Reconstructing Continuation-Passing Semantics for WebAssembly

Guannan Wei[1,2], Alexander Y. Bai[2,3], Dinghong Zhong[4], and Jiatai Zhang[2]

[1] INRIA/ENS-PSL, Paris, France guannan.wei@inria.fr
[2] Tufts University, Medford, USA
{guannan.wei, alexander.bai, jiatai.zhang}@tufts.edu
[3] MPI-SWS, Saarbrücken, Germany abai@mpi-sws.org
[4] Unaffiliated azhong.934@gmail.com

Abstract. WebAssembly is 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 develops an alternative semantics in continuation-passing style (CPS), which can be implemented as a concise, compositional, and tail-recursive interpreter or CPS transformer. Using continuations from the meta-language eliminates the need for 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 [13] from the outset and has evolved [34] in tandem with the language development.

The co-developed formal specification facilitates not only the development of language tools, but also the design and prototyping of new extensions for Wasm. For example, there are a number of proposals for adding richer control abstractions to Wasm, such as exception handling [30], continuations [32], effect handlers [19], and more, all building on top of the reference semantics.

The specification of Wasm describes a small-step reduction semantics [20], which is a standard approach and straightforward to implement as an interpreter. However, the reduction semantics [13, 34] is complicated by the additional *administrative instructions*, which are extensions to handle control constructs (e.g., loops and breaks) at runtime 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 [29], arguably adding cognitive overhead to understand the semantics. Since control transfer needs to search over the stack, these redundancies lead to inefficiency if just naively implementing the reference interpreter [27].

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 [16], which are useful to derive other tools or optimizations based on the semantics. The lack of compositionality adds complication both in reasoning about the semantics and in developing tools that adhere to the semantics.

This Work Following Reynolds' seminal work of definitional interpreters [23], 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 high-level, compositional, and tail-recursive semantics in CPS that captures the essence of Wasm's control-flow semantics. 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 [13, 34]. Such a CPS semantics can be straightforwardly implemented as a big-step interpreter or CPS transformer in a functional language.

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 [25] 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 [24], tail calls [31], exceptions [24, 30], and effect handlers [19, 21, 22]. Since our CPS semantics is compositional, it facilitates equational reasoning of Wasm programs, for which we demonstrate the calculation of tail call semantics.

Contributions This work makes the following contributions:

- We examine WebAssembly's control-flow semantics by developing a CPS-based semantics à la 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 μWasm, a minimal language following the essence of Wasm. Section 3 presents the core CPS semantics for μWasm. 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 μWasm: A Minimal Language à la WASM

In this section, we introduce a minimal language μWasm, including 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

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

Fig. 1: The abstract syntax of μWasm.

Figure 1 presents the abstract syntax of μWasm. A μWasm module consists of a sequence of function definitions. Each function defines its type, identifier, a list of types for 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.

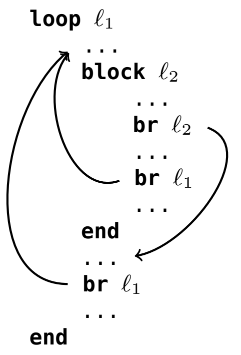
Control flow constructs in Wasm are structured and can be nested. For example, block-like constructs (block, loop, if) contain sequences of instructions.

They also take a function type regulating the shape of the stack before and after the block. Block-like constructs introduce an implicit numeric label à la de Bruijn indices, which can be referred by br within the block. Break instruction (br) takes a label, referring to the jump target.

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 symbols. 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 μWasm as in standard WebAssembly. We also assume that our definitional interpreter takes well-typed WebAssembly programs as input.

2.2 Example

Although WebAssembly looks like a standard stack-based low-level language, one of the uncommon features of WebAssembly is that the semantics of a break contextually depends on its target enclosing block. For example, the following code snippet defines a loop block containing a regular block, labeled ℓ_1 and ℓ_2 respectively



```
loop ℓ1
    ...
    block ℓ2
        ...
        br ℓ2
        ...
        br ℓ1
        ...
    end
    ...
    br ℓ1
    ...
end
```

There are three br instructions in this 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 behavior.

3 A CPS Semantics for μWasm

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

Fig. 2: Semantic domains and auxiliary functions.

Evaluation function: $[\![\cdot]\!] : \mathsf{List}[\mathsf{Inst}] \to (\mathsf{Stack} \times \mathsf{Env} \times \mathsf{Cont} \times \mathsf{Trail}) \to \mathsf{Ans}$

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

Fig. 3: The continuation-passing style semantics of μWasm.

We now present a continuation-passing semantics for μWasm. The semantics $[\![\cdot]\!]$ is defined as a recursive function mapping a *list of instructions* to functions with continuations:

$$[\![\cdot]\!] : \mathsf{List[Inst]} \rightarrow (\mathsf{Stack} \times \mathsf{Env} \times \mathsf{Cont} \times \mathsf{Trail}) \rightarrow \mathsf{Ans}$$

In the following, we first explain the domain definitions and notations, then present the definition of $[\![\cdot]\!]$ in Figure 3. We also discuss an alternative formulation and future work that bridges the gap between our core CPS semantics and the full Wasm language.

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) ρ is represented as a list of values, mapping numeric-represented local variables as indices to values. We represent continuations κ 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 θ 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 θ (representing Stack, Env, or Trail), $\theta(\ell)$ accesses the ℓ-th element in the list θ. $\sigma_1 +\!\!+ \sigma_2$ concatenates two stacks σ_1 and σ_2. $\rho[x \mapsto v]$ updates the environment ρ 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 σ, $\lfloor \sigma \rfloor_m$ takes the top m elements from the stack σ, which are of type t^m guaranteed by Wasm's type system. This is useful, e.g., for truncating 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 $\sigma_1 \,{}_m\!+\!\!+\, \sigma_2$ to split the stack into two parts, where the first part σ_1 has m values, and the second part σ_2 contains the rest of the stack.

3.2 CPS Semantics

Figure 3 presents the CPS semantics of μWasm, 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 the immediately enclosing block, (2) breaking out of a block, and (3) returning from the current function. Therefore, to accommodate such control flow, $[\![\cdot]\!]$ takes the following additional control arguments:

– a continuation function κ representing the control-flow after consuming the current list of instructions (i.e. the case for $[\![nil]\!]$, yielding to the enclosing block), and
– a trail of continuations θ representing the control-flow for breaking (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, μWasm models three kinds of block-like structures (block, loop, and if) which introduce labels that can be used by 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 brs 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 θ to keep track of the lexical control-flow structure, then labels of brs 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 κ_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 θ). Similarly, for a if instruction, we prepare the same kind of continuation, but choose to 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) κ_1 represents the fall-through continuation that evaluates the rest of the instructions syntactically after the loop, and (2) κ_2 represents the break-out continuation that jumps back to the beginning of the loop. Note that κ_2 is defined as a fixed-point, so that it can be recursively pushed onto the trail θ within its definition. Lastly, we initiate the evaluation of the loop body by invoking κ_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 $\lfloor \sigma_1 \rfloor_n$ to truncate the output stack for the return values.

Call and Return For function calls, we use an auxiliary function $\mathsf{lookupFunc}$ to find the definition (i.e., its function type, local variable types, and body) by the function index. Our presentation omits the definition of $\mathsf{lookupFunc}$, as it is straightforward to implement and not the focus of this paper. Another auxiliary function $\mathsf{buildEnv}$ (Appendix A shows its definition) constructs a new environment ρ_1 containing the arguments. The environment has also pre-allocated default values for local variables. The continuation κ_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, and trail.

When we enter the function body es, we use an empty stack, the new environment ρ_1, and the return continuation κ_1. Since a function introduces a *fresh* block scope too, the trail θ is a just singleton list containing the return continuation κ_1, discarding all other jump targets that are non-local to the function. Within a function body, this continuation κ_1 will always be the last one in the trail, and will be 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 $[\![\cdot]\!]$ 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 $[\![\cdot]\!]$ 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 μWasm.

Single-Instruction Semantics Notice that our evaluation function takes a list of instructions as an argument. Alternatively, we could define it as only taking a single instruction instead of a list of instructions:

$$[\![\cdot]\!]_{\mathbb{1}} : \mathsf{Inst} \to (\mathsf{Stack} \times \mathsf{Env} \times \mathsf{Cont} \times \mathsf{Trail}) \to \mathsf{Ans}$$

For example, the semantics for add would be

$$[\![t.\mathsf{add}]\!]_{\mathbb{1}}(v_1 :: v_2 :: \sigma, \rho, \kappa, \theta) = \kappa(v_1 + v_2 :: \sigma, \rho).$$

In this alternative semantics, we would 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 κ of the current instruction.

The evaluation function presented in Figure 3 can be considered as the fused version of the single-instruction evaluation and the iterating driver. Or, inversely, we can mechanically derive the single-instruction evaluation from the fused version by equational reasoning, i.e., unfolding the evaluation of an *singleton* instruction list:

$$[\![e :: nil]\!](\sigma, \rho, \kappa, \theta) = [\![e]\!]_{\mathbb{1}}(\sigma, \rho, \kappa, \theta)$$

We choose to only present the fused evaluation function in Figure 3, since it suits well the recursive structures of the language, such as block and loop.

Scaling to Full WebAssembly Many features of WebAssembly, such as memory, global variables, and tables for function references, are orthogonal to control flow semantics. They are omitted in our semantics but nevertheless can be added on top of it, following the official 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 μWasm. In this section, we study how the CPS semantics can be extended to support additional control abstractions. 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 μWasm. 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 [24].

Syntax and Statics The syntax of μWasm is extended with the following construct:

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

The initialization instructions es_{init} run only once before the loop begins. The condition es_{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_{post} are executed (every time) after the loop body es. As for typing, only the condition es_{cond} has type $[] \rightarrow [\mathsf{i32}]$; all other constructs 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:

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

Similar to the semantics of loop-block, we use a fixed-point to define continuation κ_1 that evaluates the condition and possibly the loop body followed by the post-instructions. This continuation κ_1 is used as the continuation when recursively applying the semantic function to es_{post}.

Note that in the above definition, we have deliberately left out the branch semantics in for-loop, i.e. a br 0 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 C-style continue or break. For example, we can append κ_1 to the trail continuation θ when evaluating the loop body es, and use it as the target for a br 0. Similarly, to implement a break behavior, we can append the evaluation of *rest* to the trail θ. 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 tail calls ensure constant stack space consumption. A recent proposal [31] 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 [31] 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:

$$e \in \mathsf{Instruction} ::= \cdots \mid \mathsf{return\_call}\ x$$

Here, return_call x represents a tail call to the function at index x. We omit the typing rules, which follows the proposal [31].

CPS Semantics As its name suggests, the instruction return_call combines the semantics of return and call, performing them in a single step. In the CPS semantics for standard call/return, 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:

$$\begin{aligned}
&[\![\mathsf{return\_call}\ x :: \mathit{rest}]\!](\sigma_{arg\ m} + \sigma, \rho, \kappa, \theta) = \\
&\quad \mathsf{let}\ \{\mathsf{type} : t^m \to t^n, \mathsf{locals} : ts, \mathsf{body} : es\} \coloneqq \mathsf{lookupFunc}(x)\ \mathsf{in} \\
&\quad \mathsf{let}\ \rho_1 \coloneqq \mathsf{buildEnv}(\sigma_{arg}, ts) \\
&\quad [\![es]\!]([], \rho_1, \theta.\mathsf{last}, [\theta.\mathsf{last}])
\end{aligned}$$

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 now we are at a tail position, es does not need to return to the current context (i.e. continue to *rest*), but instead can be optimized to return to the caller of the current context. Therefore, when evaluating the body of the callee, we directly use the the return continuation $\theta.\mathsf{last}$.

Calculation Although it is straightforward to directly define the semantics of tail calls in CPS, we show an alternative way to *derive* the semantics of return_call by calculation. The derivation is possible because the semantics is compositional, and justified by equational reasoning.

Since the semantics of return_call is a synthesis of return and call, our starting point is their syntactic combination. We syntactically construct a call at a tail position, and calculate the semantics by unfolding $[\![\cdot]\!]$ on call and return:

$$\begin{aligned}
&\quad [\![\mathsf{call}\ x :: \mathsf{return} :: \mathit{rest}]\!](\sigma_{arg\ m} + \sigma, \rho, \kappa, \theta) \\
&= \{\mathit{unfold}\ \mathsf{call}\} \\
&\quad \mathsf{let}\ \{\mathsf{type} : t^m \to t^n, \mathsf{locals} : ts, \mathsf{body} : es\} \coloneqq \mathsf{lookupFunc}(x)\ \mathsf{in} \\
&\quad \mathsf{let}\ \rho_1 \coloneqq \mathsf{buildEnv}(\sigma_{arg}, ts) \\
&\quad \mathsf{let}\ \kappa_1 \coloneqq \lambda(\sigma_1, \rho_1).[\![\mathsf{return} :: \mathit{rest}]\!](\lfloor \sigma_1 \rfloor_n + \sigma, \rho, \kappa, \theta)\ \mathsf{in} \\
&\quad [\![es]\!]([], \rho_1, \kappa_1, [\kappa_1]) \\
&= \{\mathit{unfold}\ \mathsf{return}\} \\
&\quad \mathsf{let}\ \{\mathsf{type} : t^m \to t^n, \mathsf{locals} : ts, \mathsf{body} : es\} \coloneqq \mathsf{lookupFunc}(x)\ \mathsf{in} \\
&\quad \mathsf{let}\ \rho_1 \coloneqq \mathsf{buildEnv}(\sigma_{arg}, ts) \\
&\quad \mathsf{let}\ \kappa_1 \coloneqq \lambda(\sigma_1, \rho_1).\theta.\mathsf{last}(\lfloor \sigma_1 \rfloor_n + \sigma, \rho)\ \mathsf{in} \\
&\quad [\![es]\!]([], \rho_1, \kappa_1, [\kappa_1]) \\
&= \{\kappa_1\ \mathit{is}\ \eta\text{-}\mathit{equivalent\ to}\ \theta.\mathsf{last},\ \mathit{inlining}\ \kappa_1\} \\
&\quad \mathsf{let}\ \{\mathsf{type} : t^m \to t^n, \mathsf{locals} : ts, \mathsf{body} : es\} \coloneqq \mathsf{lookupFunc}(x)\ \mathsf{in} \\
&\quad \mathsf{let}\ \rho_1 \coloneqq \mathsf{buildEnv}(\sigma_{arg}, ts)\ \mathsf{in} \\
&\quad [\![es]\!]([], \rho_1, \theta.\mathsf{last}, [\theta.\mathsf{last}]) \\
&= \{\mathit{definition\ of}\ \mathsf{return\_call}\} \\
&\quad [\![\mathsf{return\_call}\ x :: \mathit{rest}]\!](\sigma_{arg\ m} + \sigma, \rho, \kappa, \theta)
\end{aligned}$$

In the third step, the replacement of κ_1 with $\theta.\mathsf{last}$ needs additional justification. It amounts to show that $\lambda(\sigma_1, \rho_1).\theta.\mathsf{last}(\lfloor \sigma_1 \rfloor_n + \sigma, \rho)$ is behaviorally equivalent to $\theta.\mathsf{last}$. First notice that the return type t^n must be the same for function x and the current function that returns to $\theta.\mathsf{last}$, therefore the caller-provided $\theta.\mathsf{last}$ itself will truncate the stack $\lfloor\sigma_1\rfloor_n + \!\!+ \sigma$ by the top n elements too. In other words, $\lfloor\lfloor\sigma_1\rfloor_n + \!\!+ \sigma\rfloor_n$ is equivalent to $\lfloor\sigma_1\rfloor_n$, so we can safely replace it with σ_1. Second, the caller-provided $\theta.\mathsf{last}$ uses the environment from its own caller frame (recall Figure 3), therefore it does not matter whether we pass ρ or ρ_1 here. After these reasoning steps, we can show that κ_1 is indeed behaviorally equivalent to $\theta.\mathsf{last}$, thus can be η-reduced and inlined.

Finally, we derive the same semantics for return_call as we have defined in the previous section.

4.3 Exceptions

In standard WebAssembly and μWasm, the jump target ℓ in a break instruction br ℓ is entirely static and local within a function. It is not possible to jump out of nested functions 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 [30]. Without introducing heavy mechanisms (e.g. a table describing multiple handlers as in the proposal), we choose to demonstrate a higher-level, hypothetical try/catch construct, which is close to the description in one of the assignment problems in Stanford's CS242 course [24]. 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:

$$e \in \mathsf{Instruction} ::= \cdots \mid \mathsf{try}\ es_1\ \mathsf{catch}\ es_2 \mid \mathsf{throw}$$

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 $[\![\cdot]\!]$ with an additional failure continuation [25] (or, handler), denoted as γ argument of type Cont:

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

The success continuation κ_1 is prepared to evaluate the rest of the instructions after the try-catch block. The failure continuation γ_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 γ 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 κ_1 to the trail θ.

The interpretation of other constructs retains the failure continuation γ. 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 [21, 22] are known to be a generalization of exception handling. They enable handlers to access the (delimited) continuation at the point where the effectful operation is 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 [19]. 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 transfers 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:

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

The resume instruction assumes that the top element of the stack is a resumable continuation. In essence, our extension is equivalent to unlabelled effect handlers (e.g. as in [5]), where handlers only handle a single kind of effects. One can readily read $\mathsf{try}\ es_1\ \mathsf{catch}\ es_2$ as $\mathsf{handle}\ es_1\ \mathsf{with}\ \{x, k \mapsto es_2\}$, where x is the error code and k is the resumption.

We do not intend to develop a type system for this extension, but one can borrow some ideas from from WasmFX [19]. One caveat is that we disallow using br in the try/catch block, since invoking br in the (possibly escaped) delimited continuation is ill-defined.

CPS Semantics Following Danvy and Filinski [7], we extend our semantics with a meta-continuation, which conceptually is the continuation of continuations Cont. The notion of meta-continuations is useful for delimiting the context when evaluating the try block, since when suspended by a throw instruction only the continuation within the try block should be captured. Below we show the changes in the semantic domain definitions:

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

We also extend the value domain to include resumable continuation values (denoted by r), in the sense that they are first-class values[5] that can be stored on the stack or in local variables. The resume instruction expects such a continuation value on the stack. With the changes in the domain definitions, the signature of the semantic function is the following:

$$[\![\cdot]\!] : \mathsf{List[Inst]} \to (\mathsf{Stack} \times \mathsf{Env} \times \mathsf{Cont} \times \mathsf{Trail} \times \mathsf{MCont} \times \mathsf{Handler}) \to \mathsf{Ans}$$

Figure 4 shows the CPS semantics for resumable exceptions. When evaluating a try-catch instruction, *rest* along with its continuation κ is delayed as a meta-continuation m_1. This meta-continuation serves as the join point for both the success control flow (i.e., no exception is thrown during the dynamic extent of

[5] Supporting first-class functions is already an ongoing proposal to WebAssembly [33].

$$
\begin{aligned}
&[\![\textsf{try}\ es_1\ \textsf{catch}\ es_2 :: rest]\!](\sigma, \rho, \kappa, \theta, m, \gamma) = \\
&\quad \textsf{let}\ m_1 := \lambda.(\sigma_1, \rho_1).[\![rest]\!](\sigma_1, \rho_1, \kappa, \theta, m, \gamma) \\
&\quad \textsf{let}\ \gamma_1 := \lambda(\sigma_1, \rho_1, \kappa_1, m_1).[\![es_2]\!](\sigma_1, \rho_1, \kappa_1, [], m_1, \gamma)\ \textsf{in} \\
&\quad [\![es_1]\!]([], \rho, \kappa_0, [], m_1, \gamma_1) \\
&[\![\textsf{throw} :: rest]\!](v :: \sigma, \rho, \kappa, \theta, m, \gamma) \qquad = \\
&\quad \textsf{let}\ r := \lambda(\sigma_1, \rho_1, m_1, \gamma_1).[\![rest]\!](\sigma_1, \rho_1, \kappa, \theta, m_1, \gamma)\ \textsf{in} \\
&\quad \gamma([v, r], \rho, \kappa_0, m) \\
&[\![\textsf{resume} :: rest]\!](r :: \sigma, \rho, \kappa, \theta, m, \gamma) \qquad = \\
&\quad \textsf{let}\ m_1 := \lambda.(\sigma_1, \rho_1).[\![rest]\!](\sigma_1, \rho_1, \kappa, \theta, m, \gamma) \\
&\quad r([], \rho, m_1, \gamma)
\end{aligned}
$$

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

es_1) and the failure control flow via the handler es_2. The handler γ_1 evaluates es_2 with the latest continuation and meta-continuation, but with the current handler γ. Finally, the evaluation of es_1 is delimited by the identity continuation κ_0 (defined as $\lambda(\sigma, \rho, m).m(\sigma, \rho)$), with the new meta-continuation and handler.

When throwing an exception, we define the resumption continuation r to evaluate *rest* and its continuation κ, parameterized by the new meta-continuation m_1. The same handler γ 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 identity continuation κ_0.

The program can choose to resume to the point where the exception was thrown via the resume instruction in the catch block [6]. The resume instruction expects that a continuation value r is also on top of the stack. To invoke the resumption, we provide an empty stack (which is nonetheless irrelevant since r will not use it), the current environment, and a new meta-continuation m_1.

Towards WasmFX-Style Effect Handlers Our presentation of effect handlers to μWasm is both simpler and higher-level compared to the existing WasmFX proposal [19], as we 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 single block of instructions as in our extension. WasmFX's design combines handling and resumption, leading to a combination of deep handler and shallow handler: the same handler is not reinstalled for the resumption, but any invocation of resumptions/continuations must be wrapped within some handler. In contrast, our CPS semantics simply follows the deep handler semantics. Different from

[6] Although the resumption can escape via assignments.

WasmFX that is designed for one-shot continuations, we do not restrict the number of times a resumption can be invoked.

Implementing WasmFX-style effect handlers in our framework is possible, but would require a trail of continuations (instead of a meta-continuation), which supports more flexible manipulation of the context. We leave the full formalization of this as future work.

5 Implementation

The core CPS semantics presented in Figure 3 can be viewed as either a big-step definitional interpreter for μWasm, or a translation from μWasm to λ-calculus with some functional data structures (e.g. lists).

We have implemented a big-step interpreter in Scala for standard WebAssembly [7]. 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] ⇒ 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 ⇒ k(stack, env)
    case Binary(op) ⇒
      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 μWasm, our interpreter supports more top-level definitions (e.g. type definitions, tables, etc.) and 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 full set of WebAssembly instructions, we have validated the control-flow semantics of the core instructions, ensuring that they behave consistently with other Wasm runtimes (e.g., Wasmtime, Wizard). In the future, we would like to further extend the interpreter with a larger set of instructions and validate its full semantics against the official WebAssembly specification test suite [35].

6 Related Work

Semantics of WebAssembly The WebAssembly specification [13, 34] defines a reference small-step reduction semantics, which has been used as the basis for

[7] Available at `https://github.com/Generative-Program-Analysis/wasm-cps`

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. [26] 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. [27] 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 [11], Brandl et al. [4] 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. Unlike ours that uses a list of continuations, Brandl et al. [4]'s definitional interpreter uses exceptions in Scala to encode 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. [19] proposed WasmFX, an extension for WebAssembly with typed delimited continuations and effect handlers. As an active proposal, the formal reduction semantics of WasmFX [28] 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 [7]. Similar to our account of effect handlers, Hillerström et al. [14, 15] have studied the CPS translation of effect handlers for a fine-grained call-by-value λ-calculus.

The extensions of for-loop (Section 4.1) and non-resumable exceptions (Section 4.3) are inspired by the Stanford CS242 course [24]. Although relatively straightforward, we believe these examples are pedagogically valuable in demonstrating their CPS semantics.

7 Conclusion and Future Work

We have presented a compositional, tail-recursive, and 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 interpreters (as we showed in this paper), partial evaluators [3, 17], compiler optimizations [1], and program analyses [36] informed by concrete semantics.

While we use a test suite to validate the correctness of our CPS semantics, formally proving its correctness or establishing its equivalence to the reference semantics remains a task for future work. One could conjecture that the reference reduction semantics can be transformed into a CPS semantics through refunctionalization [9] – and conversely, from the CPS semantics to its original form via defunctionalization [10, 12]. Additionally, it would be interesting to transform the CPS semantics back to direct style [6, 8, 18], leveraging control operators in the meta-language. Given the correspondence between different formalizations of semantics, it is plausible that the CPS semantics could be mechanically derived from a single source of truth, such as SpecTec [37]. We leave these explorations for future work.

Acknowledgments. We would like to thank the reviewers of TFP for their insightful and constructive feedback, some of which has been incorporated into this revision of the paper. We also extend our gratitude to Andreas Rossberg, Elizabeth Gilbert, Ben Titzer, and Tiark Rompf for their valuable discussions, and to Mikail Khan for his contributions to the earlier implementations. Additionally, we thank the audiences at TFP and the WAW workshop for their questions and comments. The first author would like to thank Caterina Urban and the ANTIQUE team at INRIA for their support of this work.

A Auxiliary Definitions

We provide the definition of buildEnv that builds the initial environment for a function call according to the stack arguments and the types of local variables.

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

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 C-style "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:

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

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

$$
\begin{array}{l}
[\![\mathsf{for}\ (es_{\mathsf{init}}; es_{\mathsf{cond}}; es_{\mathsf{post}})\ es :: rest]\!](\sigma, \rho, \kappa, \theta) = \\
\quad \mathsf{let}\ \kappa_0 := \lambda(\sigma_1, \rho_1).[\![rest]\!](\sigma, \rho_1, \kappa, \theta)\ \mathsf{in} \\
\quad \mathsf{fix}\ \kappa_1 := \lambda(\sigma_2, \rho_2).[\![es_{\mathsf{cond}}]\!]([], \rho_2, \lambda(v :: \sigma_3, \rho_3). \\
\qquad\qquad\qquad\qquad \mathsf{if}\ v \equiv 0\ \mathsf{then}\ \kappa_0(\sigma, \rho_3) \\
\qquad\qquad\qquad\qquad \mathsf{else}\ [\![es]\!]([], \rho_3, \lambda(\sigma_4, \rho_4).[\![es_{\mathsf{post}}]\!]([], \rho_4, \kappa_1, \theta), \kappa_0 :: \theta))\ \mathsf{in} \\
\quad [\![es_{\mathsf{init}}]\!]([], \rho, \kappa_1, \theta)
\end{array}
$$

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. Bondorf, A.: Improving binding times without explicit cps-conversion. In: LISP and Functional Programming, pp. 1–10, ACM (1992)
4. 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)
5. 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)
6. Danvy, O.: Back to direct style. In: ESOP, Lecture Notes in Computer Science, vol. 582, pp. 130–150, Springer (1992)
7. Danvy, O., Filinski, A.: Abstracting control. In: LISP and Functional Programming, pp. 151–160, ACM (1990)
8. Danvy, O., Lawall, J.L.: Back to direct style II: first-class continuations. In: LISP and Functional Programming, pp. 299–310, ACM (1992)
9. Danvy, O., Millikin, K.: Refunctionalization at work. Sci. Comput. Program. **74**(8), 534–549 (2009)
10. Danvy, O., Nielsen, L.R.: Defunctionalization at work. In: PPDP, pp. 162–174, ACM (2001)
11. 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)
12. Gibbons, J.: Continuation-passing style, defunctionalization, accumulations, and associativity. Art Sci. Eng. Program. **6**(2), 7 (2022)
13. 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)
14. Hillerström, D., Lindley, S., Atkey, R.: Effect handlers via generalised continuations. J. Funct. Program. **30**, e5 (2020)
15. 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)
16. 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)
17. Lawall, J.L., Danvy, O.: Continuation-based partial evaluation. In: LISP and Functional Programming, pp. 227–238, ACM (1994)

18. 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)
19. 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)
20. Plotkin, G.D.: A structural approach to operational semantics. J. Log. Algebraic Methods Program. **60-61**, 17–139 (2004)
21. Plotkin, G.D., Pretnar, M.: Handlers of algebraic effects. In: ESOP, Lecture Notes in Computer Science, vol. 5502, pp. 80–94, Springer (2009)
22. Plotkin, G.D., Pretnar, M.: Handling algebraic effects. Log. Methods Comput. Sci. **9**(4) (2013)
23. Reynolds, J.C.: Definitional interpreters for higher-order programming languages. In: ACM Annual Conference (2), pp. 717–740, ACM (1972)
24. 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
25. Thielecke, H.: Comparing control constructs by double-barrelled CPS. High. Order Symb. Comput. **15**(2-3), 141–160 (2002)
26. 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)
27. 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)
28. WebAssembly Contributors: WasmFX Specification. `https://wasmfx.dev/specs/core/` (2024), accessed: 2024-11-11
29. WebAssembly Contributors: Webassembly core specification: Runtime stack. `https://webassembly.github.io/spec/core/exec/runtime.html#stack` (2024), accessed: 2024-10-17
30. WebAssembly Contributors: WebAssembly Proposal: Exceptions (2024), URL `https://github.com/WebAssembly/exception-handling/blob/main/proposals/exception-handling/Exceptions.md`, accessed: 2024-09-25
31. WebAssembly Contributors: WebAssembly Proposal: Tail-Call (2024), URL `https://github.com/WebAssembly/tail-call`, accessed: 2024-10-24
32. WebAssembly Contributors: WebAssembly Proposal: Type Continuations (2024), URL `https://github.com/WebAssembly/stack-switching/blob/main/proposals/continuations/Explainer.md`, accessed: 2024-09-25
33. WebAssembly Contributors: WebAssembly Proposal: Typed Function References (2024), URL `https://github.com/WebAssembly/function-references`, accessed: 2024-10-24

34. WebAssembly Contributors: WebAssembly Specification (2024), URL `https://webassembly.github.io/spec/core/`, accessed: 2024-09-25
35. WebAssembly Contributors: Webassembly specification testsuite (2024), URL `https://github.com/WebAssembly/spec/tree/main/test/core`, accessed: 2024-11-11
36. Wei, G., Jia, S., Gao, R., Deng, H., Tan, S., Bracevac, O., Rompf, T.: Compiling parallel symbolic execution with continuations. In: ICSE, pp. 1316–1328, IEEE (2023)
37. Youn, D., Shin, W., Lee, J., Ryu, S., Breitner, J., Gardner, P., Lindley, S., Pretnar, M., Rao, X., Watt, C., Rossberg, A.: Bringing the webassembly standard up to speed with spectec. Proc. ACM Program. Lang. **8**(PLDI), 1559–1584 (2024)