Iris-WasmFX: Modular Reasoning for Wasm Stack Switching

MAXIME LEGOUPIL, Aarhus University, Denmark
MATHIAS PEDERSON, Aarhus University, Denmark
LARS BIRKEDAL, Aarhus University, Denmark
SAM LINDLEY, University of Edinburgh, United Kingdom
JEAN PICHON-PHARABOD, Aarhus University, Denmark

WebAssembly, abbreviated Wasm, is a low-level portable bytecode. WasmFX is a proposed extension of Wasm with primitives for explicitly manipulating execution stacks as continuations. By exposing an interface in the style of effect handlers, WasmFX enables non-local control flow features to be compiled in a modular way: one handcrafts a library that directly implements such features in WasmFX, and compilation then merely calls into the library. Alas, code involving non-local control flow is notoriously challenging, and so this proposal raises the questions of the soundness of the language extension, and of the correctness of such handcrafted libraries. In this paper, we first describe WasmFXCert, a mechanisation of WasmFX in Rocq, and prove the expected type soundness result. We then develop Iris-WasmFX, a program logic to reason about WebAssembly programs that use effect handlers, and illustrate its application to two key use cases of effect handlers: a coroutine library, and a generator. Together, these validate the design of stack switching, and provide a modular framework for verifying future effect-based libraries.

1 Introduction

Wasm [Haas et al. 2017; Rossberg 2019, 2023] is a low-level portable bytecode extensively used both on the client- and server-side. As such, it is the compilation target for code that involves non-local control flow, including features such as promises, async/await, generators, and more. Compiling such features to current versions of Wasm requires a whole-program transformation such as the one used by Binaryen's Asyncify [Zakai 2019]. However, such transformations suffer from three problems. First, such transformations introduce complexity. Second, they incur a significant overhead, in particular in code size. And third, they are not compositional: transformed code can only be composed with transformed code, not with off-the-shelf Wasm code.

To enable simpler, more efficient, and compositional compilation to Wasm, WasmFX [Phipps-Costin et al. 2023] extends Wasm with non-local control flow primitives based on Plotkin and Pretnar's effect handlers [Plotkin and Pretnar 2009, 2013]. WasmFX is in the process of being standardised under the name *stack switching* [McCabe et al. 2025] as it provides a means for switching stacks. Previously the design was also referred to as *typed continuations* [Hillerström 2021; Rossberg 2020; Rossberg et al. 2020], as it is founded on a notion of continuations that follow the stack typing discipline of Wasm. In this paper we stick with *WasmFX* for conciseness. We will sometimes refer to the official standardisation documents for WasmFX [McCabe et al. 2025] as the *WasmFX proposal.* WasmFX already has implementations and industrial uses, but two important aspects remain to be investigated. First, although the static and dynamic semantics of Wasm are given in the proposal, the typing rules for runtime *administrative* instructions and a type soundness proof are missing. Second, the expected use case of WasmFX is to combine compiled code with hand-optimised libraries efficiently implementing non-local control flow features of source languages, raising the question of how to prove correctness of such libraries.

Authors' Contact Information: Maxime Legoupil, Aarhus University, Aarhus, Denmark, maxime@cs.au.dk; Mathias Pederson, Aarhus University, Aarhus, Denmark, mp@cs.au.dk; Lars Birkedal, Aarhus University, Aarhus, Denmark, birkedal@cs.au.dk; Sam Lindley, University of Edinburgh, Edinburgh, United Kingdom, mymail@mail.com; Jean Pichon-Pharabod, Aarhus University, Aarhus, Denmark, jean.pichon@cs.au.dk.

Contributions. In this paper, we make two main contributions to the formal study of WasmFX: WasmFXCert, a mechanisation of the operational semantics and type system of WasmFX; and Iris-WasmFX, a program logic that enables modular reasoning for programs using WasmFX.

First, in order to formalise the semantics of WasmFX, we introduce WasmFXCert, a mechanisation of WasmFX in the Rocq proof assistant, built atop the existing WasmCert [Watt et al. 2021] formalisation of plain Wasm. WasmFXCert defines the full operational semantics and syntactic type system of WasmFX, including formally defined typing of administrative instructions, which existing specifications did not cover beyond an informal description [Phipps-Costin et al. 2023, §3.4]. WasmFXCert also provides the first result on the type safety of WasmFX, in the form of a mechanised proof which covers administrative instructions — thus validating the basic consistency of the WasmFX proposal.

Second, because effect handlers introduce non-local control flow, they famously hinder modular reasoning at the level of the operational semantics [Dijkstra 1968; Madore 2001]. In particular, capturing and storing of continuations make it challenging [Timany and Birkedal 2019] to reason about a program module-by-module, as doing so means that control does not flow in a well-bracketed way — and therefore not in a module-bracketed way — anymore. To address this difficulty, we introduce Iris-WasmFX, the first modular reasoning method for WasmFX programs. Iris-WasmFX is a program logic for WasmFX defined in the Iris separation logic framework [Jung et al. 2018] by extending the existing Iris-Wasm program logic [Rao et al. 2023] for plain WebAssembly, and takes inspiration from Hazel [de Vilhena 2022; de Vilhena and Pottier 2021, 2023], an Iris-based program logic for an ML-style calculus with effect handlers.

In this paper, we outline the difficulties that we faced when defining Iris-WasmFX, and highlight the challenges of doing machine-checked proofs of properties of industrial-size languages. Our mechanisation and program logic are crucial tools for gaining a deep understanding of the behaviour of WasmFX programs and establishing confidence in desired properties of specific programs. Libraries whose behaviour is critical for safety can be verified manually, and users of WasmFX can establish a high-degree of assurance in the soundness of its design. Together, WasmFXCert and Iris-WasmFX validate the design of WasmFX, and illustrate how to achieve modular reasoning for the prototypical kind of libraries that WasmFX was designed to support.

Plan. We first present our running example of a typical use-case of WasmFX, which we use throughout the paper to demonstrate how to program, specify, and verify with WasmFX (§2). We then describe the design of WasmFX and our formalisation, WasmFXCert, (§3), and establish type safety. On top of this foundation, we describe our program logic, Iris-WasmFX (§4), and apply it to two case studies (§5). Finally, we discuss related work (§6), and conclude (§7).

2 A Canonical Use Case of WasmFX: a Coroutine Library

In the remainder of this paper, we illustrate the design of WasmFX, our mechanisation WasmFX-Cert, and our program logic Iris-WasmFX, on a canonical use case of WasmFX: a small cooperative coroutine library module. The coroutine module exposes an interface comprising two functions. Library function $par takes two function references as arguments; these define two coroutines whose execution is interleaved. Library function $yield is used to signal that control should be handed over to the other coroutine, thereby cooperatively interleaving the execution of the coroutines. Behind the scenes, $par implements a scheduler that starts running one function, and if that function stops by invoking $yield, then starts running the other function, and so on.

This example illustrates the kind of code that WasmFX makes possible. Without it, this kind of functionality would require a somewhat complex compilation of the client code, likely incurring some code size inflation. Instead, with WasmFX, a concise definition can be given to $par and

$yield, as we show below. Thanks to Wasm's modularity, from the point of view of a client, the functions $par and $yield can be called like normal functions, and the code of the client can thus be written without using the new instructions in the WasmFX semantics.

A Simple Specification of Coroutines. A natural specification for this coroutine module in the style of Concurrent Separation Logic (CSL) [Brookes and O'Hearn 2016] is centred around the notion of an invariant: if both client functions $\$\mathsf{f}_\mathsf{i}$ (for i in {1, 2}) being run ensure that they establish the invariant I before calling $yield, then they can rely on the fact that this same invariant is restored when control returns to them — that is, when $yield returns. Assuming this, if each function $\$\mathsf{f}_\mathsf{i}$, starting from precondition P_i, ensures postcondition Q_i at termination, with access to invariant I at the start and relinquishing it at the end, then running both functions via $par starting with P_1, P_2 and I ensures that at termination, Q_1, Q_2 and the invariant hold. We give a more detailed intuition for the shape of the specification by way of Hoare triples for the $yield and $par functions. The Hoare triple for $par uses nested Hoare triples [Schwinghammer et al. 2009]: it requires in its precondition corresponding specifications for $\$\mathsf{f}_1$ and $\$\mathsf{f}_2$ in the form of Hoare triples:

$$
\begin{array}{l}
\forall P_1, P_2, Q_1, Q_2, I. \\
\quad \{I\}[\texttt{invoke}\ \mathit{addryield}]\{I\} \ast \\
\quad \left\{\begin{array}{l} P_1 \ast P_2 \ast I \ast \\ \{P_1 \ast I\}[\texttt{invoke}\ \mathit{addrf_1}]\{Q_1 \ast I\} \ast \\ \{P_2 \ast I\}[\texttt{invoke}\ \mathit{addrf_2}]\{Q_2 \ast I\} \end{array}\right\}
\left[\begin{array}{l} \texttt{func.ref}\ \$\mathsf{f}_1; \\ \texttt{func.ref}\ \$\mathsf{f}_2; \\ \texttt{invoke}\ \mathit{addrpar} \end{array}\right]
\{Q_1 \ast Q_2 \ast I\}
\end{array}
$$

where *addryield*, *addrpar*, *addrf₁* and *addrf₂* are the addresses of the closures for $yield, $par, $\$\mathsf{f}_1$ and $\$\mathsf{f}_2$. We show how to make this informal specification precise in Appendix E.

Modularity. Importantly, this specification can be proved for the coroutine module by itself, without access to the code of the client functions $\$\mathsf{f}_\mathsf{i}$, which can themselves be verified against this specification, without reference to the code of the coroutine module.

Implementation. This coroutine module can be implemented using WasmFX: $yield `suspends` the current program, saving the current continuation, and $par `resumes` the continuations of $\$\mathsf{f}_1$ and $\$\mathsf{f}_2$, round-robin, using the corresponding WasmFX instructions. We describe this simple implementation of coroutines in Section 3, and give its code in Figure 3.

Library Client. A simple example use of this coroutine library is a client implementing a classic 'message passing' concurrency example [Sewell et al. 2010, 8–1][Maranget et al. 2012, §3]. We implement this in Wasm (the code is in the appendix in Figure 13) in the form of a client module that uses the coroutine module to concurrently run a 'writer' function $\$\mathsf{f}_1$ that writes 42 to a 'data' **global** $x, and then 1 to a 'flag' **global** $y to signal that the data is ready in $x, with a 'reader' function $\$\mathsf{f}_2$ that reads the 'flag' $y and then reads the 'data' in $x, with both functions calling $yield in between their two accesses to globals. Using this specification of the coroutine library, the goal is to prove that if the reader sees that the flag is set, then it should see 42 as the data. In the rest of this paper, we build up to making the above specification precise and prove it correct.

3 WasmFXCert: Formal Operational Semantics for WasmFX

In this section, we introduce WasmFX and our mechanisation of it in the Rocq proof assistant, WasmFXCert. Phipps-Costin et al. [2023, §2–3] give a more detailed introduction to WasmFX.

3.1 WasmFX, Informally

Design of WasmFX. WasmFX is effectively a variant of *effect handlers*, a mechanism for non-local control flow that can be thought of as resumable exceptions: when an effect is performed, execution

```
1 (loop $l
2   (block $b
3     ...                      ;; prepare function to run
4     resume ... (on $t $b) ;; run
5     ...                      ;; handle normal termination
6     br $l)                   ;; resume the loop, circumventing the code for yield below
7   ...                        ;; handle yield, changing function to run
8   br $l)                     ;; resume the loop
```

Fig. 1. High-level structure of the implementation of $par (full code in Figure 3)

is halted, and control is handed over to the innermost *handler* for that effect, thereby defining a delimited continuation [Felleisen 1988]. If it was intended for programmers to routinely write code directly in Wasm, it would make most sense to define a specific, user-friendly mechanism. But since Wasm is designed as a compilation target, it makes most sense to offer a flexible mechanism that can be used to implement different non-local control flow mechanisms, even if it is at the cost of being less human-readable. Therefore, the design of Wasm's effect handlers in WasmFX is somewhat unusual, and we illustrate it on our motivating example. Phipps-Costin et al. [2023, §4] show how to implement other types of non-local control flow using the instructions of WasmFX.

WasmFX Instructions. The core of WasmFX consists of two instructions: `suspend`, which stops the current execution context, and reifies it as a *continuation*, a new kind of Wasm value with a corresponding new Wasm type; and `resume`, which starts executing such a continuation — thereby switching the stack, hence the name of the Wasm proposal. In our coroutine example, $yield can be implemented by a `suspend`, interrupting the execution of the current function, and $par can be implemented by `resume`-ing the other function, in a loop, as sketched in Figure 1. We now focus on the `resume` instruction on line 4.

Block-Based Effect Handling. The main unusual feature of WasmFX is that `suspend` suspends to a code `block` specified in the `resume`, as opposed to some more user-friendly mechanism. As such, WasmFX continuations are *delimited* continuations [Felleisen 1988]. In our coroutine example, this means that the code just after the `resume` (line 5, within the block) handles normal termination, and the code after the block (line 7) handles the $yield. Because execution is fall-through, normal termination (in `block` $b) has to jump over the handling of the yield (lines 7–8, after `block` $b). Confusingly, in Wasm, `loop` only declares a `block` to loop to: actual looping only takes place given an explicit `br` $l (lines 6 and 8).

Tags. To compose different effects modularly, `suspend` is annotated with a `tag` (called an *effect operation* in the effect handler literature). Each `tag` specifies what types of values (the *payload* of the effect) are to be passed along with it. Then `resume` can specify, for each `tag`, to which `block` it should be suspended, using the `on` keyword. In our coroutine example, we only need one tag to (ask $par to) pass over control to the other function: $yield then just suspends with that tag.

3.2 Operational Semantics

Pedigree. The WasmFX proposal builds upon Wasm 3.0. However, it only makes use of a few features not present in Wasm 1.0: function references, and exception handling [Ahn and Titzer 2022]. Our mechanisation, WasmFXCert, takes advantage of this, and is built on top of WasmCert 1.0 [Watt et al. 2021], to which we have manually added the features of Wasm 3.0 that WasmFX uses. This is because Iris-Wasm [Rao et al. 2023], which we used as a departure point to define our program logic (see §4), is defined for Wasm 1.0; defining it atop Wasm 3.0 would have required dealing with the numerous features of Wasm 3.0 that are orthogonal to WasmFX.

In the rest of this section, we give an introduction to the semantics of Wasm. A reader familiar with Wasm can safely skip to §3.3, where we describe the new instructions in WasmFX. A reader familiar with WasmFX can skip to §3.4, where we describe type safety.

Syntax. We give the syntax for WasmFX in Figure 2, and illustrate how a Wasm module is structured with our implementation of a coroutine module in Figure 3. To avoid confusion with the symbol $*$ for the separating conjunction, we depart from the Wasm standard and use an 's' suffix to indicate a list; e.g. bs indicates a list of b elements. The coroutine module in Figure 3 defines a type (line 2), a continuation type (line 3), a tag (line 4), and the two library functions $yield (lines 5–6) and $par (lines 7–39).

Frame and Store. The state of a Wasm program is separated in two parts: the *frame* and the *store*. The store S contains all objects (e.g. global variables, functions, memories, as well as, in WasmFX, tags and continuations) declared by all modules instantiated thus far, making imports and exports easy to deal with. The frame F contains information about the environment of the function currently being run. It consists of two components: the values of the function's local variables and the *instance*, a dictionary that translates local indices into addresses in the store. This indirection is crucial to obtain Wasm's local state encapsulation property, but increases the complexity of defining a mechanisation and program logic.

Reductions. Reductions are of the form $(S, F, es) \hookrightarrow (S', F', es')$, where S and F are the aforementioned store and frame, and es is a list of *administrative instructions*, a superset of basic instructions that includes instructions used to represent a program mid-execution; WasmFX introduces a number of administrative instructions, and we describe them in the rest of this subsection.

Control Flow. To express the semantics of `loop` and `block`, Wasm defines an administrative instruction $\texttt{label}_n\{es_{cont}\}\ es\ \texttt{end}$, that both `loop` and `block` reduce to, where es represents the body of the `loop` or `block`; es_{cont} represents the code that should be run in case of a break (i.e. the entire loop again in the case of a `loop`, or nothing in case of a `block`), and n is the number of values that should be on the stack in case of a break. Wasm defines evaluation by the composition of head reduction rules with the following evaluation-under-context rule:

$$\textsc{reduce-label}\quad \frac{(S, F, es) \hookrightarrow (S', F', es')}{(S, F, L[es]) \hookrightarrow (S', F', L[es'])}$$

where evaluation contexts L are as per Figure 2 (they are written lh_i in Iris-Wasm [Rao et al. 2023], and B^i in the standard [Rossberg 2023].) Taking $L = [\texttt{label}_n\{es_{cont}\}\ [\_]\ \texttt{end}]$, we get that after a `loop` or `block` has reduced to a `label`, the code inside the label can run. If that code terminates on a value, the final value exits the label and execution continues with the instructions that follow the label. If however a `br` instruction (for *break* or *branch*) is encountered, the appropriate amount of values is taken from the stack and code es_{cont} is run. In WasmFX, as we will detail in §3.3, the `resume` instruction also reduces to a `br` $b when an effect is performed, meaning that the code after the instruction corresponds to the code that will be run if no effect is performed, whereas the code es_{cont} of the `label` instruction labelled $b will run if an effect is performed.

Function Calls. Wasm defines several mechanisms for function calls, and an administrative instruction `invoke` *addr* that all the primitive call instructions reduce to. The simplest calling instruction is `call` $f. The frame is used to determine at what address *addr* in the store to find the (numerical value type) nt ::= $\mathbf{i32} \mid \mathbf{i64} \mid \mathbf{f32} \mid \mathbf{f64}$ (value type) t ::= $nt \mid rt$

(reference value type) rt ::= $\boxed{ft.\texttt{func} \mid \texttt{exn} \mid ft.\texttt{cont}}$ (function type) ft ::= $ts \longrightarrow ts$

(continuation clause) cc ::= $\texttt{on}\ i\ i' \mid \texttt{on}\ i\ \texttt{switch}$ (value) v ::= $nt.\texttt{const}\ c \mid rv$

(immediate) $i, addr, max$ ::= $\mathbb{N}$ (i for local indices, $addr$ for store addresses)

(reference value) rv ::= $rt.\texttt{null} \mid \texttt{ref.func}\ addr \mid \texttt{ref.exn}\ addr\ addr' \mid \texttt{ref.cont}\ addr$

(basic instructions) b ::= $nt.\textbf{const}\ c \mid t.\texttt{add} \mid \textit{other stackops} \mid \texttt{local.\{get/set\}}\ i \mid$
$\texttt{global.\{get/set\}}\ i \mid t.\texttt{load}\ \textit{flags} \mid t.\texttt{store}\ \textit{flags} \mid \texttt{memory.size} \mid$
$\texttt{memory.grow} \mid \texttt{block}\ ft\ bs \mid \texttt{loop}\ ft\ bs \mid \texttt{if}\ ft\ bs\ bs' \mid \texttt{br}\ i \mid \texttt{br\_if}\ i \mid$
$\texttt{br\_table}\ \textbf{is} \mid \texttt{call}\ i \mid \texttt{call\_indirect}\ \ i \mid \texttt{return} \mid rt.\texttt{null} \mid$
$\texttt{ref.is\_null} \mid \texttt{ref.function}\ i \mid \texttt{ref.call}\ i \mid$
exception handling instructions $\mid \texttt{resume}\ i\ ccs \mid$
$\texttt{suspend}\ i \mid \texttt{switch}\ i\ i' \mid \texttt{cont.bind}\ i\ i' \mid \texttt{resume\_throw}\ i\ i'\ ccs$

(translated continuation clause) dcc ::= $\texttt{on}\ addr\ i \mid \texttt{on}\ addr\ \texttt{switch}$

(administrative instruction) e ::= $b \mid \texttt{trap} \mid \texttt{invoke}\ addr \mid \texttt{label}_n\{es'\}\ es\ \texttt{end} \mid$
$\texttt{frame}_n\{F\}\ es\ \texttt{end} \mid \texttt{call\_host}\ ft\ i\ vs \mid \texttt{ref.func}\ addr \mid$
exception handling instructions $\mid \texttt{ref.cont}\ addr \mid$
$\texttt{prompt}_{\boxed{ts}}\{dccs\}\ es \mid \texttt{suspend.addr}\ \boxed{vs}\ addr$
$\texttt{switch.addr}\ \boxed{vs}\ addr\ ft\ addr'$

(functions) $func$ ::= $\texttt{func}\ i\ ts\ bs$ (tables) tab ::= $\texttt{tab}\ min\ max$

(memories) mem ::= $\texttt{mem}\ min\ max$ (globals) $glob$ ::= $\texttt{glob mutable}\ t\ e_{\texttt{init}}$

(import descriptions) $importdesc$ ::= $\texttt{func}_{\texttt{i}}\ i \mid \texttt{tab}_{\texttt{i}}\ i\ i' \mid \texttt{mem}_{\texttt{i}}\ i\ i \mid \texttt{glob}_{\texttt{i}}\ \texttt{mutable}^?\ t \mid \texttt{tag}_{\texttt{i}}\ i$

(export descriptions) $exportdesc$ ::= $\texttt{func}_{\texttt{e}}\ i \mid \texttt{tab}_{\texttt{e}}\ i \mid \texttt{mem}_{\texttt{e}}\ i \mid \texttt{glob}_{\texttt{e}}\ i \mid \texttt{tag}_{\texttt{e}}\ i$

(modules) m ::= $\{\text{types} : fts,\ \text{tags} : fts,\ \text{funcs} : funcs,\ \text{globs} : globs,\ \text{mems} : mems, \ldots\}$

(module instance) $inst$ ::= $\{\text{types} : fts,\ \text{funcs} : addrs,\ \text{globs} : addrs,\ \text{mems} : addrs,$
$\text{tabs} : addrs,\ \text{tags} : addrs\ \}$

(frame) F ::= $\{\text{locs} : vs,\ \text{inst} : inst\ \}$

(evaluation context) L ::= $\boxed{vs +\!\!+ [\_] +\!\!+ es} \mid vs +\!\!+ \texttt{label}_n\{es'\}\ L\ \texttt{end} +\!\!+ es \mid$
$vs +\!\!+ \texttt{prompt}_{ts}\{dccs\}\ L\ \texttt{end} +\!\!+ es$

(handler context) H ::= $vs +\!\!+ [\_] +\!\!+ es \mid vs +\!\!+ \texttt{label}_n\{es'\}\ H\ \texttt{end} +\!\!+ es \mid$
$vs +\!\!+ \texttt{frame}_n\{F\}\ H\ \texttt{end} +\!\!+ es \mid$
$vs +\!\!+ \texttt{prompt}_{ts}\{dccs\}\ H\ \texttt{end} +\!\!+ es$

(function instances) $finst$ ::= $\{(inst; ts); bs\}^{\texttt{NativeCl}}_{ft} \mid \{hidx\}^{\texttt{HostCl}}_{ft}$

(continuation instance) $cinst$ ::= $\texttt{cont}\ \boxed{ft}\ H \mid \texttt{dagger}\ \boxed{ft}$

(store) S ::= $\{\text{funcs} : finsts,\ \text{globs} : ginsts,\ \text{mems} : minsts,\ \text{tabs} : tinsts,$
$\text{tags} : fts,\ \text{exns} : einsts,\ \text{conts} : cinsts\ \}$

(the meaning of some of the fields in the store is orthogonal to stack switching and is omitted)

Fig. 2. The AST of the WasmFXCert language. Black parts are in WasmCert 1.0 (plain Wasm 1.0). Indigo parts are reference instructions from Wasm 3.0. Orange parts are the additions of exception handling, that are also needed for WasmFX. Magenta parts belong to WasmFX proper. A full AST can be found in appendix, Figures 7 and 8. Places where WasmFXCert departs from WasmFX are shown $\boxed{\text{boxed}}$.

```
(module ;; coroutine
  (type $func (func))
  (type $cont (cont $func))
  (tag $t)
  (func $yield                         ;; takes no arguments
    (suspend $t))                      ;; and just suspends
  (func $par
    (param $f1 (ref $func))            ;; takes two functions as arguments
    (param $f2 (ref $func))
    (local $current (ref $cont))       ;; local variables:
    (local $next (ref $cont))
    (local $next_is_done i32)
    i32.const 0                        ;; $next_is_done := false
    local.set $next_is_done
    local.get $f1                      ;; $current := cont(f1)
    cont.new $cont
    local.set $current
    local.get $f2                      ;; $next := cont(f2)
    cont.new $cont
    local.set $next
    (loop $l                           ;; while true do
      (block $b (result (ref $cont))
        local.get $current             ;; resume continuation $current
        resume $cont (on $t $b)
        local.get $next_is_done        ;; if it terminates, and $next_is_done
        br_if 2                        ;; return
        i32.const 1                    ;; $next_is_done := true
        local.set $next_is_done
        local.get $next                ;; $current := $next
        local.set $current
        br $l)                         ;; loop
      local.set $current               ;; else (if effect is triggered) $current := cont
      local.get $next_is_done          ;; if $next_is_done
      br_if $l                         ;; loop
      local.get $current               ;; swap $current $next
      local.get $next
      local.set $current
      local.set $next
      br $l))                          ;; loop
  (export "yield" (func $yield))       ;; make $yield available
  (export "par" (func $par)))          ;; and $par
```

Fig. 3. Implementation of our simple coroutine module using WasmFX

closure for $f, and `call` $f reduces to `invoke` *addr*. A module can also define a function table and use `call_indirect` to call functions from it. In Wasm 3.0, the `ref.function` $f instruction creates a reference to an existing function, which can be called using the `ref.call` instruction.

Closures. The closures in the store can either be defined by the embedding host language (see §3.5, in which case control is handed over to the host language, which in WasmCert is rendered by reducing to a special `call_host` administrative instruction [Rao et al. 2023, §2.2]); or it can be defined in Wasm, in which case `invoke` *addr* reduces to $(\texttt{frame}_{|ts_2|}\{F'\}\ \texttt{block}\ ([] \longrightarrow ts_2)\ bs\ \texttt{end})$ where *bs* is the body of the closure, ts_2 is the function's return type, and F' is the frame in which this body is meant to be executed. This frame F' contains the function's arguments as local variables, and the instance of the module in which the function was defined, hence ensuring the code of the function can only access the elements of the store it is intended to.

The $\texttt{frame}_n\{F\}\ es\ \texttt{end}$ administrative instruction (sometimes called `local` [Haas et al. 2017; Rao et al. 2023]) represents the execution of a function call, and is responsible for ensuring encapsulation. Its rule is similar to REDUCE-LABEL, but updates the frame in its premise:

REDUCE-FRAME
$$\frac{(S, F, es) \hookrightarrow (S', F', es')}{(S, F_0, [\texttt{frame}_n\{F\}\ es\ \texttt{end}]) \hookrightarrow (S', F_0, [\texttt{frame}_n\{F'\}\ es'\ \texttt{end}])}$$

A `return` instruction is defined similarly to `br` and allows to exit a function by jumping out of the closest `frame` instruction. Since an `invoke` reduces to the body wrapped in both a `frame` and a `block`, the programmer can alternatively use a `br` instruction instead of a `return`, for example to take advantage of the `br_if` instruction, as is done in the code of `$par` at line 26.

3.3 New Instructions in WasmFX

We now introduce formally the new WasmFX instructions sketched in §3.1. WasmFX defines a new type, *ft*.`cont`, for continuations of type *ft*. All new continuations are created from a function reference using the `cont.new` instruction. The new continuation is placed in the store at an address *addr*, and `ref.cont` *addr* is returned:

REDUCE-CONTNEW
$$\frac{F.\texttt{inst.types}[i] = ft \qquad S' = \{S \text{ with } \texttt{conts} \mathrel{+\!\!+\!\!=} \texttt{cont}\ ft\ ([\_] +\!\!+ [\texttt{ref.func}\ addr; \texttt{ref.call}\ ft])\}}{(S, F, [\texttt{ref.func}\ addr; \texttt{cont.new}\ i]) \hookrightarrow (S', F, [\texttt{ref.cont}\ |S.\texttt{conts}|])}$$

Behaviour of `resume` *and* `prompt`. The `resume` instruction expects a value of the form `ref.cont` *addr* on the stack, and runs the continuation present in the store at address *addr*. The instruction takes a list of *clauses* as an immediate argument, describing the intended behaviour in case an effect is triggered. This means that in WasmFX, there are not two distinct instructions for running a continuation and installing a handler: both operations are dealt with by the `resume` instruction. This means the usual distinction between shallow and deep handlers is meaningless.

Similarly to `label` and `frame`, WasmFX defines an administrative instruction `prompt` to represent a continuation being run. Just like `resume`, `prompt` is decorated with a list of *clauses* that inform the desired behaviour in case of a `suspend` or a `switch`. One small difference is that the `resume` instruction's clauses reference the local tag names as written by the programmer, whereas the clauses in the `prompt` instruction has the explicit store addresses of these tags.

To account for the fact that in WasmFX, continuations are one-shot [Phipps-Costin et al. 2023, §2.4], upon resuming, the store is updated with a `dagger` token to indicate the continuation has already been resumed, and that therefore resuming it again would cause a `trap`:

REDUCE-RESUME
$$\frac{\begin{array}{cccc} F.\texttt{inst.types}[i] = ft & ft = ts_1 \longrightarrow ts_2 & |vs| = |ts_1| & S.\texttt{conts}[addr] = \texttt{cont}\ ft\ H \\ H[vs] = es & F \vdash ccs \longrightarrow dccs & S' = \{S \text{ with } \texttt{conts}[addr] = \texttt{dagger}\ ft\} & \end{array}}{(S, F, vs +\!\!+ [\texttt{ref.cont}\ addr; \texttt{resume}\ i\ ccs]) \hookrightarrow (S', F, [\texttt{prompt}_{ts_2}\{dccs\}\ es\ \texttt{end}])}$$

The *ccs* clauses in the resume are translated into explicit *dccs* clauses in the prompt instruction: on $i\ i'$ and on i switch are translated to on $addr\ i'$ and on $addr$ switch if $addr = F.\mathsf{inst.tags}[i]$. We write $F \vdash ccs \longrightarrow dccs$ to describe this translation. WasmFX extends execution contexts L to also include prompt contexts, so execution carries out inside of a prompt exactly like it does under a label. Likewise, if this execution terminates in a value, WasmFX defines a reduction rule making the value exit the prompt and continuing execution with the instructions following the prompt: this constitutes the 'normal return' case of the effect handling.

Behaviour of suspend. The suspend instruction takes a tag $t as an immediate argument. Informally, it takes the required amount of values from the stack to constitute the effect's payload, creates a delimited continuation out of the current state of the stack, and yields control to the innermost enclosing prompt that has a clause handling tag $t. This clause has the form on $t $l: this means that the handler intends for a break to label $l to occur when the effect identified by tag $t is triggered; hence the right-hand-side of reduction rule EWP-SUSPEND below is a break.

Translated suspend*.* One caveat is that the suspend instruction's immediate argument is a tag name $t and the prompt instructions use tag *addresses* (into the store) in their clauses. The WasmFX proposal originally attempted to do this translation implicitly but ended up using the wrong function instance to do so. This bug in the proposal was exposed by the mechanisation and has now been fixed. The reference interpreter avoided this by using a smaller-step operational semantics.

Our proposed solution in WasmFXCert is to make this translation from tag names to tag addresses explicit: we distinguish a basic instruction suspend with a local index for the tag, and an administrative instruction suspend.addr, with the actual address into the store. This solution has now also been adopted by the WasmFX proposal, albeit with a slight difference in naming, as the proposal prefers to broaden the definition of the suspend instruction by allowing it to alternatively take an address as an argument. This means that in the WasmFX proposal, suspend i is translated to suspend $addr$ instead of a new suspend.addr instruction as we do in WasmFXCert (see rule REDUCE-SUSPEND-TRANSLATE below). This is in order to maintain consistency with other places in the standard and limit the proliferation of administrative instructions. WasmFXCert uses two distinct instructions for clarity, but both solutions have equivalent semantics.

In WasmFXCert, we find it convenient to also let this translation take the payload of the effect from the stack and place it into the suspend.addr instruction itself. This has no effect on the behaviour of the execution, but gives the language a property that will be crucial when defining the Iris-WasmFX program logic (see Figure 5): inspecting an expression (without knowledge of the store) is enough to know whether the expression is an effect, and if so what the effect's payload is. Without placing the payload into the instruction itself, it is impossible to know how many values to take from the stack without knowing the type signature of the tag, which is in the store and not visible in the expression itself. We explain in §4.2 why this is important.

Thus we introduce the following rule for explicit translation of the suspend instruction; the standard has now introduced the same rule, but without relocating the payload:

$$\textsc{reduce-suspend-translate}\quad \frac{F.\mathsf{inst.tags}[i] = addr \qquad S.\mathsf{tags}[addr] = ts_1 \longrightarrow ts_2 \qquad |vs| = |ts_1|}{(S, F, vs \mathbin{+\!\!+} [\mathsf{suspend}\ i]) \hookrightarrow (S, F, [\mathsf{suspend.addr}\ vs\ addr])}$$

The Rule for Suspension. The suspend.addr instruction itself is stuck and needs to be placed under an appropriate context to reduce. When reducing $H[\mathsf{suspend.addr}\ vs\ addr]$, we look up for the innermost prompt in H which has a clause for tag address $addr$: this allows us to decompose the expression into $H_1[\mathsf{prompt}_{ts}\{dccs\}\ H_2[\mathsf{suspend.addr}\ vs\ addr]\ \mathsf{end}]$ where H_2 is capture-avoiding; we can use the REDUCE-LABEL and REDUCE-FRAME rules to focus reasoning inside of H_1, and then apply the rule REDUCE-SUSPEND rule below. We use notation $H^{\mathsf{sus}}_{addr}[es]$ to denote capture-avoiding plugging (i.e. no prompts with a suspend clause handling $addr$ in H) of es into context H.

REDUCE-SUSPEND

$$\frac{S.\mathsf{tags}[addr] = ts_1 \longrightarrow ts_2 \qquad \mathsf{first\_suspend\_occurrence}(addr, dccs) = \mathsf{on}\ addr\ i \qquad H^{\mathsf{sus}}_{addr}[\mathtt{suspend.addr}\ vs\ addr] = es \qquad S' = \{S\ \mathsf{with\ conts} \mathrel{+\!\!+}= \mathtt{cont}\ (ts_2 \longrightarrow ts)\ H\}}{(S, F, [\mathtt{prompt}_{ts}\{dccs\}\ es\ \mathtt{end}]) \hookrightarrow (S', F, vs \mathrel{+\!\!+} [\mathtt{ref.cont}\ |S.\mathsf{conts}|;\ \mathtt{br}\ i])}$$

Behaviour of `switch`. The `switch` instruction optimises the common combination of a `suspend` followed by immediately resuming a different continuation. It expects a continuation H on the stack, creates a continuation out of the current state of the stack, and yields control to H. Just like for `suspend`, in WasmFXCert, we explicitly translate the local index by distinguishing two instructions `switch` and `switch.addr`. We write $H^{\mathsf{sw}}_{addr}[es]$ to denote capture-avoiding (i.e. no prompts with a switch clause handling $addr$ in H) plugging of expression es in context H.

REDUCE-SWITCH-TRANSLATE

$$\frac{F.\mathsf{inst.types}[i] = ft \qquad ft = ts_1 \longrightarrow ts_2 \qquad F.\mathsf{inst.tags}[i'] = addr' \qquad |vs| + 1 = |ts_1|}{(S, F, vs \mathrel{+\!\!+} [\mathtt{ref.cont}\ addr;\ \mathtt{switch}\ i\ i']) \hookrightarrow (S, F, [\mathtt{switch.addr}\ vs\ addr\ ft\ addr'])}$$

REDUCE-SWITCH

$$\frac{(\mathtt{on}\ addr'\ \mathtt{switch}) \in dccs \qquad ft = (ts_1 \mathrel{+\!\!+} ft'.\mathtt{cont}) \longrightarrow ts_2 \qquad S.\mathsf{conts}[addr] = \mathtt{cont}\ (ts_1' \longrightarrow ts_2')\ H' \qquad H^{\mathsf{sw}}_{addr'}[\mathtt{switch.addr}\ vs\ addr\ ft\ addr'] = es \qquad S' = \{S\ \mathsf{with\ conts} \mathrel{+\!\!+}= \mathtt{cont}\ ft'\ H\} \qquad S'' = \{S'\ \mathsf{with\ conts}[addr] = \mathtt{dagger}\ (ts_1' \longrightarrow ts_2')\}}{(S, F, [\mathtt{prompt}_{ts}\{dccs\}\ es\ \mathtt{end}]) \hookrightarrow (S'', F, [\mathtt{prompt}_{ts}\{dccs\}\ (H'[vs \mathrel{+\!\!+} \mathtt{ref.cont}\ |S.\mathsf{conts}|])\ \mathtt{end}])}$$

Behaviour of `cont.bind` *and* `resume_throw`. Finally, WasmFX defines two more instructions for ease of programming: `cont.bind` partially applies a continuation, and `resume_throw` resumes a continuation but plugs in an exception-throwing instruction, as a way to cleanly discard a continuation that is no longer useful. These instructions are not crucial to understanding WasmFX, so we omit further mention of them, but we model them in WasmFXCert, and give them rules in our program logic Iris-WasmFX, which can be seen in our Rocq development.

Departures from the WasmFX Proposal in WasmFXCert. We make a few minor departures from the official WasmFX semantics. These are shown $\boxed{\text{boxed}}$ in Figure 2:

- Different versions of the standard and proposals use equivalent but different grammars for reference types and contexts. We pick one that is readable for the features we use.
- As discussed, we place the effect payload in `suspend.addr` and `switch.addr`.
- We add a type annotation on the `prompt` instruction and on the continuations in the store. Those apply only to administrative instructinos, not source instructions, so this does not affect typechecking of source code. We added these annotations because identifying the type of these expressions is likely to be useful for future users of Iris-WasmFX when conducting deeper analyses, like proofs by logical relation.

3.4 Syntactic Typing and Type Safety

To prove type safety for WasmFX, we need to extend the type system of Phipps-Costin et al. [2023] to define typing rules for administrative instructions. In doing so, we identify a few minor defects in their definitions (see Appendix C). We then prove the following theorem in Rocq (see Appendix C or the Rocq development for precise definitions):

Theorem 3.1 (Type Safety). *If the store S typechecks and $S, C(S,F) \vdash es : [] \rightarrow ts$ and $(S, F, es) \hookrightarrow^\star (S', F', es')$, then either es' reduces, or it is a constant value, or it is stuck on a host call, an unhandled suspend, switch, or exception throw.*

3.5 Instantiation

The process of preparing the modules for running is called *instantiation*: the code is typechecked, the imports are fetched, the objects defined by the module itself are added to the store, and the module's exports are prepared for subsequent imports by other modules.

This process cannot be performed by Wasm itself but instead is done by an embedding *host language*. The Iris-Wasm program logic [Rao et al. 2023] comes with a custom-designed host language that can instantiate Wasm modules and do a few other Wasm operations like calling Wasm functions or reading Wasm global variables. For example, to run our example from §2, the host program (shown in Theorem 5.1) consists of three steps: instantiating the coroutine module, instantiating the client module, and calling the $main function of the client module (see top of Figure 13 in appendix for how to run these three steps in the reference interpreter of WasmFX). This host language is mechanised in Rocq and has its own program logic; for simplicity, it does introduce its own effects.

4 Iris-WasmFX: Modular Reasoning for WasmFX Programs

In this section, we introduce Iris-WasmFX, our program logic for WasmFX.

4.1 Iris and Iris-Wasm

Our program logic is defined in Iris [Jung et al. 2018], a framework for higher-order separation logic which has already been used to reason about Wasm with Iris-Wasm [Rao et al. 2023] and Memory-Safe Wasm (MSWasm) [Michael et al. 2023] with Iris-MSWasm [Legoupil et al. 2024], among others. Once instantiated with our operational semantics, Iris provides us with proof rules that can be used to reason about the execution of programs, and we can derive instruction-specific rules to allow for mostly syntax-oriented verification proofs.

The central construct in our logic, used to give specifications, is our *extended weakest precondition*

$$\text{ewp } es \,; F \, \langle \Psi \rangle \, \{w \; F, \Phi(w, F)\}$$

We define it by adding Wasm-specific components (see §4.2 and §4.4) to Hazel's extended weakest precondition [de Vilhena 2022; de Vilhena and Pottier 2021, 2023], which is itself built on the (plain) *weakest precondition* statement from the Iris logic, $\text{wp } es \, \{w, \Phi(w)\}$.

This weakest precondition statement can be read as 'the expression es runs safely, and if it terminates on a value w, then *postcondition* Φ holds of w'. For readability, we also write specifications under the form of a *Hoare triple* $\{P\} es \{w, \Phi(w)\}$, which means that as long as the *precondition* P holds before execution, es runs safely and if it terminates on a value w, then Φ holds of w. This Hoare triple is defined in terms of the weakest precondition, as $\Box(P \twoheadrightarrow \text{wp } es \, \{w, \Phi(w)\})$, where $\twoheadrightarrow$ is a separating implication, and the persistent modality $\Box$ means that this separating implication can be used any number of times, but cannot rely on non-persistent knowledge when it is proven.

Resources. In the above definition, P and $\Phi(w)$ are separation logic predicates that describe individual pieces of the program state. These can be thought of as (the conjunction of) *resources*, each representing *ownership* of part of the state. In Iris-Wasm [Rao et al. 2023], these represent individual pieces of the Wasm store, like function closures ($addr \xmapsto{\text{wf}} cl$), globals ($addr \xmapsto{\text{wg}} gval$), etc. The arrow label differentiates the various kinds of resources. To accommodate WasmFX, Iris-WasmFX defines two new resources: one for tags, and one for continuations.

Tag Resources. Tag resource $addr \xmapsto{\texttt{wtag}} ft$ represents ownership of a tag. We often use fractional resources $addr \xmapsto{\texttt{wtag}}_q ft$ ($q \in \mathbb{Q}$), as an easy way to share the resource between parts of a program.

Continuation Resources. The second resource we introduce represents exclusive ownership of a *safe* continuation and its type: $addr \xmapsto{\texttt{wcont}} (ft, scont)$. We define a safe continuation as being either a dead continuation (which arise because WasmFX continuations are one-shot), or a *safe context*, which is either the context $[\_] +\!\!+ [$`ref.func` $f; `ref.call` $ft]$ for some $f and ft, or a context of the form [`frame`$_n\{F\}$ H `end`] (with H a handler context as defined in Figure 2). This means that in our program logic, we enforce that continuations always start off as a function call. If H is a safe context, then once plugged with any expression es, the resulting expression $H[es]$ reduces independently of the current frame (i.e. (S, F_1, es) reduces to (S', F_1', es') if and only if (S, F_2, es) reduces to (S', F_2', es')), since a `ref.call` reduces to a call no matter the frame, and reducing the body of a `frame` is always done with the frame F specified in the `frame` instruction itself rather than the frame F_1 or F_2 used when reducing the `frame` instruction. We show why this property is crucial for soundness when we introduce the proof rule for `resume` in §4.3.

Proof Rules. The Iris-Wasm program logic [Rao et al. 2023] for Wasm 1.0 defines proof rules for all Wasm instructions using the plain weakest precondition statement, most of which we inherit almost as-is. For example, the Iris-Wasm rule for updating a global variable is:

$$\frac{\overset{\textsc{wp-global-set}}{\xhookrightarrow{\textsc{Fr}} F * F.\mathsf{inst.globs}[\$g] = addr * addr \xmapsto{\texttt{wg}} v_1}}{\mathsf{wp}\ [v_2; \texttt{global.set}\ \$g]\ \left\{w, w = \texttt{immV}\ [] * \xhookrightarrow{\textsc{Fr}} F * addr \xmapsto{\texttt{wg}} v_2\right\}} \quad \text{in plain Iris-Wasm}$$

where `immV` is the constructor for *logical values* of Iris-Wasm:

Logical Values. In order to enable modular reasoning, Iris-Wasm defines several kinds of *logical values*, all representing expressions that do not reduce:

- Immediate values `immV` vs represent lists of Wasm values.
- Trap values `trapV` represent a `trap` failure value.
- Breaking values `brV` i vh represent `br` i expressions in a context vh syntactically enforced (by dependent typing, which we omit here) to be too shallow to break out from.
- Return values `retV` sh represent `return` expression in a context sh without a `frame`.
- Host call values `callhostV` vs k tf llh represent calls to host closures.

The reason that Iris-Wasm defines logical values more broadly than Wasm values is that it makes reasoning modular by making it possible to have a standard *bind rule* [Jung et al. 2018, Fig 13], which decomposes reasoning about a compound expression into reasoning about its parts in turn: reasoning about es in context L, that is, $L[es]$, can be decomposed into reasoning first about es itself, and then about the result w of es in context L. In Iris-Wasm, this rule looks like this:

$$\frac{\overset{\textsc{wp-bind-label}}{\mathsf{wp}\ es\ \left\{w, \mathsf{wp}\ L[w]\ \{v, \Phi(v)\}\right\}}}{\mathsf{wp}\ L[es]\ \{v, \Phi(v)\}} \quad \text{in plain Iris-Wasm}$$

This rule mirrors reduction rule REDUCE-LABEL, and makes it possible to focus reasoning on a specific instruction so that one can, say, apply the proof rule specific to that instruction. Thanks to `brV` i vh being a value, one can focus on code that will get stuck: once it is reduced to the value w = `brV` i vh, the postcondition must hold of w, which means there remains to prove a weakest precondition on $L[vh[$`br` $i]]$. If L contains the block targeted by `br` i, then one can apply Iris-Wasm's WP-BR rule to continue reasoning about the execution of the code after the jump. Otherwise, the expression is again a value `brV` i $(L \circ vh)$, and one must now prove that Φ holds of that value.

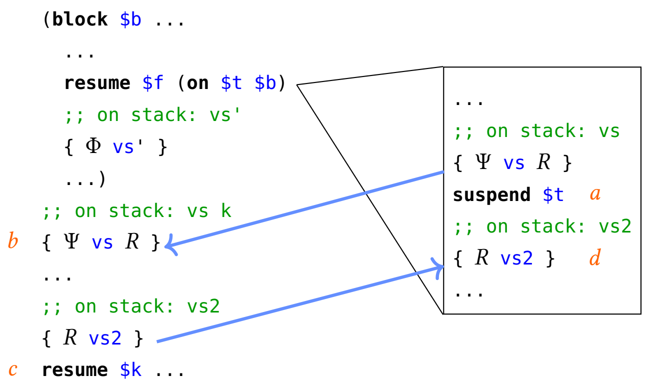
```
    (block $b ...
      ...
      resume $f (on $t $b)
      ;; on stack: vs'
      { Φ vs' }
      ...)
    ;; on stack: vs k
b   { Ψ vs R }
    ...
    ;; on stack: vs2
    { R vs2 }
c   resume $k ...
```
```
...
;; on stack: vs
{ Ψ vs R }
suspend $t   a
;; on stack: vs2
{ R vs2 }    d
...
```

Fig. 4. The protocol Ψ relates the payload vs of the suspend (from *a* to *b*) to the resumption condition *R*: the condition under which the continuation can be resumed (from *c* to *d*).

This rule is sound in Iris-WasmFX *provided one limits* the evaluation context *L* to only contain `label` and not `prompt`, since non-local control flow breaks the bind rule [Timany and Birkedal 2019]. Binding into a `prompt` requires a more intricate rule, as we show in §4.3 and Appendix B.

Frame. Iris-Wasm features a resource $\overset{\text{FR}}{\hookrightarrow} F$ that symbolises ownership of the frame. Instead, our custom weakest precondition considers the frame as part of the expression rather than part of the state. As this is both orthogonal to our main contribution and can be used intuitively, we describe it in detail in Appendix B, and merely use our new approach in the rest of the paper.

4.2 Hazel

Motivation. When dealing with effects, the standard weakest precondition definition of Iris does not provide a satisfying level of modularity. The problem is caused by the `suspend` instruction, which translates to `suspend.addr` as per rule REDUCE-SUSPEND-TRANSLATE, and is then stuck: it is neither a value, nor can it take a step *by itself*, only in the context of a `prompt`, as per rule REDUCE-SUSPEND. Contrary to `br`, the enclosing `prompt` handling the effect might be in a different function or even a different module, and hence making suspend a logical value as is done for `br` in Iris-Wasm would require knowledge of the entire handling environment and break modularity.

In our running example, this would mean that to reason about the client functions that invoke $yield, and thereby `suspend`, one would have to consider the code of $par, breaking modularity.

Extended Weakest Precondition. The approach of Hazel [de Vilhena 2022; de Vilhena and Pottier 2021, 2023] is to extend the weakest precondition with a new component that locally accounts for non-local effects. Hazel's *extended weakest precondition* ewp $es\,\langle\Psi\rangle\,\{w, \Phi(w)\}$ means not only (as usual) that es runs safely, and that if it terminates on a value w, then postcondition Φ holds of w, but also (as new in Hazel) that if es performs an effect, then that effect *follows* protocol Ψ. That protocol relates the effect's payload (some values vs) with a resumption condition [Liu et al. 2023] (a predicate on values) describing the desired behaviour of the program when the suspended continuation is later resumed, as illustrated in Figure 4. As such, the type of a protocol is $vs \to (vs \to \text{iProp}) \to \text{iProp}$, where iProp is the type of Iris propositions. $\Psi\ v\ R$ is thus a proposition that should hold whenever an effect is performed (site *a* in Figure 4) with payload v, and $R(w)$ is expected to hold when the continuation is later resumed (site *c* in Figure 4) with argument w. This resumption condition must be established when suspending (site *a* in Figure 4) and can be relied upon when handling (site *b* in Figure 4). We give a few examples below of what protocols can look like in practice.

Logical Effects. In Iris-WasmFX, `suspend` is not a logical value (in fact, we retain the same definition of logical value as Iris-Wasm), but a *logical effect*, defined to be of the form `suspendE`, `switchE` or `throwE`. For instance, effect `suspendE` vs $addr$ H represents expression $H[$`suspend.addr` vs $addr]$ where H is syntactically enforced to not contain a `prompt` that captures the tag with address $addr$ (otherwise, the expression would not be stuck). Similarly, `switchE` and `throwE` represent a `switch.addr` or a `throw` (from the exception handling suite) under a sufficiently shallow context.

Our Extended Weakest Precondition. In Iris-WasmFX, each `tag` gets its own protocol, so our extended weakest precondition uses a *metaprotocol* Ψ instead of a simple protocol. A metaprotocol is defined as the combination of three maps: the first is used for `suspend` effects, on which we focus now, and reused for `switch` effects; the second only for `switch` effects, which we return to in §4.4; and the last for exception handling. The first map in a metaprotocol maps tag addresses to protocols. In this section, when we write Ψ $addr$, we mean application of the first map in metaprotocol Ψ.

Hence, our custom extended weakest-precondition is of the form

$$\mathsf{ewp}\ es\,;F\ \langle\Psi\rangle\ \{w\ F', \Phi(w, F')\}$$

where the Wasm frame F comes alongside the expression es, the metaprotocol Ψ is given to specify the intended behaviour in case of an effect being performed, and the postcondition Φ is a predicate on final values w and final frames F'.

Practical Protocols. While in principle a protocol can be any predicate at all, de Vilhena and Pottier [2021] define a specific constructor inspired by session types [Honda et al. 1998] that is sufficient for many practical examples: $!x(v_1)\langle P\rangle\ ?y(v_2)\langle Q\rangle$. Informally, this says "the suspender *sends* payload v_1 and resources P to the handler; if execution is ever resumed, it expects to *receive* return value v_2 and resources Q". Formally, this protocol applied to values vs and predicate Φ is

$$\exists x, (vs = v_1 * P * \forall y, (Q \mathrel{-\!\!*} \Phi(v_2)))$$

where x, which can be used to describe the sent values, can be mentioned in v_1, P, v_2 and Q; and y, which can be used to describe the received values, can be mentioned in v_2 and Q.

Another useful protocol is the always false 'bottom' protocol $\perp$, which symbolises an effect that should not be performed. In practice, we often use at top-level a metaprotocol where all tags map to the bottom protocol, and only start introducing non-bottom protocols when reasoning about continuations run by way of a `resume`. As we will see in the next section, the reasoning rule for `resume` lets the verifier use a different protocol in the premises than in the conclusion.

Running Example Protocol. For our coroutine example, the tag does not expect any values (and this is enforced by typing), but the resumption condition is that the invariant has been restored. Concretely, the metaprotocol Ψ_0 is the bottom protocol $\perp$ for all tags, except the tag address corresponding to local index $t, for which the protocol is $!()\{I\}\ ?()\{I\}$.

Definition. We define the extended weakest precondition for Iris-WasmFX formally in Figure 5. We give an informal reading before explaining the more technical elements.

- When es is a value, we check (as usual) that the postcondition holds.
- When es is a suspend effect, that is, a `suspend.addr` with some values vs and a tag address $addr$, under a (capture-avoiding) context H, we mandate that the protocol $\Psi(addr)$ must be followed. If that protocol is of the form $!x(v)\langle P\rangle\ ?y(w)\langle Q\rangle$, this means there must exist an x such that the payload `immV` vs is equal to v and we have the resources P; and we

$$
\begin{array}{lll}
 & & \text{ewp}\ es\,;F\ \langle\Psi\rangle\ \{w\ F, \Phi(w,F)\} = \text{match to\_val}(es)\ \text{with} \\
\text{value} & & \mid \text{Some}\ w \implies\ \models\!\!\!\Rrightarrow \Phi(w,F) \\
 & & \mid \text{None} \implies \text{match to\_eff}(es)\ \text{with} \\
\text{effect:}\ \ \text{suspend} & & \quad \mid \text{Some}\ (\texttt{suspendE}\ vs\ addr\ H) \implies \\
 & & \quad\quad \uparrow (\Psi\ addr)(\texttt{immV}\ vs)\ (\lambda w, \triangleright\, \text{ewp}\ H[w]\,;F\ \langle\Psi\rangle\ \{w\ F, \Phi(w,F)\}) \\
\qquad\qquad \text{switch} & & \quad \mid \text{Some}\ (\texttt{switchE}\ vs\ kaddr\ ft\ taddr\ H) \implies \text{see §4.4} \\
\text{neither} & & \quad \mid \text{None} \implies \forall S, \text{state\_interp}(S) \Rrightarrow \\
 & & \quad\quad (\exists S'\ F'\ es', (S,F,es) \hookrightarrow (S',F',es')) * \\
 & & \quad\quad \forall S'\ F'\ es', (S,F,es) \hookrightarrow (S',F',es') \Rrightarrow \\
 & & \quad\quad\quad \triangleright(\text{state\_interp}(S') * \text{ewp}\ es'\,;F'\ \langle\Psi\rangle\ \{w\ F, \Phi(w,F)\})
\end{array}
$$

Fig. 5. The extended weakest precondition of Iris-WasmFX

must guarantee that as long as we are provided with a y such that we are given return value w and the resources Q, we can continue execution safely, i.e. we have a weakest precondition statement for $H[w]$. This is the point that motivated moving the payload into the `suspend.addr` instruction, as per §3.3: inspecting the expression without knowledge of the store needs to be enough to identify that it is an effect and what its payload is.

- We explain the case where *es* is a switch effect in §4.4
- When *es* is neither a value nor an effect, we require (as usual) that for any *valid* store S, configuration (S, F, es) is reducible, and for any configuration (S', F', es') that it reduces to, S' is still valid, and the extended weakest precondition holds of (F', es').

Formal Notation. In Figure 5, the update modality $\Rrightarrow$ means that Iris ghost state can be updated — a reader unfamiliar with Iris can think of it as a regular implication. The later modality $\triangleright P$ means that P holds after one execution step; this is crucial to avoid cyclicity in the recursive definition, but can be ignored by readers unfamiliar with Iris. Functions to_val and state_interp are language-specific functions present in the language-agnostic definition of a weakest precondition in Iris. The first attempts to transform an expression into a value and returns None if it cannot, and the second describes what a *valid* store is. We use mostly unchanged definitions from Iris-Wasm. Third, the function to_eff is a language-specific function in Hazel's definition of the extended weakest precondition. It attempts to transform an expression into an effect and returns None if the expression is not an effect. We provide our own definition for it in the context of WasmFX. Finally, the upwards closure of the protocol $\uparrow \Psi$ allows strengthening the resumption condition ($\uparrow \Psi\ v\ \Phi$ is defined as $\exists \Phi', \Psi\ v\ \Phi' * (\forall w, \Phi'(v) \mathrel{-\!\!*} \Phi(v))$), which helps in practice but can be ignored for simplicity [de Vilhena and Pottier 2021, §3.4].

4.3 Proof Rules

In this section, we show the reasoning rules for the new instructions of WasmFX. We focus our presentation on the most crucial rules; see our Rocq development for a full view of the logic, including failure rules, exception handling, the `cont.bind` and `resume_throw` instructions, etc.

Building Continuations. The proof rule for `cont.new` is shown in Figure 6. It mirrors its reduction rule: using the frame to translate the function type parameter of the `cont.new` function, the rule allows reasoning about creating a continuation from a function reference. This rule is the only one that creates a new continuation resource from scratch.

EWP-CONTNEW

$$\frac{F.\mathtt{inst.types}[i] = \mathit{ft}}{\mathsf{ewp}\ [\mathtt{ref.func}\ \mathit{addr}; \mathtt{cont.new}\ i]\ ; F\ \langle \Psi \rangle \left\{ w\ F',\ \begin{array}{c} \exists \mathit{kaddr}, w = \mathtt{immV}\ [\mathtt{ref.cont}\ \mathit{kaddr}] * F' = F * \\ \mathit{kaddr} \xmapsto{\mathsf{wcont}} \\ (\mathit{ft}\ ,\ [\_] +\!\!+ [\mathtt{ref.func}\ \mathit{addr}; \mathtt{ref.call}\ \mathit{ft}]) \end{array} \right\}}$$

EWP-SUSPEND

$$\frac{\begin{array}{c} |\mathit{vs}| = |\mathit{ts_1}| * \mathit{addr} \xmapsto{\mathsf{wtag}}_q (\mathit{ts_1} \to \mathit{ts_2}) * F.\mathtt{inst.tags}[i] = \mathit{addr} * \\ \triangleright \left(\mathit{addr} \xmapsto{\mathsf{wtag}}_q (\mathit{ts_1} \to \mathit{ts_2}) \mathrel{-\!\!*} \uparrow (\Psi\ \mathit{addr})\ (\mathtt{immV}\ \mathit{vs})\ (\lambda v, \triangleright \Phi\ v\ F) \right) \end{array}}{\mathsf{ewp}\ \mathit{vs} +\!\!+ [\mathtt{suspend}\ i]\ ; F\ \langle \Psi \rangle\ \{v\ F, \Phi(v, F)\}}$$

EWP-RESUME

$$\frac{\begin{array}{lc} (1) & F.\mathtt{inst.types}[i] = \mathit{ts_1} \to \mathit{ts_2}\ *\ |\mathit{vs}| = |\mathit{ts_1}|\ *\ F \vdash \mathit{ccs} \to \mathit{dccs}\ * \\ (2) & (\forall \mathit{addr}, \mathit{addr} \notin \mathit{dccs} \implies \Psi(\mathit{addr}) = \Psi'(\mathit{addr}))\ *\ \mathit{addr} \xmapsto{\mathsf{wcont}} (\mathit{ts}_1 \to \mathit{ts}_2, H)\ * \\ (3) & (\forall F, \neg\Phi(\mathtt{trapV}, F))\ *\ \neg\Phi'(\mathtt{trapV})\ *\ \triangleright \mathsf{ewp}\, H[\mathit{vs}]\ ; \varnothing\ \langle \Psi \rangle\ \{w\ F, \Phi(w, F)\}\ * \\ (4) & \triangleright(\forall w, \Phi(w, \varnothing) \mathrel{-\!\!*} \mathsf{ewp}\ [\mathtt{prompt}_{\mathit{ts}_2}\{\mathit{dccs}\}\ w\ \mathtt{end}]\ ; \varnothing\ \langle \Psi' \rangle\ \{v\ \_, \Phi'(v)\})\ * \\ (5) & \triangleright(\forall \mathit{dcc} \in \mathit{dccs}, \langle \Psi \rangle\ \{\Phi\}\ \mathit{dcc};\ \mathit{ts}_2\ \langle \Psi' \rangle\ \{\lambda v\ \_, \Phi'(v)\}) \end{array}}{\mathsf{ewp}\ \mathit{vs} +\!\!+ [\mathtt{ref.cont}\ \mathit{addr}; \mathtt{resume}\ i\ \mathit{ccs}]\ ; F\ \langle \Psi' \rangle\ \{v\ F', \Phi'(v)\ *\ F' = F\}}$$

Fig. 6. The reasoning rule for the WasmFX instructions. See §4.3 and §4.4 for the definition of (5)

Suspend. The rule for `suspend` is shown in Figure 6. It merely appeals to the protocol corresponding to the tag. The last premise indicates that if given back the tag resource, the payload vs should follow the specified protocol, and if the continuation is ever resumed, postcondition Φ will hold. To understand what this means in practice, let us consider the case where protocol $\Psi(addr)$ has form $!x(v)\{P\}\ ?y(w)\{Q\}$. In that case, we must show that $\exists x, vs = v * P * \forall y, Q \mathrel{-\!\!*} \Phi(w, F)$, i.e. that there exists an x such that payload vs is equal to v, we have resource P and for all y, having Q is enough to guarantee the postcondition $\Phi(w, F)$. Note that the frame F in $\Phi(v, F)$ is the same as the one in the conclusion of the rule: when the continuation is resumed and control flow comes back to the current point, we are back in the same environment and should thus use the same frame F as before the `suspend`.

Running Example. We can now describe our proof of \$yield: we want to show

$$\begin{array}{l} \forall F\ I\ q, I \mathrel{-\!\!*}\ \mathit{addryield} \xmapsto{\mathsf{wf}} \mathit{cl}_{\mathrm{yield}} \mathrel{-\!\!*}\ \mathit{addrt} \xmapsto{\mathsf{wtag}}_q ([] \to []) \mathrel{-\!\!*} \\ \qquad \mathsf{ewp}\ [\mathtt{invoke}\ \mathit{addryield}]\ ; F \left\langle \Psi_0 \right\rangle \left\{ w\ F',\ \begin{array}{l} w = \mathtt{immV}\ [] * F = F' * I * \\ \mathit{addryield} \xmapsto{\mathsf{wf}} \mathit{cl}_{\mathrm{yield}} * \mathit{addrt} \xmapsto{\mathsf{wtag}}_q ([] \to []) \end{array} \right\} \end{array}$$

where $addryield = F.\mathtt{inst.funcs}[\$yield]$, $addrt = F.\mathtt{inst.tags}[\$t]$, and cl_{yield} is the closure of \$yield. We phrase this specification using our extended weakest precondition, with metaprotocol Ψ_0. We begin by applying Iris-Wasm rules to unfold the function invocation and bind onto the code of the function. This requires using the function closure resources, and gives it back. We show this in detail in Appendix B. Next, we reason about the code of the function itself, which consists of a single `suspend` for which rule EWP-SUSPEND requires the tag resource, which we have, and it remains to show that when given back the tag resource, protocol $\Psi_0(\$t)$ is followed. This protocol requires us, for the sending, to show that the payload is the empty list, which is straightforward, and to provide I, which is the precondition; and for the receiving, to show that the postcondition for `immV` [] and F can be established from I. Since we hold both resources, it is easy to conclude.

Resume. We show the rule for the `resume` instruction in Figure 6. The premises can be divided into three parts. First, we want the code being resumed to be safe to run. This is rendered with an extended weakest precondition on line (3). The H in that premise is tied by the continuation resource relinquished (to symbolise that the continuations are one-shot) on line (2) to the address *addr* that `resume` took as an argument from the stack, and per line (1) the right number of values *vs* are taken from the stack. Note that the metaprotocol used on line (3) can differ from the one in the conclusion (though, as per line (2), only for tags handled by the `resume`): this is because the effects handled by the clauses of the `resume` can now be triggered in the resumed code.

Second, line (4) covers the 'normal return' case, when the continuation terminates on a value w. We know that in that case w satisfies postcondition Φ, and must show that this is enough to safely plug w into the `prompt` environment the resumed continuation was running in.

Third, line (5) covers the effect case: for each clause in the list of clauses specified by the `resume`, we must show that the effect handled in the clause is safe to run. We capture this by defining a *clause triple* which each clause has to respect. We explain how this clause triple is defined for a `suspend` effect now, and we come back to the `switch` effect in §4.4.

Clause Triple. When a `suspend` is performed, reduction rule REDUCE-SUSPEND mandates that a continuation is created out of the state of the stack, and the handler reduces to a `br` instruction with the label specified by the clause from the `resume` instruction, with the effect's payload and the newly created continuation on the stack. Hence, we wish to require that in that case, we hold an extended weakest precondition statement for $vs +\!\!+ $ [`ref.cont` *kaddr*; `br` *ilab*], where *kaddr* is the address of the newly created continuation. Little is known about the form of that continuation, except that the `suspend` instruction will have followed a protocol from metaprotocol Ψ. Accordingly, the clause triple for a `suspend` (which closely follows that of Hazel [de Vilhena and Pottier 2021, Fig. 7]) requires us to prove the extended weakest precondition statement on the `br` using a continuation resource for the new continuation, and knowledge that the effect was performed in a way that follows the protocol Ψ *taddr*. Since the continuation H is universally quantified, the only way to reason about resuming the continuation again is by knowing that the protocol is followed:

$$
\begin{aligned}
&\langle \Psi \rangle\, \{\Phi\}\ \mathsf{on}\ \mathit{taddr}\ \mathit{ilab};\ \ \mathit{ts}\, \langle \Psi' \rangle\, \{\Phi'\} = \\
&\quad \exists \mathit{ts}_1\ \mathit{ts}_2\ q, \mathit{taddr} \xmapsto{\texttt{wtag}}_q (\mathit{ts}_1 \longrightarrow \mathit{ts}_2) * \\
&\quad \forall \mathit{vs}\ \mathit{kaddr}\ H,\ \ \mathit{kaddr} \xmapsto{\texttt{wcont}} (\mathit{ts}_2 \longrightarrow \mathit{ts}, H) \mathrel{-\!\!*} \mathit{taddr} \xmapsto{\texttt{wtag}}_q (\mathit{ts}_1 \longrightarrow \mathit{ts}_2) \mathrel{-\!\!*} \\
&\qquad \uparrow (\Psi\ \mathit{taddr})\ (\texttt{immV}\ \mathit{vs})\ (\lambda w, \triangleright\, \mathsf{ewp}\, H[w]\,;\varnothing\, \langle \Psi \rangle\, \{w\ F, \Phi(w, F)\}) \mathrel{-\!\!*} \\
&\qquad\quad \mathsf{ewp}\ \mathit{vs} +\!\!+ [\texttt{ref.cont}\ \mathit{kaddr}; \texttt{br}\ \mathit{ilab}]\,;\varnothing\, \langle \Psi' \rangle\, \{w\ F, \Phi'(w, F)\}
\end{aligned}
$$

Frame Agnosticity. The last two lines of the definition of the clause triple refer to an empty frame. According to the operational semantics, the frame to be used in this position is the frame as it would be at the suspension site (for the last line) and re-resumption site (for the penultimate line), but when scrutinising a `resume` instruction, the frame at those later times is unknown. Fortunately, the choice of the frame to use at these points is irrelevant since continuations always reduce in a frame-agnostic way, as mentioned in §4.1. Therefore, we can choose whatever frame we want. (This does not work for exceptions, so we define much stronger rules for the exception handling suite and the `resume_throw` instruction. We leave it as a future work to refine these rules, perhaps using adjoint separation logic [Wagner et al. 2025].) We find it convenient to use the empty frame everywhere (including lines (3–4) in EWP-RESUME), so that all the frames match.

To link the frame F in the conclusion of EWP-RESUME with the empty frame in premise (3), we show that if *es* reduces frame-agnostically and is safe to run under the empty frame, then it is safe to run with any frame, ends with an unchanged frame, and the same post-condition.

Proving the Resume Rule. To prove soundness of the EWP-RESUME rule above, we unfold the operational semantics one step, which reduces `resume` to a `prompt` instruction. As discussed, binding into a `prompt` using the simple EWP-LABEL rule would be unsound, hence we define a bind rule specific to `prompt`, as we discuss in Appendix D. The premises of this rule are similar to those in EWP-RESUME: a premise for the body, a premise for the 'normal return' case, and a premise for the handled effects using clause triples.

4.4 Reasoning About the Switch Instruction

Given that the operational semantics for the `switch` instruction is intuitively just a suspension immediately followed by the resumption of a different continuation, one might expect that the insertion of `switch` into the logic would be straightforward and that the corresponding rule would merely be a combination of the other rules. However, this is not the case, because we want to reason at a different site. When reasoning about separate `resume` and `suspend`, what executes after the `suspend` is static: it is determined by the code around the `resume` and by the values it gets passed from the `suspend`, which is described by the protocol. On the other hand, with a `switch`, what executes next is dynamic: it is determined by the argument to the `switch`, which we want, for modularity, to be able to reason about at the site of `switch`, not at the site of the `resume`.

To address this, we use an extra component in our metaprotocols. For tags used for `switch`, the metaprotocol specifies not only a protocol, but also a predicate on continuations that the `switch` instruction is meant to switch to. We write $\Psi^1\ \mathit{taddr}$ and $\Psi^2\ \mathit{taddr}$ respectively for the protocol and for the predicate. Using this, we now complete the definition of our extended weakest precondition from Figure 5 with the case for `switch`:

$$
\begin{array}{l}
\mid \mathsf{Some}\ (\mathsf{switchE}\ \mathit{vs}\ \mathit{kaddr}\ \mathit{ft}\ \mathit{taddr}\ H) \implies \exists H'\ \mathit{ts}_1\ \mathit{ts}_2\ \mathit{ft}'\ \mathit{ts}\ q, \\
\qquad \mathit{taddr} \xmapsto{\mathsf{wtag}}_q [\,] \to \mathit{ts}\ \ast\ \mathit{kaddr} \xmapsto{\mathsf{wcont}} (\mathit{ft}, H')\ \ast\ \mathit{ft}' = \mathit{ts}_1 \to \mathit{ts}\ \ast \\
\qquad \mathit{ft} = (\mathit{ts}_1 \mathbin{+\!\!+} [\mathit{ft}'.\mathsf{cont}]) \to \mathit{ts}_2\ \ast\ \Psi^2\ \mathit{taddr}\ H'\ \ast \\
\qquad \Big(\ \mathit{taddr} \xmapsto{\mathsf{wtag}}_q [\,] \to \mathit{ts}\ \mathrel{-\!\!\ast} \uparrow (\Psi^1\ \mathit{taddr})\ \mathit{vs}\ (\lambda\, w, \triangleright\, \mathsf{ewp}\, H[w]\,;F\,\langle\Psi\rangle\,\{\Phi\})\Big)
\end{array}
$$

In order to switch, we need to give up a tag resource and a continuation resource corresponding to the continuation we are switching to; we need to know that the types have the required form; and that the metaprotocol is being followed, meaning that the continuation satisfied the required predicate, and when given back the tag resource, the payload follows the protocol. The continuation resource is not given back, because WasmFX continuations are one-shot.

The proof rule for the `switch` instruction itself simply mirrors the definition above.

EWP-SWITCH

$$
\frac{\begin{array}{c}
\mathit{ft} = \mathit{ts}_1 \to \mathit{ts}\ \ast\ F.\mathsf{inst.types}[i] = \mathit{ft}'\ \ast\ \mathit{ft}' = (\mathit{ts}_1 \mathbin{+\!\!+} [\mathit{ft}.\mathsf{cont}]) \to \mathit{ts}_2\ \ast\ F.\mathsf{inst.tags}[i'] = \mathit{taddr} \ast \\
|\mathit{vs}| = |\mathit{ts}_1|\ \ast\ \mathit{taddr} \xmapsto{\mathsf{wtag}}_q [\,] \to \mathit{ts}\ \ast\ \mathit{kaddr} \xmapsto{\mathsf{wcont}} (\mathit{ft}', H)\ \ast\ \Psi^2\ \mathit{taddr}\ H \ast \\
\triangleright(\mathit{taddr} \xmapsto{\mathsf{wtag}}_q [\,] \to \mathit{ts} \mathrel{-\!\!\ast} \uparrow (\Psi^1\ \mathit{taddr})\ \mathit{vs}\ \Phi)
\end{array}}{\mathsf{ewp}\ \mathit{vs} \mathbin{+\!\!+} [\mathsf{ref.cont}\ \mathit{kaddr}; \mathsf{switch}\ i\ i']\,;F\,\langle\Psi\rangle\,\{\Phi\}}
$$

Recall that the 'effect' premise of the EWP-RESUME rule uses the clause triple to require that for all clauses, if an effect is triggered targeting that clause, continuing execution is safe. We can now give the following definition of a clause triple for a `switch` clause:

$$
\begin{array}{l}
\langle\Psi\rangle\,\{\Phi\}\ \mathsf{on}\ \mathit{taddr}\ \mathsf{switch};\ \mathit{ts}\,\langle\Psi'\rangle\,\{\Phi'\} = \\
\quad \exists q, \mathit{taddr} \xmapsto{\mathsf{wtag}}_q ([\,] \to \mathit{ts})\ \ast \\
\quad \square\forall \mathit{vs}\ \mathit{kaddr}\ H\ H'\ \mathit{ts}_1,\ \mathit{kaddr} \xmapsto{\mathsf{wcont}} (\mathit{ts}_1 \to \mathit{ts}, H) \mathrel{-\!\!\ast} \Psi^2\ \mathit{taddr}\ H' \mathrel{-\!\!\ast} \\
\qquad \uparrow (\Psi^1\ \mathit{taddr})\ (\mathsf{immV}\ \mathit{vs})\ (\lambda w, \triangleright\, \mathsf{ewp}\, H[w]\,;\varnothing\,\langle\Psi\rangle\,\{w\ F, \Phi(w,F)\}) \mathrel{-\!\!\ast} \\
\qquad\quad \mathsf{ewp}\, H'[\mathit{vs} \mathbin{+\!\!+} [\mathsf{ref.cont}\ \mathit{kaddr}]]\,;\varnothing\,\langle\Psi\rangle\,\{w\ F, \Phi(w,F)\}
\end{array}
$$

In the case of a `switch`, the operational semantics mandates that execution continues by running the continuation given to `switch` as an argument with the payload taken from the stack and a newly created continuation corresponding to the current environment. Hence we must establish an extended weakest precondition for running that new continuation, when given an appropriate payload and continuation. The $\square$ ensures duplicability, because the new continuation is run under the same clauses and hence we need the clause triple to hold again for the new continuation.

We have used these rules to reason about simple examples using `switch`, these can be found in our Rocq development. We note that realistic programs using the `switch` instructions typically require recursive types, which we do not support. We leave this as future work, as it is orthogonal to supporting the WasmFX operations themselves.

4.5 Soundness and Adequacy

All proof rules showed in this paper have been proved sound in the Rocq proof assistant; that is, we show that holding all premises is sufficient to obtain the extended weakest precondition statement in the conclusion, as defined in Figure 5. Our Rocq development also includes proof rules for every instruction of WasmFX.

THEOREM 4.1 (SOUNDNESS). *All proof rules displayed in this paper have been proven sound with respect to WasmFXCert in the Rocq proof assistant.*

We also prove *adequacy*, which allows us to extract, from each Iris-WasmFX proof of a pure extended weakest precondition, a theorem phrased purely in terms of the operational semantics, making no mention of Iris. The proof is similar to that of plain Iris [Jung et al. 2018, §6.4].

THEOREM 4.2 (ADEQUACY). *If* $\text{state\_interp}(S) \ast\ \mathit{ewp}\ \mathit{es}\,;F\ \langle\bot\rangle\ \{w\ F, \Phi(w,F)\}$ *for some* pure *predicate* Φ *(meaning* Φ *does not contain any resourceful Iris propositions), then for any reduction trace* $(S, F, \mathit{es}) \hookrightarrow^\star (S', F', \mathit{vs})$ *(where vs are values), it holds that* $\Phi(\mathit{vs}, F')$*.*

5 Case Studies

In this section, we briefly consider two canonical features with which we exemplify the kind of modular reasoning that Iris-WasmFX enables. Coroutines illustrate switching of stacks, where the tag is internal to the library, and passes through the client. Generators illustrate passing of values with tags, where the tag is used as the interface between the library and client to *send* values from the generator to the client. Full details are given in Appendix E.

Coroutines. We now precisely state a modular specification for our coroutine library using the protocol of Section 4.2, and reason about the client module with respect to that specification, as desired, and use adequacy to derive a statement in terms of the operational semantics, without reference to Iris:

THEOREM 5.1. *If Cor is the coroutine module whose code is shown in Figure 3, and Cl is the message-passing client module, and host program* [`instantiate` *Cor*; `instantiate` *Cl*; `invoke` *addrmain*] *terminates on a value w, then w is of the form* $w = \texttt{immV}\ [\mathit{vala}; \mathit{valb}]$*, and if vala is 1, then valb is 42.*

Generators. Phipps-Costin et al. [2023] use effects to implement a generator of the naturals: 0, 1, 2, ... The generator consists of an infinite loop which suspends with the value of the current counter, and then increments it. Every time a client resumes the generator, they get the next natural number. They use this generator to implement a function $sum_until, which adds all naturals from 0 to a given number. Again, we give a modular proof: we first verify the generator, giving it a generic specification, and then verify the client in terms of that specification, to show:

Theorem 5.2. *For any i32 n, if Gen is the generator module containing the generator of the naturals and the* `$sum_until` *function, and if the host program* [`instantiate` *Gen*; `invoke` *addrsum_until n*] *terminates on a value w, then* $w = \texttt{immV}\left[\sum_{i=0}^{n} i\right]$ *(in i32 arithmetic).*

6 Related Work

Iris-WasmFX is, to our knowledge, the first program logic for WasmFX, and as such pulls together several strands of research:

Formalising WasmFX. There is an ongoing project [Liang et al. 2025] to model WasmFX in SpecTec [Youn et al. 2024], a domain-specific language designed specifically to write the semantics of Wasm with tooling to export theorem prover definitions. SpecTec has been officially adopted for authoring the Wasm specification [Rossberg 2025] (from 3.0 on), and could eventually replace the hand-written definitions of WasmFXCert. However, this would require more mature theorem prover output, and would in any case not replace the proofs of §3.4. Moreover, it raises the question of how to handle the modifications we make to the language definition for the sake of compatibility with the theorem prover and program logic, as described in §3.3.

Program Logics for Effect Handlers. Iris-WasmFX applies the ideas of Hazel, a program logic for an ML-style calculus with effect handlers, to a full-fledged programming language, Wasm. Osiris [Daby-Seesaram et al. 2023; Seassau et al. 2025a,b] is a program logic for OLang, a substantial fragment of another full-fledged programming language that features effect handlers, namely sequential OCaml 5.3. Osiris has a sibling logic, Horus, for pure expressions, that makes reasoning simpler for them, and is compatible with Osiris; we have not explored this angle for Wasm, as it is much more imperative, but the idea could nevertheless prove useful. The two projects differ in the challenges they face: one of the main sources of complexity of OLang is the loose evaluation order of OCaml, which Wasm was designed to avoid; on the other hand, the functional style of OLang integrates with effect handlers in a more human-readable way than in WasmFX. They also differ in their implementations: OLang is defined by elaboration into a monad, whereas WasmFXCert follows the usual operational semantics style, as used in the Wasm standard. The combined existence of Osiris and Iris-WasmFX opens up the possibility of formally relating programs in OCaml and Wasm, be it by verified compilation, translation validation, etc.

Integration with JavaScript. The extensive use of non-local control flow in JavaScript/ECMAScript is another motivation for extending Wasm with WasmFX. Khayam et al. [2022] formalise parts of ECMAScript and describe how, even though they do not model generators and asynchronous function definitions, they identified corner cases of the language semantics to do with manipulation of the evaluation context, showcasing the challenges raised by non-local control flow, and how precise definitions and formal verification can be helpful in this context.

JavaScript Promise Integration (JSPI) [McCabe et al. 2024] is the 'reverse' of WasmFX: it allows plain Wasm, not written to be asynchronous, to work with host code (in JavaScript) that uses promises. WasmFXCert is the first building block to consider the semantics of Wasm running in a host language with its own non-local control flow features, and to study their interaction. In particular, part of the contract between JavaScript and Wasm is that Wasm is only meant to suspend or resume the execution of JavaScript via promises. Verifying this is outside of the scope of this paper, but we lay the foundations to do so.

Compilation to WasmFX. RichWasm [Fitzgibbons et al. 2024] is a richly typed language based on Wasm, designed as a target for typed compilation enforcing strong memory safety guarantees. Extending RichWasm to cover WasmFX, as modelled by WasmFXCert, could offer a useful staging post in proving correctness of the compilation of non-local control features into WasmFX.

7 Conclusion

By developing WasmFXCert and Iris-WasmFX, we have validated the design of the WasmFX proposal in time to feed back into its development. In addition, with Iris-WasmFX, we have laid the foundations for verifying effect-based libraries.

Acknowledgments

Ohad Kammar, for helpful discussions. Andreas Rossberg, for comments on a draft of this paper. Amal Ahmed, for ... JP: AUFF LB: Villum???

References

Heejin Ahn and Ben Titzer. 2022. *Exception Handling Proposal for WebAssembly.* Technical Report. https://webassembly.github.io/exception-handling/

Stephen Brookes and Peter W. O'Hearn. 2016. Concurrent separation logic. *ACM SIGLOG News* 3, 3 (2016), 47–65. doi:10.1145/2984450.2984457

Arnaud Daby-Seesaram, François Pottier, and Armaël Guéneau. 2023. Osiris: an Iris-based program logic for OCaml. OCaML'23: Caml Users and Developers Workshop 2023. https://icfp23.sigplan.org/details/ocaml-2023-papers/7/Osiris-an-Iris-based-program-logic-for-OCaml.

Paulo de Vilhena. 2022. *Proof of Programs with Effect Handlers. (Preuve de Programmes avec Effect Handlers).* Ph. D. Dissertation. Paris Cité University, France. https://tel.archives-ouvertes.fr/tel-03891381

Paulo Emílio de Vilhena and François Pottier. 2021. A separation logic for effect handlers. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–28. doi:10.1145/3434314

Paulo Emílio de Vilhena and François Pottier. 2023. Verifying an Effect-Handler-Based Define-By-Run Reverse-Mode AD Library. *Log. Methods Comput. Sci.* 19, 4 (2023). doi:10.46298/LMCS-19(4:5)2023

Edsger W. Dijkstra. 1968. Letters to the editor: go to statement considered harmful. *Commun. ACM* 11, 3 (1968), 147–148. doi:10.1145/362929.362947

Matthias Felleisen. 1988. The Theory and Practice of First-Class Prompts. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*, Jeanne Ferrante and Peter Mager (Eds.). ACM Press, 180–190. doi:10.1145/73560.73576

Jean-Christophe Filliâtre and Mário Pereira. 2016. A Modular Way to Reason About Iteration. In *NASA Formal Methods - 8th International Symposium, NFM 2016, Minneapolis, MN, USA, June 7-9, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9690)*, Sanjai Rayadurgam and Oksana Tkachuk (Eds.). Springer, 322–336. doi:10.1007/978-3-319-40648-0_24

Michael Fitzgibbons, Zoe Paraskevopoulou, Noble Mushtak, Michelle Thalakottur, Jose Sulaiman Manzur, and Amal Ahmed. 2024. RichWasm: Bringing Safe, Fine-Grained, Shared-Memory Interoperability Down to WebAssembly. *Proc. ACM Program. Lang.* 8, PLDI (2024), 1656–1679. doi:10.1145/3656444

Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and J. F. Bastien. 2017. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 185–200. doi:10.1145/3062341.3062363

Daniel Hillerström. 2021. Typed Continuations: A Basis for Stack-switching in Wasm. https://raw.githubusercontent.com/WebAssembly/meetings/main/stack/2021/presentations/2021-9-20-TypedContinuations.pdf

Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. 1998. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings (Lecture Notes in Computer Science, Vol. 1381)*, Chris Hankin (Ed.). Springer, 122–138. doi:10.1007/BFB0053567

Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*. 256–269. doi:10.1145/2951913.2951943

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. doi:10.1017/S0956796818000151

Adam Khayam, Louis Noizet, and Alan Schmitt. 2022. A Faithful Description of ECMAScript Algorithms. In *PPDP 2022: 24th International Symposium on Principles and Practice of Declarative Programming, Tbilisi, Georgia, September 20 - 22, 2022.* ACM, 8:1–8:14. doi:10.1145/3551357.3551381

Maxime Legoupil, June Rousseau, Aïna Linn Georges, Jean Pichon-Pharabod, and Lars Birkedal. 2024. Iris-MSWasm: Elucidating and Mechanising the Security Invariants of Memory-Safe WebAssembly. *Proc. ACM Program. Lang.* 8, OOPSLA2 (2024), 304–332. doi:10.1145/3689722

Yalun Liang, Sam Lindley, and Andreas Rossberg. 2025. Experience Report: Stack Switching in Wasm SpecTec. presented at WAW: the WebAssembly Workshop https://effect-handlers.org/static/papers/sss-waw-2025.pdf.

Zongyuan Liu, Sergei Stepanenko, Jean Pichon-Pharabod, Amin Timany, Aslan Askarov, and Lars Birkedal. 2023. VMSL: A Separation Logic for Mechanised Robust Safety of Virtual Machines Communicating above FF-A. *Proc. ACM Program. Lang.* 7, PLDI (2023), 1438–1462. doi:10.1145/3591279

David Madore. 2001. The Unlambda Programming Language. http://www.madore.org/~david/programs/unlambda/.

Luc Maranget, Susmit Sarkar, and Peter Sewell. 2012. A Tutorial Introduction to the ARM and POWER Relaxed Memory Models. http://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf.

Francis McCabe, Sam Lindley, and WebAssembly Community Group. 2025. Stack Switching WebAssembly proposal. https://github.com/WebAssembly/stack-switching/blob/main/proposals/stack-switching/Explainer.md.

Francis McCabe, Thibaud Michaud, Ilya Rezvov, and Brendan Dahl. 2024. Introducing the WebAssembly JavaScript Promise Integration API. https://v8.dev/blog/jspi

Alexandra E. Michael, Anitha Gollamudi, Jay Bosamiya, Evan Johnson, Aidan Denlinger, Craig Disselkoen, Conrad Watt, Bryan Parno, Marco Patrignani, Marco Vassena, and Deian Stefan. 2023. MSWasm: Soundly Enforcing Memory-Safe Execution of Unsafe Code. *Proc. ACM Program. Lang.* 7, POPL (2023), 425–454. doi:10.1145/3571208

Luna Phipps-Costin, Andreas Rossberg, Arjun Guha, Daan Leijen, Daniel Hillerström, K. C. Sivaramakrishnan, Matija Pretnar, and Sam Lindley. 2023. Continuing WebAssembly with Effect Handlers. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 460–485. doi:10.1145/3622814

Gordon D. Plotkin and Matija Pretnar. 2009. Handlers of Algebraic Effects. In *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5502)*, Giuseppe Castagna (Ed.). Springer, 80–94. doi:10.1007/978-3-642-00590-9_7

Gordon D. Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Log. Methods Comput. Sci.* 9, 4 (2013). doi:10.2168/LMCS-9(4:23)2013

François Pottier. 2017. Verifying a hash table and its iterators in higher-order separation logic. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017*, Yves Bertot and Viktor Vafeiadis (Eds.). ACM, 3–16. doi:10.1145/3018610.3018624

Xiaojia Rao, Aïna Linn Georges, Maxime Legoupil, Conrad Watt, Jean Pichon-Pharabod, Philippa Gardner, and Lars Birkedal. 2023. Iris-Wasm: Robust and Modular Verification of WebAssembly Programs. *Proc. ACM Program. Lang.* 7, PLDI (2023), 1096–1120. doi:10.1145/3591265

Andreas Rossberg. 2019. WebAssembly (Release 1.0). https://webassembly.github.io/spec/. Accessed 2020-01-01.

Andreas Rossberg. 2020. Issue #1359: Typed continuations to model stacks. https://github.com/WebAssembly/design/issues/1359

Andreas Rossberg. 2023. WebAssembly (Release 2.0). https://webassembly.github.io/spec/. Accessed 2023-20-02.

Andreas Rossberg. 2025. SpecTec has been adopted. https://webassembly.org/news/2025-03-27-spectec/

Andreas Rossberg, Daan Leijen, Daniel Hillerström, KC Sivaramakrishnan, Matija Pretnar Sam Lindley, and Stephen Dolan. 2020. Stacks and Continuations for Wasm — Idea Sketch. https://github.com/WebAssembly/meetings/blob/master/main/2020/presentations/2020-02-rossberg-continuations.pdf Accessed Jul 4 2020.

Jan Schwinghammer, Lars Birkedal, Bernhard Reus, and Hongseok Yang. 2009. Nested Hoare Triples and Frame Rules for Higher-Order Store. In *Computer Science Logic, 23rd international Workshop, CSL 2009, 18th Annual Conference of the EACSL, Coimbra, Portugal, September 7-11, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5771)*, Erich Grädel and Reinhard Kahle (Eds.). Springer, 440–454. doi:10.1007/978-3-642-04027-6_32

Remy Seassau, Irene Yoon, Jean-Marie Madiot, and François Pottier. 2025a. Formal Semantics & Program Logics for a Fragment of OCaml. In *ICFP*.

Remy Seassau, Irene Yoon, Jean-Marie Madiot, and François Pottier. 2025b. Osiris: Towards Formal Semantics and Reasoning for OCaml.

Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM* 53, 7 (2010), 89–97. doi:10.1145/1785414.1785443

Amin Timany and Lars Birkedal. 2019. Mechanized relational verification of concurrent programs with continuations. *Proc. ACM Program. Lang.* 3, ICFP (2019), 105:1–105:28. doi:10.1145/3341709

Andew Wagner, Zachary Eisbach, and Amal Ahmed. 2025. An Adjoint Separation Logic for the Wasm Call Stack. presentation at Discussions at Dagstuhl Seminar 25241: Utilising and Scaling the WebAssembly Semantics. https://www.andrewwagner.io/assets/slides/adj-wasm-dagstuhl.pdf

Conrad Watt, Xiaojia Rao, Jean Pichon-Pharabod, Martin Bodin, and Philippa Gardner. 2021. Two Mechanisations of WebAssembly 1.0. In *Formal Methods - 24th International Symposium, FM 2021, Virtual Event, November 20-26, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 13047)*, Marieke Huisman, Corina S. Pasareanu, and Naijun Zhan (Eds.). Springer, 61–79. doi:10.1007/978-3-030-90870-6_4

Dongjun Youn, Wonho Shin, Jaehyun Lee, Sukyoung Ryu, Joachim Breitner, Philippa Gardner, Sam Lindley, Matija Pretnar, Xiaojia Rao, Conrad Watt, and Andreas Rossberg. 2024. Bringing the WebAssembly Standard up to Speed with SpecTec. *Proc. ACM Program. Lang.* 8, PLDI (2024), 1559–1584. doi:10.1145/3656440

Alon Zakai. 2019. Pause and Resume WebAssembly with Binaryen's Asyncify. https://kripken.github.io/blog/wasm/2019/07/16/asyncify.html.

A WasmFXCert AST

$$
\begin{array}{rcl}
\text{(numerical value type) } nt & ::= & \mathbf{i32} \mid \mathbf{i64} \mid \mathbf{f32} \mid \mathbf{f64} \\
\text{(reference value type) } rt & ::= & \boxed{ft.\mathtt{func} \mid \mathtt{exn} \mid ft.\mathtt{cont}} \\
\text{(value type) } t \ ::= \ nt \mid rt & & \text{(function type) } ft \ ::= \ ts \rightarrow ts
\end{array}
$$

$$\text{(immediate) } i, min, max ::= \mathbb{N} \quad (i \text{ for local indices})$$

$$
\begin{array}{rcl}
\text{(exception clause) } ec & ::= & \mathtt{catch}\ i\ i' \mid \mathtt{catch\_ref}\ i\ i' \mid \mathtt{catch\_all}\ i \mid \mathtt{catch\_all\_ref}\ i \\
\text{(continuation clause) } cc & ::= & \mathtt{on}\ i\ i' \mid \mathtt{on}\ i\ \mathtt{switch}
\end{array}
$$

$$
\begin{array}{rl}
\text{(basic instructions) } b ::= & nt.\mathbf{const}\ c \mid t.\mathrm{add} \mid \textit{other stackops} \mid \mathtt{local.\{get/set\}}\ i \mid \\
& \mathtt{global.\{get/set\}}\ i \mid t.\mathtt{load}\ \textit{flags} \mid t.\mathtt{store}\ \textit{flags} \mid \mathtt{memory.size} \mid \\
& \mathtt{memory.grow} \mid \mathtt{block}\ ft\ bs \mid \mathtt{loop}\ ft\ bs \mid \mathtt{if}\ ft\ bs\ bs' \mid \mathtt{br}\ i \mid \mathtt{br\_if}\ i \mid \\
& \mathtt{br\_table}\ \mathbf{is} \mid \mathtt{call}\ i \mid \mathtt{call\_indirect}\ \ i \mid \mathtt{return} \mid rt.\mathtt{null} \mid \\
& \mathtt{ref.is\_null} \mid \mathtt{ref.function}\ i \mid \mathtt{ref.call}\ i \mid \ \mathtt{try\_table}\ i\ ecs\ bs \mid \\
& \mathtt{throw}\ i \mid \mathtt{throw\_ref} \mid \mathtt{cont.new}\ i \mid \mathtt{resume}\ i\ ccs \mid \\
& \mathtt{suspend}\ i \mid \mathtt{switch}\ i\ i' \mid \mathtt{cont.bind}\ i\ i' \mid \mathtt{resume\_throw}\ i\ i'\ ccs
\end{array}
$$

$$
\begin{array}{rlrl}
\text{(functions) } func & ::= \mathtt{func}\ i\ ts\ bs & \text{(tables) } tab & ::= \mathtt{tab}\ min\ max \\
\text{(memories) } mem & ::= \mathtt{mem}\ min\ max & \text{(globals) } glob & ::= \mathtt{glob}\ \mathtt{mutable}\ t\ e_{\mathrm{init}} \\
\text{(elem segments) } elem & ::= \mathtt{elem}\ i\ bs_{\mathrm{off}}\ is & \text{(data segments) } data & ::= \mathtt{data}\ i\ bs_{\mathrm{off}}\ bytes
\end{array}
$$

$$
\begin{array}{rcl}
\text{(import descriptions) } importdesc & ::= & \mathtt{func_i}\ i \mid \mathtt{tab_i}\ min\ max \mid \mathtt{mem_i}\ min\ max \mid \\
& & \mathtt{glob_i}\ \mathtt{mutable}^{?}\ t \mid \mathtt{tag_i}\ i \\
\text{(imports) } import & ::= & \mathtt{import}\ string\ string\ importdesc \\
\text{(export descriptions) } exportdesc & ::= & \mathtt{func_e}\ i \mid \mathtt{tab_e}\ i \mid \mathtt{mem_e}\ i \mid \mathtt{glob_e}\ i \mid \mathtt{tag_e}\ i \\
\text{(exports) } export & ::= & \mathtt{export}\ string\ exportdesc \\
\text{(start) } start & ::= & \mathtt{Some}\ i \mid \mathtt{None}
\end{array}
$$

$$
\text{(modules) } m ::= \left\{ \begin{array}{l} \mathtt{types} : fts,\ \ \mathtt{tags} : fts,\ \mathtt{funcs} : funcs,\ \ \mathtt{globs} : globs,\ \ \mathtt{mems} : mems,\ \ \mathtt{tabs} : tabs, \\ \mathtt{data} : datas,\ \ \mathtt{elem} : elems,\ \ \mathtt{imports} : imports,\ \ \mathtt{exports} : exports,\ \ \mathtt{start} : start \end{array} \right\}
$$

Fig. 7. The AST for structures necessary to evaluate code in WasmFXCert. Black parts are borrowed directly from WasmCert (plain Wasm 1.0). Parts in indigo pertain to the references from Wasm 3.0, parts in orange pertain to exception handling, and parts in magenta are new in WasmFX. Places where WasmFXCert departs from WasmFX are shown $\boxed{\text{boxed}}$.

$$\text{(immediate) } i, addr, max ::= \mathbb{N}$$
$$\text{(} i \text{ refers to local indices, while } addr \text{ refers to addresses in the store)}$$

$$\begin{array}{rcl}
\text{(reference value) } rv & ::= & rt.\mathtt{null} \mid \mathtt{ref.func}\ addr \mid \mathtt{ref.exn}\ addr\ addr' \mid \mathtt{ref.cont}\ addr \\
\text{(value) } v & ::= & nt.\mathtt{const}\ c \mid rv
\end{array}$$

$$\begin{array}{rl}
\text{(module instance) } inst & ::= \{\mathsf{types} : fts,\ \ \mathsf{funcs} : addrs,\ \ \mathsf{globs} : addrs,\ \ \mathsf{mems} : addrs, \\
 & \qquad \mathsf{tabs} : addrs,\ \ \mathsf{tags} : addrs\ \} \\
\text{(frame) } F & ::= \{\mathsf{locs} : vs,\ \ \mathsf{inst} : inst\ \}
\end{array}$$

$$\begin{array}{rcl}
\text{(translated exception clause) } dec & ::= & \mathtt{catch}\ addr\ i \mid \mathtt{catch\_ref}\ addr\ i \mid \\
 & & \mathtt{catch\_all}\ i \mid \mathtt{catch\_all\_ref}\ i \\
\text{(translated continuation clause) } dcc & ::= & \mathtt{on}\ addr\ i \mid \mathtt{on}\ addr\ \mathtt{switch}
\end{array}$$

$$\begin{array}{rcl}
\text{(administrative instruction) } e & ::= & b \mid \mathtt{trap} \mid \mathtt{invoke}\ addr \mid \mathtt{label}_n\{es'\}\ es\ \mathtt{end} \mid \\
 & & \mathtt{frame}_n\{F\}\ es\ \mathtt{end} \mid \mathtt{call\_host}\ V\!ft\ i\ vs \mid \mathtt{ref.func}\ addr \mid \\
 & & \mathtt{ref.exn}\ addr\ addr' \mid \mathtt{handler}\{decs\}\ es\ \mathtt{end} \mid \\
 & & \mathtt{throw\_ref.addr}\ \boxed{vs}\ addr\ addr' \mid \mathtt{ref.cont}\ addr \mid \\
 & & \mathtt{prompt}_{\boxed{ts}}\{dccs\}\ es \mid \mathtt{suspend.addr}\ \boxed{vs}\ addr \\
 & & \mathtt{switch.addr}\ vs\ \boxed{addr}\ ft\ addr' \\
\text{(evaluation context) } L & ::= & \boxed{vs +\!\!+ [\_] +\!\!+ es} \mid vs +\!\!+ \mathtt{label}_n\{es'\}\ L\ \mathtt{end} +\!\!+ es \mid \\
 & & vs +\!\!+ \mathtt{handler}\{decs\}\ L\ \mathtt{end} +\!\!+ es \mid \\
 & & vs +\!\!+ \mathtt{prompt}_{ts}\{dccs\}\ L\ \mathtt{end} +\!\!+ es \\
\text{(handler context) } H & ::= & vs +\!\!+ [\_] +\!\!+ es \mid vs +\!\!+ \mathtt{label}_n\{es'\}\ H\ \mathtt{end} +\!\!+ es \mid \\
 & & vs +\!\!+ \mathtt{frame}_n\{F\}\ H\ \mathtt{end} +\!\!+ es \mid \\
 & & vs +\!\!+ \mathtt{handler}\{decs\}\ H\ \mathtt{end} +\!\!+ es \mid \\
 & & vs +\!\!+ \mathtt{prompt}_{ts}\{dccs\}\ H\ \mathtt{end} +\!\!+ es
\end{array}$$

$$\begin{array}{rl}
\text{(function instances) } finst & ::= \{(inst; ts); bs\}^{\mathtt{NativeCl}}_{ft} \mid \{hidx\}^{\mathtt{HostCl}}_{ft} \\
\text{(table instances) } tinst & ::= \{\mathsf{elem} : is,\ \ \mathsf{max} : max^?\ \} \\
\text{(memory instance) } minst & ::= \{\mathsf{data} : bytes,\ \ \mathsf{max} : max^?\ \} \\
\text{(global instance) } ginst & ::= \{\mathsf{mut} : mutable^?,\ \ \mathsf{value} : v\ \} \\
\text{(exception instance) } einst & ::= \{\mathsf{etag} : addr,\ \ \mathsf{efields} : vs\ \} \\
\text{(continuation instance) } cinst & ::= \mathtt{cont}\ \boxed{ft}\ H \mid \mathtt{dagger}\ \boxed{ft} \\
\text{(store) } S & ::= \{\mathsf{funcs} : finsts,\ \ \mathsf{globs} : ginsts,\ \ \mathsf{mems} : minsts,\ \ \mathsf{tabs} : tinsts, \\
 & \qquad \mathsf{tags} : fts,\ \ \mathsf{exns} : einsts,\ \ \mathsf{conts} : cinsts\ \}
\end{array}$$

$$\begin{array}{rl}
\text{(typing context) } C ::= \{ & \mathsf{types} : fts,\ \ \mathsf{func} : fts,\ \ \mathsf{global} : gts,\ \ \mathsf{table} : tts,\ \ \mathsf{memory} : mts, \\
 & \mathsf{local} : ts,\ \ \mathsf{label} : (ts)s,\ \ \mathsf{return} : ts^?,\ \ \mathsf{exns} : addrs,\ \ \mathsf{tags} : fts\ \}
\end{array}$$

(the meaning of some of the fields in the store is orthogonal to stack switching and is omitted)

Fig. 8. The AST for structures necessary to evaluate code in WasmFXCert. Black parts are borrowed directly from WasmCert (plain Wasm 1.0). Parts in indigo pertain to the references from Wasm 3.0, parts in orange pertain to exception handling, and parts in magenta are new in WasmFX. Places where WasmFXCert departs from WasmFX are shown $\boxed{\text{boxed}}$.

B Frame

Another novelty we bring in Iris-WasmFX with respect to the existing Iris-Wasm program logic is the way we deal with the Wasm frame. In Iris-Wasm, aside from resources corresponding to different parts of the store, there is a resource $\xhookrightarrow{\text{Fr}} F$ that symbolises ownership of the entire frame. For technical reasons,[1] Iris-Wasm mandates that the frame resource must be present in the proof environment every time a reduction step is taken, cluttering the proof rules and leading to the confusing situation where the frame resource appears in proof rules for instructions that in no way interact with the frame.

Instead, our custom weakest precondition considers the frame as part of the expression rather than part of the state. Thus, our first modification to Iris-Wasm's weakest precondition statement is that ours has the shape $\mathsf{wp}\; es\,; F\;\{w\; F, \Phi(w, F)\}$ where the expression es sits alongside the frame F, and the postcondition is a predicate on both the final logical value w and the final frame F.

We illustrate this new weakest precondition statement by showing the bind rule for the `frame` instruction. Having the frame baked into the weakest precondition statement makes it much easier to update the frame in order to run the body of the `frame` in the correct environment:

$$\frac{\mathsf{wp}\;\mathbf{es}\,; F\;\{w\; F', \mathsf{wp}\;[\mathtt{frame}_n\{F'\}\;w\;\mathtt{end}]\,; F_0\;\{v\; F_1, \Phi(v, F_1)\}\}}{\mathsf{wp}\;[\mathtt{frame}_n\{F\}\;es\;\mathtt{end}]\,; F_0\;\{v\; F_1, \Phi(v, F_1)\}}\;\textsc{wp-bind-frame}$$

i.e. when reasoning under frame F_0, if we reach a `frame` instruction (representing a function call being executed), when reasoning about the code inside the `frame`, we use the frame F specified by the frame instruction. Since the frame contains the local variables and the instance (which translates local indices into addresses into the store), this change of frames is what guarantees local state encapsulation. The bind rule above specifies that once the body es has reduced to a value w and the 'inner' frame F has become a frame F', we can then place that value and that frame back in a `frame` instruction and resume reasoning from there.

Example. Let us showcase how the two bind rules above are used and how Iris-WasmFX deals with the frame, by going through the first few steps of proving the specification for the \$yield function from Figure 3. Recall from §2 we want this specification to have the form $\{I\}$ [invoke \$addryield] $\{I\}$. More precisely, given an invariant I, a fraction q and a frame F, we wish to prove that

$$I \twoheadrightarrow\; addryield \xmapsto{\mathsf{wf}} cl_{\mathsf{yield}} \;\twoheadrightarrow\; addrt \xmapsto{\mathsf{wtag}}_q ([] \to []) \twoheadrightarrow$$
$$\mathsf{wp}\;[\;\mathtt{invoke}\; addryield\;]\,; F \left\{ w\; F', \begin{array}{l} w = \mathsf{immV}\;[] * F = F' * I * \\ addryield \xmapsto{\mathsf{wf}} cl_{\mathsf{yield}} * addrt \xmapsto{\mathsf{wtag}}_q ([] \to []) \end{array} \right\}$$

where $addryield$ is F.inst.funcs[\$yield] (the address of function \$yield in the store), $addrt$ is F.inst.tags[\$t] (the address of tag \$t in the store) and cl_{yield} is the closure of yield function (containing the function's actual code, the instance of the coroutine module, the function's type and the type of its local variables). The function closure resource $addryield \xmapsto{\mathsf{wf}} cl_{\mathsf{yield}}$ is necessary to reason about the act of invoking, and is given back in the post-condition; the tag resource $addrt \xmapsto{\mathsf{wtag}}_q ([] \to [])$ is necessary to run the actual code of the function (recall it contains a

[1]These technical reasons pertain to the soundness of the proof rule for binding into `frame` instructions. Since reducing the body of a local is done using a different frame, the authoritative view associated to the state must be updated. This can only be done in the presence of the associated fragmental view, that is in this case, the frame resource. To do this, the strategy implemented by Iris-Wasm was to bake the presence of the frame resource into the definition of the weakest precondition by enforcing that it must be present every time a step in the execution is taken. As we describe in this paragraph, in Iris-WasmFX, we us a different, simpler strategy.

suspend instruction; we showcase in §4.3 the part where this resource is used) and is also given back in the post-condition.

To prove this specification, we start by applying Iris-Wasm rule WP-INVOKE-NATIVE which mirrors the operational semantics of `invoke` (see §3.2) and has two premises: the first requires us to provide a resource corresponding to the function closure for $yield, which we hold. The second gives back that resource, and requires us to prove a weakest precondition statement on the expression the `invoke` reduces to:

$$\mathsf{wp}\ [\ \mathsf{frame}_0\{F'\}\ \mathsf{block}\ ([] \to [])\, code_{\text{yield}}\ \mathsf{end}\ ]\ ; F \left\{ w\ F',\ \begin{array}{l} w = \mathsf{immV}\ [] * F = F' * \\ I * addryield \overset{\mathsf{wf}}{\longmapsto} cl_{\text{yield}} \end{array} \right\}$$

where F' is the frame under which the code $code_{\text{yield}}$ is to be executed, containing the coroutine module's instance and no local variables.

Now we apply rule WP-BIND-FRAME above, which brings us to proving

$$\mathsf{wp}\ [\mathsf{block}\ ([] \to [])\, code_{\text{yield}}]\ ; F'\ \{w\ F'', \Phi(w, F'')\}$$

with

$$\Phi(w, F'') = \mathsf{wp}\ [\ \mathsf{frame}_0\{F''\}\ w\ \mathsf{end}\ ]\ ; F \left\{ w\ F',\ \begin{array}{l} w = \mathsf{immV}\ [] * F = F' * \\ I * addryield \overset{\mathsf{wf}}{\longmapsto} cl_{\text{yield}} \end{array} \right\}$$

i.e. we have focused on the inside of the `frame` instruction and are now reasoning under frame F', the one with the closure's environment; once we will reach a value, we can reason about this value placed back into the `frame` instruction under the top-level F frame. This is where our treatment of the Wasm frame differs from Iris-Wasm, where frame resources $\overset{\text{FR}}{\hookrightarrow} F$ would be passed around in a sometimes convoluted way, especially when applying the WP-BIND-FRAME rule.

Then we apply Iris-Wasm rule WP-BLOCK, bringing us to reasoning about $[\mathsf{label}_0\{[]\}\ \ code_{\text{yield}}\ \ \mathsf{end}]$ instead of the `block` instruction; and finally, applying rule WP-BIND-LABEL from §4.1, our goal becomes

$$\mathsf{wp}\ \mathbf{code}_{\text{yield}}\ ; F'\ \{w\ F'', \mathsf{wp}\ \mathsf{label}_0\{[]\}\ w\ \mathsf{end}\ ; F''\ \{w\ F'', \Phi(w, F'')\}\}$$

i.e. we are now entirely focused on the code of $yield. As displayed in Figure 3, that code is just one line: [suspend $t]. We finish the proof in §4.3 and in Appendix E.1.1.

Other Approaches. Adjoint separation logic [Wagner et al. 2025] elegantly deals with this problem with frames by using modalities that make a given frame available or lock it away.

C Syntactic Typing and Type Safety in Detail

In this section, we introduce WasmFX's syntactic type system. When defining WasmFXCert, we give new typing rules for administrative instructions, and state and prove a type safety theorem.

Wasm defines a simple syntactic type system for all its instructions, of the form

$$C \vdash b \colon ts_1 \to ts_2$$

where C is a typing context (see Figure 8) that keeps track of the type of all the objects reachable in the current module, including all the labels that can be branched to and the return type of the current function; b is the basic instruction being typechecked; ts_1 is the list of the types of the values expected to be on the stack for the instruction to execute safely; and ts_2 is the list of the types of the values deposited back on the stack after execution. For example, for all contexts C and all numerical types nt, $C \vdash nt.\mathbf{add} \colon [nt; nt] \to [nt]$.

We show the typing rules for the new instructions of WasmFX in Figure 9.

$$\boxed{C \vdash cc \ : \ ts}$$

$$\frac{C.\mathsf{tags}[i] = ts_1 \to ts_2 \qquad C.\mathsf{label}[i'] = ts_1 \mathbin{+\!\!+} (ts_2 \to ts).\mathsf{cont}}{C \vdash \mathsf{on}\ i\ i' \ : \ ts} \qquad \frac{C.\mathsf{tags}[i] = [] \to ts}{C \vdash \mathsf{on}\ i\ \mathsf{switch} \ : \ ts}$$

$$\boxed{C \vdash b \colon ft}$$

$$\frac{C.\mathsf{types}[i] = ft}{C \vdash \mathsf{cont.new}\ i \colon ft.\mathsf{func} \to ft.\mathsf{cont}} \qquad \frac{C.\mathsf{tags}[i] = ft}{C \vdash \mathsf{suspend}\ i \colon ft}$$

$$\frac{C.\mathsf{types}[i] = ft \qquad C.\mathsf{tags}[i'] = [] \to ts \qquad ft = (ts_1 \mathbin{+\!\!+} (ts_2 \to ts).\mathsf{cont}) \to ts}{C \vdash \mathsf{switch}\ i\ i' \colon (ts_1 \mathbin{+\!\!+} ft.\mathsf{cont}) \to ts_2}$$

$$\frac{C.\mathsf{types}[i] = ft \qquad ft = ts_1 \to ts_2 \qquad \forall cc \in ccs,\ C \vdash cc \ : \ ts_2}{C \vdash \mathsf{resume}\ i\ ccs \colon (ts_1 \mathbin{+\!\!+} ft.\mathsf{cont}) \to ts_2}$$

Fig. 9. The typing rules for WasmFX

When defining WasmFXCert, we have included the first result on type safety of WasmFX, in the form of a theorem proved in the Rocq proof assistant. To formulate this theorem, we had to provide the first formal definitions of typing rules for the new administrative instructions and for the new parts of the Wasm store. While Phipps-Costin et al. [2023] define a typing rule for `prompt`, we have identified that their presentation is incomplete: it has a typo, and their strategy of stripping the context is too weak to actually complete the proof of type preservation. Our typing rules are presented in Figure 10.

Two main challenges arise when defining these typing rules. The first is that, unlike in plain Wasm 1.0, not all values automatically typecheck in all typing contexts, since function references and continuation references require the presence of a well-typed object in the typing context. This means that one needs to add typechecking hypotheses to many lemmas, and carefully keep track in proofs of which values are known to typecheck. Moreover, it also means that the store cannot be arbitrarily changed while ensuring all Wasm values keep on being well-typed (in 1.0, values always have a numerical type so they typecheck trivially as long as of the correct integer type). Hence one needs to fine-tune the definition of the store_extension(S, S') predicate, which describes the way in which the store can evolve during execution and which satisfies that values that typecheck in S still typecheck in S'.

$\boxed{S, C \vdash dcc \ : \ ts}$

$$\frac{S.\mathrm{tags}[addr] = ts_1 \to ts_2 \qquad C.\mathrm{label}[i] = ts_1 +\!\!+ \ (ts_2 \to ts).\mathrm{cont}}{S, C \vdash \mathtt{on}\ addr\ i \ : \ ts}$$

$$\frac{S.\mathrm{tags}[addr] = [\,] \to ts}{S, C \vdash \mathtt{on}\ addr\ \mathtt{switch} \ : \ ts}$$

$\boxed{S, C \vdash es : ft}$

$$\frac{S.\mathrm{conts}[\mathbf{addr}] = (\mathsf{cont}\ ft\ \_ \lor \mathsf{dagger}\ ft)}{S, C \vdash \mathtt{ref.cont}\ addr : [\,] \to ft.\mathrm{cont}} \qquad \frac{S.\mathrm{tags}[addr] = ts_1 \to ts_2 \qquad S, C \vdash vs : [\,] \to ts_1}{S, C \vdash \mathtt{suspend.addr}\ vs\ addr : [\,] \to ts_2}$$

$$\frac{\begin{array}{c} S.\mathrm{tags}[addr'] = [\,] \to ts \qquad ft = (ts_1 +\!\!+ \ (ts_2 \to ts).\mathrm{cont}) \to ts \\ S.\mathrm{conts}[\mathbf{addr}] = (\mathsf{cont}\ ft\ \_ \lor \mathsf{dagger}\ ft) \qquad S, C \vdash vs : [\,] \to ts_1 \end{array}}{S, C \vdash \mathtt{switch.addr}\ vs\ addr\ ft\ addr' : [\,] \to ts_2}$$

$$\frac{S, \emptyset \vdash es : [\,] \to ts \qquad \forall dcc \in dccs,\ S, C \vdash dcc \ : \ ts}{S, C \vdash \mathtt{prompt}_{ts}\{dccs\}\ es\ \mathtt{end} : [\,] \to ts}$$

$\boxed{S \vdash cinst\ \mathsf{ok}}$

$$\frac{}{S \vdash \mathsf{dagger}\ ft\ \mathsf{ok}} \qquad \frac{S, \emptyset \vdash vs : [\,] \to ts_1 \qquad H[vs] = es \qquad S, \emptyset \vdash es : [\,] \to ts_2}{S \vdash \mathsf{cont}\ (ts_1 \to ts_2)\ H\ \mathsf{ok}}$$

$\boxed{S \vdash einst\ \mathsf{ok}}$

$$\frac{S, \emptyset \vdash e.\mathrm{efields} : [\,] \to ts \qquad S.\mathrm{tags}[e.\mathrm{etag}] = ts \to [\,]}{S \vdash e\ \mathsf{ok}}$$

$\boxed{\vdash S\ \mathsf{ok}}$

$$\frac{\text{Premises for} \begin{array}{c}\text{functions}\\ \text{tables}\\ \text{memories}\\ \text{globals}\end{array} \text{as in WasmCert} \qquad \forall c \in S.\mathrm{conts},\ S \vdash c\ \mathsf{ok} \qquad \forall e \in S.\mathrm{exns},\ S \vdash e\ \mathsf{ok}}{\vdash S\ \mathsf{ok}}$$

Fig. 10. Our new typing rules for the administrative instructions and store elements of WasmFX

The second difficulty is the choice of a typing context for the body of a `prompt` instruction. This choice is ultimately irrelevant since in practice, all continuations start as function calls, and hence after the first few steps of execution, every continuation will be of the form [$\mathtt{frame}_n\{F\}\ H$ `end`], and the body of a `frame` instruction is typed with a specifically built typing context in the Wasm typing rules. However, the presence of this `frame` instruction immediately under every `prompt` is not enforced syntactically, and is not true for the first few steps of execution when a new continuation is created using the `cont.new` instruction. Our solution is to mandate that the body of a `prompt` should typecheck in the empty typing context, as illustrated in Figure 10. This necessitated replacing all tag and type annotations with their explicit values (as is done implicitly in WasmFX during the typing phase), and proving lemmas about typing expressions in the empty typing context, such as the following:

Lemma C.1. *For all stores S, typing contexts C, administrative instructions es, and function types ft,*

$$S, \varnothing \vdash es \colon ft \implies S, C \vdash es \colon ft$$

Type Safety Theorem. As is standard in Wasm's typing system, we can define a context $C(S, F)$ out of a store S and a frame F by setting $C.\mathsf{locs}$ to be the types of the values $F.\mathsf{locs}$ and filling all other fields of C by reading the type annotations in the store of the elements at the addresses specified by instance $F.\mathsf{inst}$.

We can now formulate the progress and preservation lemmas:

Lemma C.2 (Type Preservation). *If* $\vdash S$ ok *and* $S, C(S, F) \vdash es \colon [] \to ts$ *and* $(S, F, es) \hookrightarrow (S', F', es')$*, then* $\vdash S'$ ok *and* $S', C(S', F') \vdash es' \colon [] \to ts$

Lemma C.3 (Type Progress). *If* $\vdash S$ ok *and* $S, C(S, F) \vdash es \colon [] \to ts$*, then either es reduces, or it is a constant value, or it is stuck on a host call, or it is stuck on an unhandled suspend, switch, or exception throw.*

And together, these give the type safety result:

Theorem C.4 (Type Safety). *If* $\vdash S$ ok *and* $S, C(S, F) \vdash es \colon [] \to ts$ *and* $(S, F, es) \hookrightarrow^{\star} (S', F', es')$*, then either es' reduces, or it is a constant value, or it is stuck on a host call, or it is stuck on an unhandled suspend, switch, or exception throw.*

D Bind Rule for Prompt

Since the `resume` instruction reduces to a `prompt` instruction, the crux of the reasoning is how to use protocols to bind the body of a `prompt` instruction. As discussed, the simple EWP-LABEL rule would be unsound due to the presence of effects. Instead, we prove the following rule:

EWP-PROMPT

$$\frac{\begin{array}{ll} (1) & (\forall addr, addr \notin dccs \implies \Psi(addr) = \Psi'(addr)) \ast \\ (2) & \mathrm{CExp}(es) \ast (\forall F, \neg\Phi(\mathtt{trapV}, F)) \ast \neg\Phi'(\mathtt{trapV}) \ast \\ (3) & \mathsf{ewp}\ es\,; \varnothing\ \langle\Psi\rangle\ \{w\ F, \Phi(w, F)\} \ast \\ (4) & (\forall w, \Phi(w, \varnothing) \twoheadrightarrow \mathsf{ewp}\ [\mathtt{prompt}_{ts}\{dccs\}\ w\ \mathtt{end}]\,; \varnothing\ \langle\Psi'\rangle\ \{w\ \_, \Phi'(w)\}) \ast \\ (5) & \forall dcc \in dccs, \langle\Psi\rangle\ \{\Phi\}\ dcc;\ ts_2\ \langle\Psi'\rangle\ \{\lambda v\ \_, \Phi'(v)\} \end{array}}{\mathsf{ewp}\ [\mathtt{prompt}_{ts}\{dccs\}\ es\ \mathtt{end}]\,; F\ \langle\Psi'\rangle\ \{w\ F', \Phi'(w) \ast F = F'\}}$$

This rule has the structure of a usual bind rule: line (3) focuses on the body *es* of the `prompt` instruction, and line (4) injects the resulting value *w* into the `prompt`. It allows using a different protocol family for the body *es* than for the `prompt` instruction itself, to account for the fact that the `prompt` specifies the intended behaviour for all the effects handled in its clauses *dccs*. Line (1) enforces that the protocols families Ψ and Ψ' only differ in the effects handled by the `prompt` clauses; line (5) ties them together in those remaining places, as we will explain below.

Let us now focus on line (2): aside from enforcing that all postconditions do not hold of the trap value `trapV` (a different, simpler rule can be invoked for expressions that trap), it requires the body *es* of the `prompt` to satisfy predicate CExp, which we define as follows:

$$\begin{array}{rcll} \mathrm{CExp}(es) & \triangleq & (\exists vs\ addr\ ft,\ es = vs \mathbin{+\!\!+} [\mathtt{ref.func}\ addr;\ \mathtt{ref.call}\ ft]) \lor & (a) \\ & & (\exists vs\ addr,\ es = vs \mathbin{+\!\!+} [\mathtt{invoke}\ addr]) \lor & (b) \\ & & (\exists vs\ n\ F\ es',\ es = vs \mathbin{+\!\!+} [\mathtt{frame}_n\{F\}\ es'\ \mathtt{end}]) \lor & (c) \\ & & (\exists vs,\ es = vs \mathbin{+\!\!+} [\mathtt{trap}]) \lor & (d) \\ & & (\exists vs,\ es = vs) \lor & (e) \\ & & (\exists vs\ vs'\ k\ ft,\ es = vs \mathbin{+\!\!+} [\mathtt{call\_host}\ vs'\ k\ ft] & (f) \end{array}$$

i.e. *es* is of one of 6 forms: (a) values followed by a call to a function reference, (b) values followed by an `invoke` instruction (i.e. the intermediate representation before a call instruction is replaced by the function's actual body), (c) values followed by a `frame` instruction (representing a function body being executed), (d) values followed by a `trap` instruction, (e) *es* is constant, or (f) values followed by a `call_host` instruction (a stuck instruction representing a call to a function defined by the host language).

As previously mentioned, our continuation resource enforces all continuations to be of a special form, which means that when plugged with a value, the resulting expression is always of form (a) or (c). The other four cases have been added to guarantee closure under reduction: if $\mathrm{CExp}(es)$ and *es* reduces to es', then $\mathrm{CExp}(es')$. The most crucial property verified by expressions of these forms is that they reduce in a frame-agnostic way: if $\mathrm{CExp}(es)$ and (S, F, es) reduces to (S', F', es'), then $F = F'$ and for any other frame F'', (S, F'', es) reduces to (S', F'', es').

Line (5) of rule EWP-PROMPT specifies the behaviour of the program when an effect handled by one of the clauses of the `prompt` is performed using the `suspend` instruction. In that case, recall that reduction rule REDUCE-SUSPEND mandates that a continuation is created out of the state of the stack, and the `prompt` reduces to a `br` instruction with the label specified by the clause from the `prompt` instruction, with the effect's payload and the newly created continuation on the stack. Hence we wish to require that in that case, we hold an extended weakest precondition statement for this $vs \mathbin{+\!\!+} [\mathtt{ref.cont}\ kaddr; \mathtt{br}\ ilab]$ expression, where *kaddr* is the address of the newly created continuation. Little is known about the form of that continuation, other than that the `suspend` instruction will have followed a protocol from family Ψ. We define the *clause triple* from line (5) (which closely follows that of Hazel [de Vilhena and Pottier 2021, Fig. 7]), as:

$$\begin{array}{l}
\langle\Psi\rangle\,\{\Phi\}\ \mathsf{on}\ \mathit{taddr}\ \mathit{ilab};\ \mathit{ts}\,\langle\Psi'\rangle\,\{\Phi'\} = \\
\quad \exists \mathit{ts}_1\ \mathit{ts}_2\ q,\ \mathit{taddr} \xmapsto{\mathsf{wtag}}_q (\mathit{ts}_1 \longrightarrow \mathit{ts}_2) \ast \\
\quad \forall \mathit{vs}\ \mathit{kaddr}\ H,\ \mathit{kaddr} \xmapsto{\mathsf{wcont}} (\mathit{ts}_2 \longrightarrow \mathit{ts}, H) \mathrel{-\!\!\ast} \mathit{taddr} \xmapsto{\mathsf{wtag}}_q (\mathit{ts}_1 \longrightarrow \mathit{ts}_2) \mathrel{-\!\!\ast} \\
\qquad \uparrow (\Psi\ \mathit{taddr})\ (\mathtt{immV}\ \mathit{vs})\ (\lambda w, \triangleright\, \mathsf{ewp}\, H[w]\,;\varnothing\,\langle\Psi\rangle\,\{w\ F, \Phi(w,F)\}) \mathrel{-\!\!\ast} \\
\qquad\quad \mathsf{ewp}\ \mathit{vs} \mathbin{+\!\!+} [\mathtt{ref.cont}\ \mathit{kaddr}; \mathtt{br}\ \mathit{ilab}]\,;\varnothing\,\langle\Psi'\rangle\,\{w\ F, \Phi'(w,F)\}
\end{array}$$

When establishing this premise, we must prove the extended weakest precondition statement on the `br` instruction using these resources: a continuation resource corresponding to the new continuation, and knowledge that the effect was performed in a way that follows the protocol $\Psi\ \mathit{taddr}$. Since the continuation H is universally quantified, the only way to reason about resuming the continuation again is by knowing that the protocol is followed.

For example, when $\Psi\ \mathit{taddr}$ is of the form $!x(v)\{P\}\ ?y(w)\{Q\}$, this definition simplifies to

$$\begin{array}{l}
\exists \mathit{ts}_1\ \mathit{ts}_2\ q,\ \mathit{taddr} \xmapsto{\mathsf{wtag}}_q (\mathit{ts}_1 \longrightarrow \mathit{ts}_2) \ast \\
\forall \mathit{kaddr}\ H\ x,\ \mathit{kaddr} \xmapsto{\mathsf{wcont}} (\mathit{ts}_2 \longrightarrow \mathit{ts}, H) \mathrel{-\!\!\ast} \mathit{taddr} \xmapsto{\mathsf{wtag}}_q (\mathit{ts}_1 \longrightarrow \mathit{ts}_2) \mathrel{-\!\!\ast} \\
\quad P \mathrel{-\!\!\ast} (\forall y, Q \mathrel{-\!\!\ast} \triangleright\, \mathsf{ewp}\, H[w]\,;\varnothing\,\langle\Psi\rangle\,\{w\ F, \Phi(w,F)\}) \mathrel{-\!\!\ast} \\
\qquad \mathsf{ewp}\, v \mathbin{+\!\!+} [\mathtt{ref.cont}\ \mathit{kaddr}; \mathtt{br}\ \mathit{ilab}]\,;\varnothing\,\langle\Psi'\rangle\,\{w\ F, \Phi'(w,F)\}
\end{array}$$

i.e. we must show that breaking with value v is safe, knowing that P holds, and that we can safely resume the continuation with w as long as we can provide Q.

Frame Agnosticity. The last two lines of the definition of the clause triple refer to an empty frame. In theory, according to the operational semantics, the frame to be used in this position is the frame as it would be at the suspension site (for the last line) and re-resumption site (for the next-to-last line), but when scrutinising a `prompt` instruction, the state of the frame at those later times is unknown. Fortunately, the choice of the frame to use at these points is in practice irrelevant since the inside of the `prompt` instruction reduces in a frame-agnostic way, hence we can choose whatever frame we want. We find it convenient to use the empty frame everywhere (including line (3) in rule EWP-PROMPT), so that all the frames inside the `prompt` instruction match one another, simplifying reasoning.

To bridge the frame F in the conclusion of EWP-PROMPT with the empty frame in premise (3), we prove the following lemma, which says that if an expression es satisfying $\mathrm{CExp}(\mathit{es})$ is safe to run under the empty frame, then the same expression is safe to run with any frame, resulting in an unchanged frame, with the same post-condition on the resulting value:

$$\textsc{ewp-empty-frame}\quad \frac{\mathrm{CExp}(\mathit{es}) \ \ast\ \mathsf{ewp}\ \mathit{es}\,;\varnothing\,\langle\Psi\rangle\,\{w\ \_, \Phi(w)\}}{\mathsf{ewp}\ \mathit{es}\,;F\,\langle\Psi\rangle\,\{w\ F', \Phi(w)\ \ast\ F = F'\}}$$

Resume Rule. The rule for `resume` is largely the same as for EWP-PROMPT:

EWP-RESUME

$$\frac{\begin{array}{ll}(1) & F.\mathsf{inst.types}[i] = ts_1 \to ts_2 \ * \ |vs| = |ts_1| \ * \ \mathsf{translate\_clauses}(F.\mathsf{inst}, ccs) = dccs \ * \\ (2) & (\forall addr, addr \notin dccs \implies \Psi(addr) = \Psi'(addr)) \ * \ addr \xmapsto{\mathsf{wcont}} (ts_1 \to ts_2, H) \ * \\ (3) & (\forall F, \neg\Phi(\mathsf{trapV}, F)) \ * \ \neg\Phi'(\mathsf{trapV}) \ * \ \triangleright \mathsf{ewp}\, H[vs]\,;\varnothing\,\langle\Psi\rangle\,\{w\ F, \Phi(w, F)\} \ * \\ (4) & \triangleright(\forall w, \Phi(w, \varnothing) \mathrel{-\!\!*} \mathsf{ewp}\ [\mathsf{prompt}_{ts_2}\{dccs\}\ w\ \mathsf{end}]\,;\varnothing\,\langle\Psi'\rangle\,\{v\ \_, \Phi'(v)\}) \ * \\ (5) & \triangleright(\forall dcc \in dccs, \langle\Psi\rangle\,\{\Phi\}\ dcc;\ ts_2\,\langle\Psi'\rangle\,\{\lambda v\ \_, \Phi'(v)\})\end{array}}{\mathsf{ewp}\ vs \mathbin{+\!\!+} [\mathsf{ref.cont}\ addr; \mathsf{resume}\ i\ ccs]\,; F\,\langle\Psi'\rangle\,\{v\ F', \Phi'(v) \ * \ F' = F\}}$$

The additional premises are the continuation resource on line (2), and the premises on line (1) which translate the annotations of the `resume` instruction and check that the adequate number of values are being taken from the stack. The extra premises are necessary to apply reduction rule REDUCE-RESUME, which transforms the `resume` into a `prompt` instruction. Since a step is taken, we can add a later modality $\triangleright$ to lines (4) and (5). The rest of the premises are those of EWP-PROMPT, allowing to bind into the actual code of the continuation being resumed. Because the continuation resource enforces that the context H is safe, the rule does not require establishing $\mathrm{CExp}(H[vs])$.

$$
\begin{aligned}
&\exists \mathit{addryield}\ \mathit{addrpar}\ \mathit{cl}_{\text{yield}}\ \mathit{cl}_{\text{par}},\\
&J_{\text{yield}} \ast J_{\text{par}} \ast \qquad \text{(we own the closures for \$par and \$yield, \ldots)}\\
&\forall P_1\ Q_1\ P_2\ Q_2\ I, \exists \Psi,\\
&\text{(\ldots and we have specifications for both functions)}\\
&\text{(for function \$yield:)}\\
&\Box\left(\begin{array}{l}\forall F, I \twoheadrightarrow J_{\text{yield}} \twoheadrightarrow\\ \mathsf{ewp}\ [\mathtt{invoke}\ \mathit{addryield}]\ ; F\ \langle\Psi\rangle \left\{w\ F',\ \begin{array}{l} w = \mathtt{immV}\ [] \ast F = F' \ast\\ I \ast J_{\text{yield}}\end{array}\right\}\end{array}\right) \ast\\
&\text{(and for function \$par:)}\\
&\forall \mathit{addrf_1}\ \mathit{addrf_2}\ \mathit{cl_1}\ \mathit{cl_2},\\
&\Box\left(\begin{array}{l}
\forall F, P_1 \twoheadrightarrow P_2 \twoheadrightarrow I \twoheadrightarrow J_1 \twoheadrightarrow J_2 \twoheadrightarrow J_{\text{yield}} \twoheadrightarrow J_{\text{par}} \twoheadrightarrow\\
\text{(precondition of specification for \$par contains a specification for \$f}_1\text{:)}\\
\Box\left(\begin{array}{l}\forall F', P_1 \twoheadrightarrow I \twoheadrightarrow J_1 \twoheadrightarrow J_{\text{yield}} \twoheadrightarrow\\ \mathsf{ewp}\left[\begin{array}{l}\mathtt{ref.func}\ \mathit{addrf_1};\\ \mathtt{ref.call}\end{array}\right]; F'\ \langle\Psi\rangle \left\{w\ F'',\ \begin{array}{l} w = \mathtt{immV}\ [] \ast F' = F'' \ast\\ Q_1 \ast I \ast J_1 \ast J_{\text{yield}}\end{array}\right\}\end{array}\right) \twoheadrightarrow\\
\text{(precondition of specification for \$par contains a specification for \$f}_2\text{:)}\\
\Box\left(\begin{array}{l}\forall F', P_2 \twoheadrightarrow I \twoheadrightarrow J_2 \twoheadrightarrow J_{\text{yield}} \twoheadrightarrow\\ \mathsf{ewp}\left[\begin{array}{l}\mathtt{ref.func}\ \mathit{addrf_2};\\ \mathtt{ref.call}\end{array}\right]; F'\ \langle\Psi\rangle \left\{w\ F'',\ \begin{array}{l} w = \mathtt{immV}\ [] \ast F' = F'' \ast\\ Q_2 \ast I \ast J_2 \ast J_{\text{yield}}\end{array}\right\}\end{array}\right) \twoheadrightarrow\\
\mathsf{ewp}\left[\begin{array}{l}\mathtt{ref.func}\ \mathit{addrf_1};\\ \mathtt{ref.func}\ \mathit{addrf_2};\\ \mathtt{invoke}\ \mathit{addrpar}\end{array}\right]; F\ \langle\bot\rangle \left\{w\ F',\ \begin{array}{l} w = \mathtt{immV}\ [] \ast F = F' \ast Q_1 \ast Q_2 \ast\\ I \ast J_1 \ast J_2 \ast J_{\text{yield}} \ast J_{\text{par}}\end{array}\right\}
\end{array}\right)
\end{aligned}
$$

Where J_{yield} is a shorthand for $\mathit{addryield} \xmapsto{\mathsf{wf}} \mathit{cl}_{\text{yield}}$, J_{par} is a shorthand for $\mathit{addrpar} \xmapsto{\mathsf{wf}} \mathit{cl}_{\text{par}}$, J_1 is a shorthand for $\mathit{addrf_1} \xmapsto{\mathsf{wf}} \mathit{cl_1}$, and J_2 is a shorthand for $\mathit{addrf_2} \xmapsto{\mathsf{wf}} \mathit{cl_2}$

Fig. 11. The specification for the coroutines module

E Case Studies in Detail

E.1 Coroutine Library

We now revisit our running coroutine example from §2, focusing on proving the implementation's specification (§E.1.1) and the simple client we introduced in Figure 13 (§E.1.2).

A reader familiar with modern separation logics might notice that one could give a more implicit specification, without this explicit invariant I, using for example Iris invariants [Jung et al. 2018, §2.2]. We present the specification in CSL style for clarity.

E.1.1 Library implementation. The code for the coroutine library module is shown in Figure 3. We presented an informal version of its specification in §2. Now that we have introduced the full machinery of Iris-WasmFX, we show the full specification in Figure 11.

Compared to the snapshot given in §2, there are two main differences:

(1) We have inlined the definition of the Hoare triple and replaced the weakest precondition statement with our extended weakest precondition
(2) We have added all relevant function closure resources in the pre- and post-conditions

In essence, the meaning remains the same: in order to \$yield, one must provide the invariant I and that invariant is restituted when the function returns; and in order to run \$par on two functions \$f$_1$ and \$f$_2$, one must have established specifications for both functions.

$$
\begin{array}{rcl}
LI & = & \exists H_a, \\
& & \left(\begin{array}{l}
I * J * \$\texttt{current} \xmapsto{\texttt{wcont}} H_a * \\
\left(I \mathrel{-\!\!*} J \mathrel{-\!\!*} \triangleright \mathsf{ewp}\, H_a\, [()] \,;\, \varnothing \,\langle \Psi_0 \rangle \left\{ w\ F,\ \begin{array}{l} w = \texttt{immV}\ [] * Q_1 * \\ I * \$\texttt{f}_1 \xmapsto{\texttt{wf}} cl_1 * J \end{array} \right\}\right) * \\
\left(\begin{array}{l}
\$\texttt{next\_is\_done} = 0 * \exists H_b, \$\texttt{next} \xmapsto{\texttt{wcont}} H_b * \\
\left(I \mathrel{-\!\!*} J \mathrel{-\!\!*} \triangleright \mathsf{ewp}\, H_b\, [()] \,;\, \varnothing \,\langle \Psi_0 \rangle \left\{ w\ F,\ \begin{array}{l} w = \texttt{immV}\ [] * Q_2 * \\ I * \$\texttt{f}_2 \xmapsto{\texttt{wf}} cl_2 * J \end{array} \right\}\right) \\
\lor \quad (\$\texttt{next\_is\_done} = 1 * Q_2 * \$\texttt{f}_2 \xmapsto{\texttt{wf}} cl_2)
\end{array}\right)
\end{array}\right) \\
& \lor & \left(\begin{array}{l}
I * J * \$\texttt{current} \xmapsto{\texttt{wcont}} H_a * \\
\left(I \mathrel{-\!\!*} J \mathrel{-\!\!*} \triangleright \mathsf{ewp}\, H_a\, [()] \,;\, \varnothing \,\langle \Psi_0 \rangle \left\{ w\ F,\ \begin{array}{l} w = \texttt{immV}\ [] * Q_2 * \\ I * \$\texttt{f}_2 \xmapsto{\texttt{wf}} cl_2 * J \end{array} \right\}\right) * \\
\left(\begin{array}{l}
\$\texttt{next\_is\_done} = 0 * \exists H_b, \$\texttt{next} \xmapsto{\texttt{wcont}} H_b * \\
\left(I \mathrel{-\!\!*} J \mathrel{-\!\!*} \triangleright \mathsf{ewp}\, H_b\, [()] \,;\, \varnothing \,\langle \Psi_0 \rangle \left\{ w\ F,\ \begin{array}{l} w = \texttt{immV}\ [] * Q_1 * \\ I * \$\texttt{f}_1 \xmapsto{\texttt{wf}} cl_1 * J \end{array} \right\}\right) \\
\lor \quad (\$\texttt{next\_is\_done} = 1 * Q_1 * \$\texttt{f}_1 \xmapsto{\texttt{wf}} cl_1)
\end{array}\right)
\end{array}\right)
\end{array}
$$

$$\text{where } J \text{ is } \mathit{addryield} \xmapsto{\texttt{wf}} cl_{\text{yield}}.$$

Fig. 12. Loop invariant for proving the specification of the $par function

Note the quantification: the invariant I, as well as the pre- and post-conditions P_1, Q_1, P_2 and Q_2 of $\$f_1$ and $\$f_2$, can be chosen later by the client. This makes the specification more general and useful for more clients. However, the actual addresses and the actual closures of the functions $yield and $par, as well as the metaprotocol $\mathbf{\Psi}$ used by $yield, are hidden behind existentials, meaning that a client has no access to the code itself. When the client wishes to reason about invocation of one of these library functions, all they can do is apply this specification. By hiding the metaprotocol $\mathbf{\Psi}$ behind an existential quantifier, we can enforce that the only way for the client to use effects in a way that can be specified is by making a call to $yield.

Proving the Specification. To prove this specification, it suffices to show that the coroutines module satisfies the existentials with the actual addresses and closures, and to define $\mathbf{\Psi}$ as the metaprotocol $\mathbf{\Psi}_0$ described in §4.2. The function resources for $yield and $par are provided by the host language's program logic (see §3.5) during the instantiation process.

Then it remains to prove the two specifications; we gave an outline of the proof for $yield in Appendix B and in §4.3. Note that the resource for the tag $t does not feature in the specification. This is because this resource is not useful for the client and thus is not shown in the specification. When proving the specification, however, the resource is provided by the instantiation process together with the function closure resources.

For $par, the proof is slightly longer and requires establishing a loop invariant for the loop, and applying the EWP-RESUME rule. We show the loop invariant LI in Figure 12.

The invariant is a disjunction where the left case corresponds to the case where local variable $current is the continuation obtained originally from function $\$f_1$ and $next is the continuation obtained originally from function $\$f_2$; and the right disjunct is vice-versa, $current is $\$f_2$ and $next is $\$f_1$. In both cases, we assert that we own the invariant I, the function closure resource J, and a continuation H_a at the address specified by local variable $current, and we require that:

- it is sufficient to give I and J to be able to safely run $H_a[()]$, and
- one of the following is true:

- the variable \$next_is_done is **false**, which means that the continuation in variable \$next is not done executing: we require ownership of a continuation H_b at the address specified by \$next, and holding I and J must be enough to safely run $H_b[()]$; or
- the variable \$next_is_done is **true**, which means that the continuation in variable \$next has terminated: we must hold the postcondition of the corresponding function.

Let us go through the code in Figure 3 line by line.

Lines 13 to 20 create the continuations; using rule EWP-CONTNEW, we get the continuation resources corresponding to calling $\$f_1$ and $\$f_2$. To enter the loop on line 21, we must establish the loop invariant, LI (left disjunct, and left disjunct in the inner disjunct). To fulfil the extended weakest precondition premises in LI, we use the specifications for $\$f_1$ and $\$f_2$ since at this moment the continuations in \$current and \$next contain a call to $\$f_1$ and $\$f_2$.

Then we reason about the inside of the loop: first we resume continuation \$current on line 24. For this, we apply rule EWP-RESUME. Most premises are either trivial or given directly by the loop invariant LI (e.g. the continuation resource on line (2) and the extended weakest precondition on line (3)). The two missing premises are lines (4) and (5).

For line (4), we must establish the termination case. We assume the postcondition of the function that just terminated (i.e. $Q_i * I * \$f_i \xmapsto{\mathsf{wf}} cl_i$ for the correct $i \in \{1, 2\}$) and prove that we can safely run lines 25 to 31 (right after the `resume`). On lines 25-26, we check the value of \$next_is_done. If it is 1, we exit the function; in virtue of our loop invariant LI, we know that the other $Q_j * \$f_j \xmapsto{\mathsf{wf}} cl_j$ also holds and we can conclude the proof. If \$next_is_done is 0, then we update its value (lines 27-28), place \$next in \$current (lines 29-30), and repeat the loop (line 31). We can do this, since we can reestablish the loop invariant LI (by using the right disjunct in the inner disjunction).

For line (5), we must show that given a continuation resource $\$current \xmapsto{\mathsf{wcont}} H$ for some safe context H, given I (from the protocol $\Psi_0(\$t)$), and given that I is enough to be able to safely run $H[()]$, we must prove that it is safe to run lines 32 to 39 (the code to which we branch when the effect is performed). We save the suspended continuation into variable \$current on line 32, then check the value of \$next_is_done on lines 33-34. If it is true, then we know \$next has done executing and we loop without swapping \$next and \$current; looping is safe since we can trivially reestablish the loop invariant LI. If \$next_is_done is false, then we swap \$next and \$current (lines 35-28) and loop (line 39), which is safe to do since we can reestablish the loop invariant LI using the left disjunct in the inner disjunction.

A full proof can be found in our Rocq development in file `coroutines_implementation_code.v`.

E.1.2 Client module. We illustrate usage of our coroutine module on a simple client whose code can be seen in Figure 13. For this client, we show the following specification:

$$addrmain \xmapsto{\mathsf{wf}} cl_{\mathsf{main}} \twoheadrightarrow$$
$$\mathsf{ewp}\ [\mathtt{invoke}\ addrmain]\ ; F\ \langle\bot\rangle \left\{ w,\ \begin{matrix} \exists vala\ valb, w = \mathtt{immV}\ [vala; valb] * \\ (vala = 1 \implies valb = 42) \end{matrix} \right\}$$

meaning that as long as we hold the corresponding closure resource, we can run the \$main function safely, and if it returns, it will return two integers such that if the first is 1, then the second is 42.

The proof of this specification is a classic concurrent separation logic example. We detail it here, as well as in our Rocq development, to illustrate that our logic is strong enough to capture reasoning principles for parallel composition for an implementation based on effects.

The proof relies on applying the specification for \$par. We show our choice of P_1, Q_1, P_2, Q_2 and I in Figure 14. Our invariant I is a three-case disjunction: either \$x and \$y are both still 0, or \$x has been updated but \$y not yet, or both variables have been updated. Importantly, it is not possible for \$y to be 1 if \$x is not 42. In order to keep track of what variable update has been done already, we

```
;; run with the specfx fork of the reference interpreter with
;; ./wasm coroutine.wat -e '(module instance)' -e '(register "coroutine")' client.wat -e '(
    module instance)' -e '(invoke "main")'
;; it should display
;; [0 42] : [i32 i32]
(module ;; client
  (type $func (func))
  (import "coroutine" "yield" (func $yield)) ;; needs yield and par
  (import "coroutine" "par" (func $par (param (ref $func)) (param (ref $func))))
  (global $x (mut i32) (i32.const 0))        ;; shared `data' $x
  (global $y (mut i32) (i32.const 0))        ;; shared `flag' $y
  (global $a (mut i32) (i32.const 0))        ;; reader's copy of the value of $y
  (global $b (mut i32) (i32.const 0))        ;; reader's copy of the value of $x
  (elem declare funcref (ref.func $f1))      ;; make $f1 available
  (elem declare funcref (ref.func $f2))      ;; and $f2
  (func $f1
    i32.const 42                             ;; write 42 to $x
    global.set $x
    call $yield                              ;; yield
    i32.const 1                              ;; set the flag
    global.set $y)
  (func $f2
    global.get $y                            ;; read from $y
    global.set $a                            ;; save in $a
    call $yield                              ;; yield
    global.get $x                            ;; read from $x
    global.set $b)                           ;; save in $b
  (func $main (result i32 i32)
    ref.func $f1                             ;; run $f1 and $f2 concurrently
    ref.func $f2
    call $par
    global.get $a                            ;; look up what the reader saw
    global.get $b)
  (export "main" (func $main)))              ;; make $main available outside
```

Fig. 13. A client module using our coroutine library to run message-passing. set and get respectively take their arguments and put their results on the stack.

use *ghost resources* that assist in reasoning but do not directly correspond to a piece of the Wasm state. We use the *one-shot algebra* [Jung et al. 2016], which has two possible states: Pending, and Shot. When allocating a new resource, we get ownership of the resource in the Pending state. We can then later decide to update Pending to Shot, symbolising that a change has taken place. Pending and Shot cannot be simultaneously owned. The resource in the Shot state is duplicable, meaning we can 'keep a record' of the fact that the change has taken place. For our example, we allocate two one-shots, which we identify by their *ghost names* γ_x and γ_y. Hence $\boxed{\text{Pending}}^{\gamma_x}$ means that $x has not been updated yet, and $\boxed{\text{Shot}}^{\gamma_x}$ means it has been updated; likewise for γ_y and $y. As displayed in Figure 14, we place a ghost resource next to each global variable resource in the definition of the invariant I, choosing Shot or Pending depending on the value of the global variable.

When establishing the pre-condition of the specification for $par, we must prove specifications for f_1$ and f_2$. For f_1$, the proof follows this scheme:

(1) $\{I\}$ Start with I (and vacuous P_1)
i32.const 42
`global.set` $x
(2) $\{I * \boxed{\text{Shot}}^{\gamma_x}\}$ Reestablish I but keep a copy of $\boxed{\text{Shot}}^{\gamma_x}$
`call` $yield
(3) $\{I * \boxed{\text{Shot}}^{\gamma_x}\}$ Using the specification for $yield
i32.const 1
`global.set` $y
(4) $\{I * \boxed{\text{Shot}}^{\gamma_x}\}$ Reestablish I

We start in (1) with the invariant I (and the pre-condition P_1, which in this case is vacuous). No matter which disjunct of I is true, it is possible to update $x to 42 and reestablish the invariant in (2), updating the ghost resource $\boxed{\text{Pending}}^{\gamma_x}$ to $\boxed{\text{Shot}}^{\gamma_x}$ if necessary. Since $\boxed{\text{Shot}}^{\gamma_x}$ is persistent, we can retain a copy of it when reestablishing the invariant I in (2). Next, we apply the specification for $yield, which consumes I and restores it upon return in (3). The resource $\boxed{\text{Shot}}^{\gamma_x}$ in the proof environment is unused and simply carried forward from (2) to (3) by framing. Now, because we own both I and $\boxed{\text{Shot}}^{\gamma_x}$, we know that we cannot be in the first disjunct of I, since it is not possible to simultaneously own $\boxed{\text{Pending}}^{\gamma_x}$ and $\boxed{\text{Shot}}^{\gamma_x}$. Thus, we are in one of the other two disjuncts, which allows us to update $y and reestablish the invariant in (4), updating the ghost resource $\boxed{\text{Pending}}^{\gamma_y}$ to $\boxed{\text{Shot}}^{\gamma_y}$ as necessary. This concludes the proof for f_1$.

For f_2$, the proof follows this scheme:

(1) $\{I * P_2\}$ Start with I and P_2
`global.get` $y
`global.set` $a If $y is 1, keep a copy of $\boxed{\text{Shot}}^{\gamma_y}$
(2) $\left\{I * addrb \xmapsto{\text{wg}} - * \left((addra \xmapsto{\text{wg}} 1 * \boxed{\text{Shot}}^{\gamma_y}) \vee addra \xmapsto{\text{wg}} 0\right)\right\}$
`call` $yield Using the specification for $yield
(3) $\left\{I * addrb \xmapsto{\text{wg}} - * \left((addra \xmapsto{\text{wg}} 1 * \boxed{\text{Shot}}^{\gamma_y}) \vee addra \xmapsto{\text{wg}} 0\right)\right\}$
`global.get` $x
`global.set` $b
(4) $\left\{I * \left((addra \xmapsto{\text{wg}} 1 * addrb \xmapsto{\text{wg}} 42) \vee (addra \xmapsto{\text{wg}} 0 * addrb \xmapsto{\text{wg}} -)\right)\right\}$

We start in (1) with the invariant I and precondition P_2. The first step is reading from $y to update $a. If we are in the third disjunct of I, then $a gets value 1, and we can keep a copy of $\boxed{\text{Shot}}^{\gamma_y}$ in (2) since that resource is duplicable. If we are in one of the other disjuncts, we know that the new value of $a is 0. Next, we apply the specification for $yield, which consumes I and gives it back upon return in (3). The other resources in our proof environment are unused by this step and simply carried forward from (2) to (3). Lastly, we read from $x to update $b. In the case where we own $addra \xmapsto{\text{wg}} 1$, we also own $\boxed{\text{Shot}}^{\gamma_y}$ so we cannot be in the first two disjuncts of I, as one cannot simultaneously own $\boxed{\text{Pending}}^{\gamma_y}$ and $\boxed{\text{Shot}}^{\gamma_y}$. Hence we know that the new value of $b is 42. In the other case, it does not matter in which disjunct of I we are; in all three cases the read and write are safe to execute, and we need not make a note of the new value of $b. The rest of the proof is simply showing that the proposition in (4) implies the post-condition Q_2, which is straightforward.

$$\begin{array}{rcl}
P_1 & = & \top \\
Q_1 & = & \boxed{\text{Shot}}^{\gamma_x} \\
P_2 & = & \exists \mathit{vala}\ \mathit{valb},\ \mathit{addra} \xmapsto{\text{wg}} \mathit{vala} \ast \mathit{addrb} \xmapsto{\text{wg}} \mathit{valb} \\
Q_2 & = & \exists \mathit{vala}\ \mathit{valb},\ \mathit{addra} \xmapsto{\text{wg}} \mathit{vala} \ast \mathit{addrb} \xmapsto{\text{wg}} \mathit{valb} \ast (\mathit{vala} = 1 \implies \mathit{valb} = 42) \\
I & = & (\mathit{addrx} \xmapsto{\text{wg}} 0 \ast \mathit{addry} \xmapsto{\text{wg}} 0 \ast \boxed{\text{Pending}}^{\gamma_x} \ast \boxed{\text{Pending}}^{\gamma_y}) \vee \\
 & & (\mathit{addrx} \xmapsto{\text{wg}} 42 \ast \mathit{addry} \xmapsto{\text{wg}} 0 \ast \boxed{\text{Shot}}^{\gamma_x} \ast \boxed{\text{Pending}}^{\gamma_y}) \vee \\
 & & (\mathit{addrx} \xmapsto{\text{wg}} 42 \ast \mathit{addry} \xmapsto{\text{wg}} 1 \ast \boxed{\text{Shot}}^{\gamma_x} \ast \boxed{\text{Shot}}^{\gamma_y})
\end{array}$$

Fig. 14. Resources for the specification of the client of our coroutine library.

Final Theorem. Using the Adequacy Theorem from §4.5 on the specification above, we can prove the following theorem, which makes no mention of Iris:

THEOREM E.1. *If Cor is the module whose code is shown in Figure 3, and Cl is the module whose code is shown in Figure 13, and if the host program*

$$[\texttt{instantiate}\ \mathit{Cor};\ \texttt{instantiate}\ \mathit{Cl};\ \texttt{invoke}\ \mathit{addrmain}]$$

terminates on a value w, then there exists two integers vala and valb such that $w = \texttt{immV}\ [\mathit{vala};\ \mathit{valb}]$*, and moreover, if vala is 1, then valb is 42.*

A full proof can be found in our Rocq development in file `coroutines_client.v`.

E.2 Generator

We define the protocol using a standard trick [Filliâtre and Pereira 2016; Pottier 2017]:

$$!x\ xs(x)\langle \text{Permitted}\ (x :: xs) \ast I\ xs\rangle\ ?()\langle I\ (x :: xs)\rangle$$

where Permitted xs states that xs is a reversed prefix of the naturals, and I is a client-chosen predicate that can be used to track which naturals have already been generated. The idea is that when the generator suspends, it sends natural number x to the client. The client has already received numbers xs, enforced with $I\ xs$, so x is the smallest natural that have not already been sent, enforced with Permitted $(x :: xs)$. Further, if the client wants to resume the generator again, they must 'admit' that they have now received x, i.e. $I\ (x :: xs)$. Using this protocol, we were able to give and prove a precise specification for the $sum_until function. Details of specifications and proofs can be found in the Rocq mechanisation.