket(\sigma, \rho_1, \kappa \oplus \kappa_1, \theta, m_1, \gamma) \text{ in} \\
&\gamma([v, r], \rho) \\
\llbracket \text{resume :: } rest \rrbracket(r :: \sigma, \rho, \kappa, \theta, m, \gamma) &= \\
&r(\llbracket \cdot \rrbracket, \rho, \lambda(\sigma_1, \rho_1, m_1). \llbracket rest \rrbracket(\sigma, \rho_1, \kappa, \theta, m_1, \gamma), m)
\end{aligned}$$

Fig. 4: CPS semantics (excerpt) for resumable exceptions.

on the stack. With the changes in the domain definitions, the signature of the semantic function is the following:

$$\llbracket \cdot \rrbracket : \text{List}[\text{Inst}] \rightarrow (\text{Stack} \times \text{Env} \times \text{Cont} \times \text{Trail} \times \text{MCont} \times \text{Handler}) \rightarrow \text{Ans}$$

Figure 4 shows the CPS semantics for resumable exceptions. When evaluating a `try-catch` instruction, `rest` along with its continuation  $\kappa$  serves as the join point for both the success control flow (i.e., no exception is thrown during the dynamic extent of  $es_1$ ) and the failure control flow via the handler  $es_2$ . We define a meta-continuation  $m_j$  for this join point. The handler  $\gamma_1$  evaluates  $es_2$  with the *identity* continuation  $\kappa_1$  and the meta-continuation  $m_j$ . The identity continuation essentially only composes its meta-continuation. Finally, the evaluation of  $es_1$  is delimited by the identity continuation, under the meta-continuation and handler.

When throwing an exception, we define the resumption value  $r$  for evaluating `rest`, which is parameterized by a new continuation  $\kappa_1$  and meta-continuation  $m_1$ . To resume the evaluation, we compose the current continuation with the new continuation  $\kappa \oplus \kappa_1$ , which is defined as  $\lambda(\sigma_1, \rho_1, m). \kappa(\sigma_1, \rho_1, \lambda(\sigma_2, \rho_2). \kappa_1(\sigma_2, \rho_2, m))$ . Note that we discard the current meta-continuation  $m$  (of `throw :: rest`), since once the exception is thrown, the success control flow is no longer needed. The same handler  $\gamma$  is installed for the resumption of `rest`, reflecting the deep handler semantics. This resumption continuation  $r$  is reified as a proper first-class value (but opaque to the user) and pushed onto the stack, along with the error code  $v$ . Lastly, we call the handler with that stack and the environment.

The program can choose to resume the computation to the point where the exception was thrown via the `resume` instruction in the catch block<sup>5</sup>. The `resume` instruction expects that a continuation value  $r$  is also on top of the stack. To

<sup>5</sup> Although the resumption can escape via assignments.

invoke the resumption, we provide an empty stack (which is nonetheless irrelevant since  $\kappa_r$  will not use it), the current environment, and the continuation to finish the handler.

**Remarks** Our presentation of effect handlers to  $\mu$ Wasm is both simpler and higher-level compared to the existing WasmFX proposal [17], albeit it only support unlabelled effects. Tags in WasmFX also allows more values to be passed to the handler instead of an error code. The handler clauses in WasmFX are specified by block labels instead of a block of instructions as in our extension. WasmFX’s design combines handling and resumption, leading to a combination of deep handler and shallow handler, in the sense that the handler is not re-installed for the resumption, but an invocation of resumptions/continuations much be wrapped within some handler. In contrast, our CPS semantics simply follows the deep handler semantics. Different from WasmFX that is only designed for one-shot continuations, we do not restrict the number of times a resumption can be invoked.

Our current framework should be expressive enough to support WasmFX-like effect handlers with slight modification. We leave this as future work.

## 5 Implementation & Validation

Following the core CPS semantics for  $\mu$ Wasm (Figure 3), we have implemented a big-step interpreter in Scala for standard WebAssembly. A textual WebAssembly file (`.wat`) is parsed to a module using ANTLR, and then fed to the interpreter for evaluation. The following code snippet shows the core definitions and the main structure of the evaluation function:

```

type Stack = List[Value]
type Env = Map[Int, Value]
type Cont[A] = List[Value]  $\Rightarrow$  A
type Trail = List[Cont[Ans]]

def eval[Ans](insts: List[Inst], stack: Stack, env: Env,
              k: Cont[Ans], trail: Trail): Ans =
  insts match
  case Nil  $\Rightarrow$  k(stack, env)
  case Binary(op)  $\Rightarrow$ 
    val v2 :: v1 :: newStack = stack
    val result = evalBinOp(op, v1, v2)
    eval(rest, result :: newStack, env, k, trail)
  ... // more instructions

```

Compared to the minimal model language  $\mu$ Wasm, our interpreter supports both more top-level definitions (e.g. type definitions, tables, etc.) and more instructions (e.g. memory operations, global variables, etc.). We have also experimentally implemented the extensions discussed in Section 4.1, Section 4.2, Section 4.4.

Although we have not yet implemented the complete set of instructions in WebAssembly, we have validated the control-flow semantics of the core instruc-

tions, ensuring that they behave as same as other Wasm runtimes (e.g. Wasmtime, Wizard). One ongoing work is to extend the interpreter with a larger set of instructions and to validate the full semantics against the official WebAssembly specification test suite [33].

## 6 Related Work

**Semantics of WebAssembly** The WebAssembly specification [11, 32] defines a reference small-step reduction semantics, which has been used as the basis for various extensions and new developments. Besides the presented work, there are a few other works on defining big-step semantics for WebAssembly. Watt et al. [24] develop a big-step evaluation relation used in the soundness proof for Wasm Logic. Instead of using continuations, the big-step evaluation relation still makes use of first-order representations for the evaluation of `block` and `loop` (e.g., a list of labels as the evaluation context). Watt et al. [25] further develop a monadic interpreter as the oracle for fuzzing WebAssembly programs. The monadic interpreter (along with an intermediate interpreter) is written in Isabelle/HOL, and similarly uses a first-order representation of evaluation context. Following the abstracting definitional interpreter approach [10], Brandl et al. [3] develop a big-step abstract definitional interpreter for WebAssembly, which by using different monads can be instantiated for different purposes, such as concrete interpretation or taint analysis. The definitional interpreter uses exceptions from the host language (Scala, in their case) to implement breaks and return. To the best of our knowledge, our work is the first to reconstruct a big-step, compositional, semantics for WebAssembly using continuations.

**Control Abstractions** Phipps-Costin et al. [17] proposed WasmFX, an extension for WebAssembly with typed delimited continuations and effect handlers. As an active proposal, the formal reduction semantics of WasmFX [26] is specified in the same style as the official WebAssembly specification, both of which make use of administrative instructions to handle control flow. In Section 4.4, we have discussed the major design difference between our extension and WasmFX. Our semantics of effect handlers is inspired by the CPS semantics of `shift/reset` [6]. Similar to our account of effect handlers, Hillerström et al. [12, 13] have study the CPS translation of effect handlers for a fine-grained call-by-value  $\lambda$ -calculus.

The extensions of `for-loop` (Section 4.1) and non-resumable exceptions (Section 4.3) are inspired by the Stanford CS242 course [22]. Although relatively straightforward, we believe these examples are pedagogically valuable in demonstrating their CPS semantics.

## 7 Conclusion and Future Work

We have presented a big-step, compositional, continuation-passing semantics for WebAssembly and demonstrated its application to several control abstraction extensions. We hope that the CPS semantics will serve as a useful reference for

developing additional tools for WebAssembly, such as partial evaluators [15], compiler optimizations [1], and various program analysis frameworks.

While we use a test suite to validate the correctness of our CPS semantics, formally establishing its correctness or proving its equivalence to the reference semantics remains a task for future work. One could conjecture that the reference reduction semantics might be transformed into a big-step CPS semantics through refunctionalization [8] (and conversely, to the original form via defunctionalization [9]). Additionally, it would be interesting to transform the CPS semantics back to direct style [5, 7, 16], making use of control operators in the meta-language. We leave these explorations for future work.

## A Auxiliary Definitions

$$\begin{aligned} \text{buildEnv} &: (\text{Stack} \times \text{List}[\text{Value}]) \rightarrow \text{Env} \\ \text{buildEnv}(\sigma, vs) &= \text{reverse}(\sigma) \text{ ++ } \text{default}(vs) \\ \text{default} &: \text{List}[\text{ValueType}] \rightarrow \text{List}[\text{Value}] \\ \text{default}(\[]) &= [] \\ \text{default}(t :: ts) &= 0_t :: \text{default}(ts) \end{aligned}$$

## B Recovering Break and Continue Semantics of `br` in `for-Loop`

Section 4.1 discussed the semantics of the structured `for-loop`, which has no interaction with `br` instructions. Here, we show how to recover the continue and break semantics of `br` in the loop body. Important changes compared to Section 4.1 are highlighted in red.

### ***br as Continue in for-Loop:***

$$\begin{aligned} \llbracket \text{for } (es_{\text{init}}; es_{\text{cond}}; es_{\text{post}}) es :: rest \rrbracket(\sigma, \rho, \kappa, \theta) = \\ \text{fix } \kappa_1 := \lambda(\sigma_1, \rho_1). \llbracket es_{\text{cond}} \rrbracket(\rho_1, \lambda(v :: \sigma_2, \rho_2). \\ \text{if } v \equiv 0 \text{ then } \llbracket rest \rrbracket(\sigma, \rho_2, \kappa, \theta) \\ \text{else } \llbracket es \rrbracket(\rho_2, \lambda(\sigma_3, \rho_3). \llbracket es_{\text{post}} \rrbracket(\rho_3, \kappa_1, \theta), \kappa_1 :: \theta)) \text{ in} \\ \llbracket es_{\text{init}} \rrbracket(\rho, \kappa_1, \theta) \end{aligned}$$

### ***br as Break in for-Loop:***

$$\begin{aligned} \llbracket \text{for } (es_{\text{init}}; es_{\text{cond}}; es_{\text{post}}) es :: rest \rrbracket(\sigma, \rho, \kappa, \theta) = \\ \text{let } \kappa_0 := \lambda(\sigma_1, \rho_1). \llbracket rest \rrbracket(\sigma, \rho_1, \kappa, \theta) \text{ in} \\ \text{fix } \kappa_1 := \lambda(\sigma_2, \rho_2). \llbracket es_{\text{cond}} \rrbracket(\rho_2, \lambda(v :: \sigma_3, \rho_3). \\ \text{if } v \equiv 0 \text{ then } \kappa_0(\sigma, \rho_3) \\ \text{else } \llbracket es \rrbracket(\rho_3, \lambda(\sigma_4, \rho_4). \llbracket es_{\text{post}} \rrbracket(\rho_4, \kappa_1, \theta), \kappa_0 :: \theta)) \text{ in} \\ \llbracket es_{\text{init}} \rrbracket(\rho, \kappa_1, \theta) \end{aligned}$$



## Bibliography

1. Appel, A.W.: *Compiling with Continuations*. Cambridge University Press (1992)
2. Biernacki, D., Danvy, O., Millikin, K.: A dynamic continuation-passing style for dynamic delimited continuations. *ACM Trans. Program. Lang. Syst.* **38**(1), 2:1–2:25 (2015)
3. Brandl, K., Erdweg, S., Keidel, S., Hansen, N.: Modular abstract definitional interpreters for webassembly. In: *ECOOP, LIPIcs*, vol. 263, pp. 5:1–5:28, Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2023)
4. Cong, Y., Asai, K.: Understanding algebraic effect handlers via delimited control operators. In: *TFP, Lecture Notes in Computer Science*, vol. 13401, pp. 59–79, Springer (2022)
5. Danvy, O.: Back to direct style. In: *ESOP, Lecture Notes in Computer Science*, vol. 582, pp. 130–150, Springer (1992)
6. Danvy, O., Filinski, A.: Abstracting control. In: *LISP and Functional Programming*, pp. 151–160, ACM (1990)
7. Danvy, O., Lawall, J.L.: Back to direct style II: first-class continuations. In: *LISP and Functional Programming*, pp. 299–310, ACM (1992)
8. Danvy, O., Millikin, K.: Refunctionalization at work. *Sci. Comput. Program.* **74**(8), 534–549 (2009)
9. Danvy, O., Nielsen, L.R.: Defunctionalization at work. In: *PPDP*, pp. 162–174, ACM (2001)
10. Darais, D., Labich, N., Nguyen, P.C., Horn, D.V.: Abstracting definitional interpreters (functional pearl). *Proc. ACM Program. Lang.* **1**(ICFP), 12:1–12:25 (2017)
11. Haas, A., Rossberg, A., Schuff, D.L., Titzer, B.L., Holman, M., Gohman, D., Wagner, L., Zakai, A., Bastien, J.F.: Bringing the web up to speed with webassembly. In: *PLDI*, pp. 185–200, ACM (2017)
12. Hillerström, D., Lindley, S., Atkey, R.: Effect handlers via generalised continuations. *J. Funct. Program.* **30**, e5 (2020)
13. Hillerström, D., Lindley, S., Atkey, R., Sivaramakrishnan, K.C.: Continuation passing style for effect handlers. In: *FSCD, LIPIcs*, vol. 84, pp. 18:1–18:19, Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2017)
14. Jones, N.D.: What not to do when writing an interpreter for specialisation. In: *Dagstuhl Seminar on Partial Evaluation, Lecture Notes in Computer Science*, vol. 1110, pp. 216–237, Springer (1996)
15. Lawall, J.L., Danvy, O.: Continuation-based partial evaluation. In: *LISP and Functional Programming*, pp. 227–238, ACM (1994)
16. Müller, M., Schuster, P., Brachthäuser, J.I., Ostermann, K.: Back to direct style: Typed and tight. *Proc. ACM Program. Lang.* **7**(OOPSLA1), 848–875 (2023)
17. Phipps-Costin, L., Rossberg, A., Guha, A., Leijen, D., Hillerström, D., Sivaramakrishnan, K.C., Pretnar, M., Lindley, S.: Continuing WebAssembly

- with Effect Handlers. *Proc. ACM Program. Lang.* **7**(OOPSLA2), 460–485 (2023)
18. Plotkin, G.D.: A structural approach to operational semantics. *J. Log. Algebraic Methods Program.* **60-61**, 17–139 (2004)
  19. Plotkin, G.D., Pretnar, M.: Handlers of algebraic effects. In: *ESOP, Lecture Notes in Computer Science*, vol. 5502, pp. 80–94, Springer (2009)
  20. Plotkin, G.D., Pretnar, M.: Handling algebraic effects. *Log. Methods Comput. Sci.* **9**(4) (2013)
  21. Reynolds, J.C.: Definitional interpreters for higher-order programming languages. In: *ACM Annual Conference (2)*, pp. 717–740, ACM (1972)
  22. Stanford University: CS242: Programming Languages (Fall 2019, Will Crichton) - Assignment 5 (2019), URL <https://stanford-cs242.github.io/f19/assignments/assign5/>, accessed: 2024-10-28
  23. Thielecke, H.: Comparing control constructs by double-barrelled CPS. *High. Order Symb. Comput.* **15**(2-3), 141–160 (2002)
  24. Watt, C., Maksimović, P., Krishnaswami, N.R., Gardner, P.: A Program Logic for First-Order Encapsulated WebAssembly. In: *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*, Leibniz International Proceedings in Informatics (LIPIcs), vol. 134, pp. 9:1–9:30, Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2019)
  25. Watt, C., Trela, M., Lammich, P., Märkl, F.: WasmRef-Isabelle: A verified monadic interpreter and industrial fuzzing oracle for webassembly. *Proc. ACM Program. Lang.* **7**(PLDI), 100–123 (2023)
  26. WebAssembly Contributors: WasmFX Specification. <https://wasmfx.dev/specs/core/> (2024), accessed: 2024-11-11
  27. WebAssembly Contributors: Webassembly core specification: Runtime stack. <https://webassembly.github.io/spec/core/exec/runtime.html#stack> (2024), accessed: 2024-10-17
  28. WebAssembly Contributors: WebAssembly Proposal: Exceptions (2024), URL <https://github.com/WebAssembly/exception-handling/blob/main/proposals/exception-handling/Exceptions.md>, accessed: 2024-09-25
  29. WebAssembly Contributors: WebAssembly Proposal: Tail-Call (2024), URL <https://github.com/WebAssembly/tail-call>, accessed: 2024-10-24
  30. WebAssembly Contributors: WebAssembly Proposal: Type Continuations (2024), URL <https://github.com/WebAssembly/stack-switching/blob/main/proposals/continuations/Explainer.md>, accessed: 2024-09-25
  31. WebAssembly Contributors: WebAssembly Proposal: Typed Function References (2024), URL <https://github.com/WebAssembly/function-references>, accessed: 2024-10-24
  32. WebAssembly Contributors: WebAssembly Specification (2024), URL <https://webassembly.github.io/spec/core/>, accessed: 2024-09-25
  33. WebAssembly Contributors: Webassembly specification testsuite (2024), URL <https://github.com/WebAssembly/spec/tree/main/test/core>, accessed: 2024-11-11