Formal Reasoning about WebAssembly and Extensions

Maxime Robert Sébastien Legoupil

PhD Dissertation

Department of Computer Science
Aarhus University
Denmark

Formal Reasoning about WebAssembly and Extensions

A Dissertation
Presented to the Faculty of Natural Sciences
of Aarhus University
in Partial Fulfilment of the Requirements
for the PhD Degree

by
Maxime Robert Sébastien Legoupil
30th September 2025

Abstract

WebAssembly is a low-level bytecode celebrated for its clever design, making it fast, safe, portable and compact. It is designed as a compilation target and provides a better alternative to other targets like Javascript, which enjoys near-universal portability but suffers from a wide range of security and performance disadvantages especially when used as a compilation target. The speed at which WebAssembly has been adopted as well as the sheer amount of research about the language are testaments to how influential WebAssembly will be for many years to come.

Rather uniquely for a language this size and with this widespread use, WebAssembly was designed with a formally defined small-steps operational semantics, opening the door to computer-assisted proofs about the behaviour of the language. This allows for much more precise, accurate and trustworthy results than what is achievable using conventional techniques like testing or hand-written proofs, with the ultimate goal of improving the language and encouraging its adoption.

To this end, Chapter 2 of this dissertation introduces the largest effort in formal reasoning about WebAssembly to this date: Iris-Wasm, a higher-order modular program logic for the full industrial definition of WebAssembly 1.0, defined in the Iris separation logic and mechanised in the Rocq proof assistant, accompanied by a logical relation that formally captures and proves the fundamental WebAssembly property of local state encapsulation.

While Iris-Wasm is a useful result in understanding and validating the design of WebAssembly, its biggest potential is in applications at the cutting edge of WebAssembly development. The vibrant research ecosystem around WebAssembly proposes new extensions to WebAssembly on a regular basis, seeking to enhance performance, expressivity or security. These new versions of WebAssembly have sometimes not been extensively tested yet, and hence formal results in machine-backed separation logic are a precious asset in validating and improving the proposals.

In this dissertation, we present two such applications of Iris-Wasm. In Chapter 3, we introduce Iris-MSWasm, an extension of Iris-Wasm to Memory-Safe WebAssembly (MSWasm), a version of WebAssembly that uses hardware capabilities as a strategy for achieving memory safety. While WebAssembly has a strong focus on security and does provide memory safety guarantees – as we prove formally using Iris-Wasm–, the usage of integers as pointers and the lack of a mechanism for selectively sharing fragments of a module's memory with other modules have been shown to expose WebAssembly to a number of memory safety vulnerabilities. MSWasm suggests

replacing integers as pointers by hardware capabilities, a form of unforgeable tokens of authority. These capabilities cannot be obtained by any unpriviledged operation and memory cannot be accessed without a capability, opening the door to finer-grained memory sharing.

When defining Iris-MSWasm, we identified mistakes in the original proposal, completed its design and formally stated and proved the properties achieved by the introduction of capabilities: robust capability safety, which we establish by extending Iris-Wasm's logical relation.

In Chapter 4, we introduce Iris-WasmFX, an extension of the Iris-Wasm program logic to stack switching, a version of WebAssembly with effect handlers. Effect handlers are a popular programming paradigm involving complex control flow patterns. Compiling an effectful program into WebAssembly requires a program-wide transformation (e.g. continuation-passing style, CPS), introducing performance overheads and making the resulting code unlinkable with other WebAssembly code not obtained via a transformation that uses the same CPS conventions. The WebAssembly stack switching extension adds new instructions to the language to implement a very general version of effect handlers, with the aim of being an easy compilation target from most effectful source languages, regardless of what exact variation of effect-handling techniques is used.

To validate this proposal, we introduce Iris-WasmFX, a mechanisation of the semantics of stack switching and a program logic for key features of the extension. Just like for MSWasm, this allows us to correct, complete and validate the proposal formally. Our program logic was designed to allow modular reasoning and represents a framework for better understanding programs that involve complicated control flow strategies.

Iris-Wasm has the potential to be used for many more extensions of WebAssembly and serve as a tool to streamline verification, improvement and validation of new proposals.

Resumé

WebAssembly er en lavniveau byte-code som er kendt for sit smart design som gør den hurtig, sikker, bærbar og kompakt. WebAssembly er designet som et kompileringsmål og er et bedre alternativ til andre kompileringsmål som Javascript, som er næsten universelt bærbar men lider fra forskellige ulemper når det gælder sikkerhed eller effektivitet, især når den er brugt som kompileringsmål. Den hurtige adoption af WebAssembly og mængden af forskning omkring det viser tydeligt hvor indflydelsesrig WebAssembly kommer til at være i mange år.

Ret unikt for et sprog med denne størelse og denne brugsbredde, var WebAssembly designet med en formelt defineret operationel semantik i små trin, hvilket åbner dørren til computerassisteret beviser om sprogets opførsel. Dette tillader mere præcise, nøjagtige og troværdige resultater end hvad kan nåes med mere konventionelle teknikker som testning eller håndskrevne beviser, med som det endelige formål forbedring af sproget og tilskyndelse til dets vedtagelse.

Med dette formål introducerer kapitel 2 i denne afhandling det største projekt i formel begrundelse i WebAssembly på nuværende tidspunkt: Iris-Wasm, en højere ordens programlogik for den fulde industrielle definition af WebAssembly 1.0, defineret i Iris separationslogikken og mekaniseret i Rocq bevisassistenten, sammen med en logisk relation som formelt beskriver og beviser den fundamentale egenskab af WebAssembly sproget: lokalstatsindkapsling.

Selvom Iris-Wasm er et brugbart resultat som hjælper med at forstå og validere designet for WebAssembly sproget, er dens største potentiale i brug på forkant med WebAssembly udviklingen. Det levende forskningsøkosystem omkring WebAssembly foreslår nye udvidelser hele tiden, med formålet at få bedre effektivitet, ekspressivitet eller sikkerhed. Nogle af disse nye versioner af WebAssembly har ikke endnu blevet testet særlig meget, hvilket betyder at formelle resultater i computerstøttet separationslogik er et værdifuldt aktiv for at validere og forbedre forslagene.

I denne afhandling præsenterer vi to sådane brug af Iris-Wasm. I kapitel 3 introducerer vi Iris-MSWasm, en udvidelse af Iris-Wasm til en hukommelsessikker variation af WebAssembly: Memory-Safe WebAssembly (MSWasm). Denne version bruger hardware kapaciteter for at forsikre hukommelsessikkerhed. Selvom WebAssembly har et starkt fokus på sikkerhed og bringer nogle hukommelsessikerhedsgarantier – som vi formelt beviser med Iris-Wasm–, brugen af heltal værdier som pointere og mangel for en måde at selektivt dele et fragment af en moduls hukommelse med andre moduler har vist sig at årsage en del sikkerhedssårbarheder i hukommelsen.

MSWasm foreslår at erstate heltal som pointere med hardware kapaciteter, en form for uforfalskelige autoritetstegn. Disse kapaciteter kan ikke skabes af en uprivilegeret operation og man kan kun få adgang til hukommelsen med brug af en kapacitet, hvilket åbner døren til finere-kornet deling.

Når vi definerede Iris-MSWasm, identificerede vi fejl i det oprindelige forslag, færdiggjorde dets design og formelt beskrev og beviste de nye egenskaber som kapaciteterne bringer: robust kapacitetssikkerhed, hvilket vi etablerer ved at udvide den logiske relation i Iris-Wasm.

I kapitel 4 introducerer vi Iris-WasmFX, en udvidelse af programlogikken i Iris-Wasm til stack switching (dvs stakskift), en version af WebAssembly med effekthåndterer. Effekthåndterer er et populariskt programmeringsparadigme som involverer komplekse kontrolflowmønstre. Når man kompilerer et program med effekter til WebAssembly behøver man gøre en programomfattende transformation (f.eks. fortsættelsespasseringsstil, på engelsk Continuation-Passing Style, CPS), hvilket gør kompileringen ueffektiv og resulterer i kode som ikke kan linkes med andre WebAssembly programmer der ikke følger samme CPS konventioner. WebAssembly stack switching udvidelsen tilføjer nye instruktioner for at implementere en meget generel version af effekthåndtererne, for at være et enkelt kompileringsmål fra de fleste programmeringssprog med effekter, uanset hvilken præcise variation af effekthåndteringsteknikkerne er brugt.

For at validere dette forslag, introducerer vi Iris-WasmFX, en mekanisation af semantikken af stack switching og en programlogik for de vigtigste delene af udvidelsen. Ligesom med MSWasm kunne vi formelt korrigere, færdiggøre og validere forslaget. Vores programlogik er designeret så at man kan ræsonnere modulært, og er et rammeværk for at bedre forstå programmer der involverer komplicerede kontrolflow strategier.

Iris-Wasm har potentialet for at blive brugt for mange flere udvidelser af WebAssembly og kan være en værktøj for at hurtigere verificere, forbedre of validere nye forslag.

Resumé

WebAssembly est un bytecode de bas niveau, célébré pour son design intelligent qui le rend rapide, sûr, portable et compact. Il est conçu comme un langage cible de compilation et apporte une meilleure alternative aux cibles existantes comme Javascript, qui jouit d'une portabilité quasi-universelle mais souffre de nombreux inconvénients en termes de sécurité et de performance, notamment quand il est utilisé comme cible de compilation. La vitesse avec laquelle WebAssembly a été adopté ainsi que la quantité de recherche autour du langage attestent de l'influence que le language aura pendant des années à venir.

De façon très inhabituelle pour un langage de cette taille et avec une utilisation si vaste, WebAssembly a été conçu avec une sémantique opérationnelle à petits pas définie formellement, ce qui ouvre la porte à des preuves assistées par ordinateur à propos du comportement du langage. Ceci permet des résultats plus précis, exacts et dignes de confiance que ce qui peut être obtenu par des méthodes plus conventionnelles comme les tests ou les preuves écrites à la main, avec pour but ultime d'améliorer le langage et d'en encourager l'adoption.

Dans cette optique, le chapitre 2 de cette dissertation présente le plus grand projet de raisonnement formel à propos de WebAssembly à ce jour : Iris-Wasm, une logique de programme modulaire et d'ordre supérieur pour l'intégralité de la définition industrielle de WebAssembly 1.0, définie dans la logique de séparation Iris et mécanisée dans l'assistant de preuve Rocq, accompagnée par une relation logique qui capture et prouve formellement la propriété fondamentale de WebAssembly : l'encapsulation de l'état local.

Si Iris-Wasm est un outil précieux pour comprendre et valider le design de WebAssembly, son plus gros potentiel est dans ses applications à la pointe du développement de WebAssembly. L'écosystème de recherche vibrant qui entoure WebAssembly propose régulièrement de nouvelles extensions au langage, cherchant à améliorer la performance, l'expressivité ou la sécurité. Ces nouvelles versions de WebAssembly n'ont pas toujours encore été testées extensivement, et des résultats formels dans une logique de séparition vérifiée par ordinateur sont donc des atouts précieux pour valider et améliorer les propositions.

Dans cette dissertation, nous présentons deux telles applications d'Iris-Wasm. Dans le chapitre 3, nous présentons Iris-MSWasm, une extension d'Iris-Wasm pour une version de WebAssembly qui utilise des capacités matérielles pour améliorer la sécurisée-mémoire : Memory-Safe WebAssembly (MSWasm). Si WebAssembly est

conçu avec une attention particulière pour la sécurité, et possède déjà des garanties de sécurité mémoire – comme nous le prouvons formellement avec Iris-Wasm–, il a été démontré que l'utilisation d'entiers comme pointeurs et l'absence d'un mécanisme pour partager sélectivement des fragments de la mémoire d'un module à d'autres modules exposent WebAssembly à un certain nombre de vulnérabilités-mémoire. MSWasm suggère de remplacer les entiers comme pointeurs par des capacités matérielles, une forme de jetons infalsifiables d'autorité. Ces capacités ne peuvent être créées que par des opérations privilégiées et l'accès à la mémoire n'est possible qu'en détenant une capacité, ce qui ouvre la porte à des partages à grain plus fin.

Lorsque nous avons défini Iris-MSWasm, nous avons identifié des erreurs dans la proposition originale, nous avons complété sa structure et avons formellement formulé et prouvé la propriété apportée par l'introduction de capacités : la sûreté robuste de capacité, que nous établissons en étendant la relation logique d'Iris-Wasm.

Dans le chapitre 4, nous présentons Iris-WasmFX, une extension de la logique de programme d'Iris-Wasm à stack switching (c'est-à-dire échange de piles), une version de WebAssembly avec des manipulateurs d'effets. Les manipulateurs d'effets sont un paradigme de programmation très utilisé, qui s'appuie sur des flots de contrôle complexes. La compilation d'un programme avec effets vers WebAssembly nécessite une transformation complète du programme (par exemple par passage de continuation, en anglais Continuation-Passing Style, CPS), ce qui nuit à la performance et fait que le code résultant ne peut pas être lié avec d'autres programmes WebAssembly qui ne suivent pas les mêmes conventions CPS. L'extension stack switching de WebAssembly ajoute de nouvelles instructions au langage afin d'implémenter une version très générale des manipulateurs d'effets, dans le but d'être une cible facile pour la compilation depuis la plupart des langages avec effets, ce quelle que soit la variante exacte des techniques de manipulation des effets utilisée.

Afin de valider cette proposition, nous présentons Iris-WasmFX, une mécanisation de la sémantique de stack switching et une logique de programme pour les éléments clés de cette extension. Comme pour MSWasm, ceci nous a permis de corriger, compléter et valider formellement cette proposition. Notre logique de programme permet de raisonner modulairement et constitue un cadre idéal pour mieux comprendre des programmes qui impliquent une stratégie complexe de flot de contrôle.

Iris-Wasm a le potentiel d'être utilisé pour de nombreuses autres extensions de WebAssembly et de servir d'outil pour accéler la vérification, l'amélioration et la validation de nouvelles propositions.

Tiivistelmä

WebAssembly on matalan tason tavukoodi jota kunnioitetaan sen älykkään suunnittelusta, joka tekee sen nopeaksi, turvalliseksi, kannettavaksi ja kompaktiksi. Sen on suunniteltu ohjelmointikielen kääntäjän kohdekieleksi ja tarjoaa paremman vaihtoehdon muille kohdekielille kuten Javascript, jolla on lähes universaalinen kannettavuus mutta kärsii monesta turvallisuus- ja tehokkaisuusviasta, varsinkin kun sitä käytetään kohdekielenä ohjelmointikielen kääntämisessä. Nopeus jolla WebAssembly:n on adoptoitu ja suuri määrä tutkimusta sen ympärillä ovat todistuksia siitä, että WebAssembly tulee olemaan vaikutusvaltainen ohjelmointikieli moniin vuosiin.

Aika harvinaisesti tämän kokoiselle ja näin laajasti käytetylle ohjelmointikielelle, WebAssembly on luotu muodollisesti määriteltyllä operatiivisella semantiikalla, joka avaa oven tietokeneella avustetuille todistuksille kielen käyttäytymisestä. Tämä tuo paljon tarkempia, todellisempia ja luotettavempia tuloksia kun mitä voi saada tavanomaisin keinoin kuten testaamisella tai käsin kirjoitetuin todistuksin, lopullisena tavoitteena kielen parantaminen ja sen käytön kannustaminen.

Tätä vuoksi esitellään tämän väitöskirjan luvussa 2 tähän mennessä suurin projekti muodollisessa päättelyssä WebAssembly:sta: Iris-Wasm, modulaarinen korkeamman asteen ohjelmalogiikka WebAssembly 1.0 version täydelle teolliselle määritelmälle, määritelty Iris erottelulogiikassa ja koneistettu Rocq todistusavustajassa, yhdessä loogisen suhteen kanssa, joka muodollisesti kuvailee ja todistaa WebAssembly:n tärkein perusominaisuus: paikallisen tilan kapselointi.

Vaikka Iris-Wasm on hyödyllinen tulos WebAssembly:n sunnittelun ymmärtämisessä ja validoinnissa, sen suurein potentiaali on sen käyttämisessä WebAssembly:n kehityksen eturintamassa. Elivoimainen tutkimusekosysteemi WebAssembly:n ympärillä ehdottaa koko ajan uusia laajennuksia, tavoitteena parantaa tehokkuutta, ilmeikkyyttä tai turvallisuutta. Näitä uusia WebAssembly versioita ei välttämättä aina ole laajasti testattu, ja niinpä muodolliset tulokset tietokoneellisesti tuetussa logiikassa ovat arvokkat omaisuudet ehdotusten validoinnissa ja parantamisessa.

Tässä väitöskirjassa esitellään kahta sellaista käyttöä Iris-Wasm ohjelmalogiikalle. Luvussa 3 esitellään Iris-MSWasm, laajennus Iris-Wasm ohjelmalogiikasta uudelle versiolle Memory-Safe WebAssembly (eli muisti-turvallinen WebAssembly, lyhennetty MSWasm). Siinä versiossa käytetään laitteistokykyjä muisti-turvallisuuden saatavaksi. Vaikka WebAssembly:lla on kova painopiste turvallisuudessa ja vaikka se tuo jotain muisti-turvallisuus takuja – kuten todistetaan muodollisesti Iris-Wasm:n avulla –, kokonaislukujen käyttö osoittimina ja mekanismin puute, jolla voisi va-

likoivasti jakaa osan yhden moduulin muistista toisten moduulien kanssa, ovat todistettu aiheuttavan kielessä monta muisti-turvallisuuteen liittyviä haavoittuvuuksia. MSWasm ehdoittaa käyttää osoittimina kokonaislukujen sijaan laitteistokykyjä, eli eräänlaisia väärentämättömiä auktoriteetin rahakkeita. Näitä kykyjä pystyy vain etuoikeutetuilla operaatioilla luoda, ja ilman niitä ei saa pääsylupaa ohjelman muistiin. Niinpä kyvyt avaa oven hienorakeisemalle jakaamiselle.

Määrittelemässämme Iris-MSWasm, tunnistimme muutaman virheen alkuperäisessä ehdotuksessa, viimeistelimme sen suunnitusta ja lausuimme ja todistimme muodollisesti uuden ominaisuuden jonka kyvyt tuo: vankka kykyturvallisuus, jonka me perustetamme Iris-Wasm:n loogisen suhteen laajentamisella.

Luvussa 4 esittelemme Iris-WasmFX, laajennus Iris-Wasm ohjelmalogiikasta stack switching (eli pinokytkentä) laajennukselle. Tässä uudessa versiossa WebAssembly:lle lisätään vaikutuskäsittelijät. Vaikutuskäsittelijät ovat suosittu ohjelmoitiparadigma johon liittyy monimutkaiset ohjausvirtauskuviot. Ohjelmoitikielen kääntäminen WebAssembly:iin kun lähdekielessä on vaikutuksia aiheuttaa koko ohjelman muutos (esimerkiksi jatkosyöttötyylissä, englanniksi Continuation-Passing Style, eli CPS), joka aiheutta pitemmän suoritusajan ja tarkoittaa että tuotettu koodi ei voi suorittaa muun WebAssembly koodin kanssa jos muu koodi ei seuraa samoja CPS yleissopimuksia. Stack switching laajennus lisää uusia ohjelmointiohjeita kieleen, toteuttaakseen hyvin yleisen version vaikutuskäsittelijöistä, tavoitteena voida helposti toimia ohjelmointikielen kääntäjän kohdekielenä suurimman osalle vaikutuksia sisältäville lähdekielille, riippumatta mitä tiettyä versiota vaikutuskäsittelytekniikkoja on käytetty.

Validoidaksemme tämän ehdoituksen esittelemme Iris-WasmFX, koneellistaminen uudesta stack switching semantiikasta ja ohjelmalogiikka laajennuksen tärkeimmille osille. Yhtälailla kun MSWasm projektissa, pystyimme siten korjata, viimeistellä ja validoida ehdoituksen muodollisesti. Ohjelmalogiikamme on suunniteltu siten, että pystyy modulaarisesti päätellä, ja sen avulla voi paremmin ymmärtää ohjelmia jossa on monimutkainen ohjausvirtausstategia.

Iris-Wasm ohjelmalogiikalla on potentiaali olla käytetty monien muitenkin WebAssembly:n laajennusten tutkimuksessa, ja se voi olla työkalu uusien ehdotusten vahvistuksen, parannuksen ja validoinnin virtaviivaistamisessa.

Acknowledgments

First and foremost, my deepest gratitude goes to my advisors, Lars Birkedal and Jean Pichon-Pharabod, who have been vital pillars for me during all four years that I have spent in Aarhus. You both are an endless source of inspiration for me and have always known how to provide me guidance in navigating research, publications, and life in general. Thank you for your groundedness, your sense of humour, and your infinite friendliness.

To everyone else at the Logic and Semantics research group, thank you for your kindness and your warmth. You have collectively created the best working environment imaginable, which has made me feel safe, at ease, and even at home while so far away from my own. I will miss our microwave-side conversations, our cake breaks, our running sessions and our hours-long Sporz games. To my officemate Alejandro Aguirre in particular, thank you for always having been a smiling presence, ready to help whenever I asked.

To my coauthor Aïna Linn Georges, thank you having been such a good mentor at the beginning of my PhD. Your help in learning the ropes of large-scale Rocq programming, understanding capabilities and logical relations, and navigating the world of academia have been crucial and have shaped the trajectory of my career. Thank you as well to Rao XiaoJia, who together with Aïna wrote most of the code for Iris-Wasm. The style in which I write code today has been vastly influenced by all the time I spent expanding on the code the two of you wrote.

Thank you to all other of my co-authors, it has been a pleasure to collaborate with all of you and I look forward to continuing our work together.

Thank you to Sam Lindley for hosting me at the University of Edinburgh during three months. It has been an absolute delight to discover Scotland and I hope our plans for working together in Edinburgh in the near future can soon become true.

Thank you to Conrad Watt for being such a leading force in the WebAssembly research community. It has been a pleasure to work with you on the Iris-Wasm project and I look forward to working with you on SpecTec in Singapore in a few months.

Thank you to Philippa Gardner for your energy and creativity. I hope we soon get a new opportunity to collaborate.

Thank you to Brianna Marshall, Ryan Doenges, Amal Ahmed and all the others on the Iris-RichWasm team at Northeastern University. It has been a pleasure to see Iris-Wasm be useful to this new exciting project, and I look forward to continuing our collaboration.

Thank you to Mathias Pedersen for being such an avid learner of Iris-Wasm. I hope you find great things to do with it, and I look forward to helping you navigate the project.

Thank you to Andreas Rossberg and Sandrine Blazy for agreeing to be on my thesis committee. I am deeply admirative of your work and am honoured to have you both on the committee.

To everyone in the Aarhus Step community, thank you for showing me what has become my favourite form of cardio. Practicing my near-daily classes of Step has been incredibly helpful in getting all of my energy out; I could not have gone through my entire PhD programme without all of you. Thank you in particular to Mette Riishøj Væggemose Svendsen for being my mentor in Step, for training me as an instructor, and for always delivering the best classes. I hope your baby is a sunshine in your life, and that we can soon see you teach again at the gym. I would also like to thank all of the international Step Presenters I have met over the years. I look forward to stepping again with all of you soon.

I would also like to extend my gratitude to everyone who has steered me into computer science when I was studying in Paris. Thank you to Jean-Christophe Filliâtre for your fascinating introduction to the world of compilers, and to Jean Goubault-Larrecq for your equally as fascinating introduction to lambda-calculus. Thank you also to my internship mentors at Inria, Tamara Rezk and Gabriel Scherer. It has been a pleasure to grow as a researcher with you. Thank you as well to Gabriel for pointing me in the direction of Lars and Aarhus.

Lastly, I would like to thank my entire family for always believing in me and encouraging me to pursue my passions. Thank you in particular to my parents Guillaume and Leena, I love you always.

Maxime Robert Sébastien Legoupil,
Aarhus, 30th September 2025.

Contents

Abstract i

Resumé (*dansk*) iii

Resumé (*français*) v

Tiivistelmä vii

Acknowledgments ix

Contents xi

I Overview 1

1 Introduction 3
1.1 WebAssembly . . . 3
1.2 Formal Reasoning . . . 6
1.3 Iris-Wasm . . . 10
1.4 Contributions and Structure . . . 19

II Publications 23

2 Iris-Wasm 25
2.1 Introduction . . . 26
2.2 Modular Reasoning for WebAssembly Modules . . . 30
2.3 Host Language and Proof Rules . . . 42
2.4 Mechanization in the Iris Framework . . . 47
2.5 Case Study . . . 48
2.6 Related Work . . . 52
2.7 Conclusion . . . 53
2.A The Full Iris-Wasm Program Logic . . . 54
2.B Logical Relation . . . 66
2.C Case Study: Landin's Knot . . . 71

3 Iris-MSWasm **73**
3.1 Introduction . . . 74
3.2 The MSWasmCert Semantics . . . 77
3.3 Program Logic . . . 86
3.4 Robust Capability Safety . . . 95
3.5 Stack Example . . . 102
3.6 Discussion and Related Works . . . 104
3.A Proof rules . . . 107
3.B Syntactic Typing for MSWasm . . . 111
3.C Logical Relation . . . 112
3.D Fundamental Theorem . . . 118

4 Iris-WasmFX **119**
4.1 Introduction . . . 120
4.2 A Coroutine Library . . . 121
4.3 WasmFXCert . . . 123
4.4 Iris-WasmFX . . . 139
4.5 Case Study: Coroutine Library . . . 153
4.6 Related Work . . . 159
4.7 Limitations . . . 161
4.8 Conclusion . . . 161

Bibliography **163**

Part I

Overview

Chapter 1

Introduction

In this dissertation, we introduce Iris-Wasm, a tool to reason formally about WebAssembly, and extensions to Iris-Wasm that can be used to reason formally about extensions of WebAssembly. In this introduction, we answer three general questions:

- What is WebAssembly and why should people care about the work detailed in this dissertation (§ 1.1)?
- What formal reasoning tools are we using in this dissertation (§ 1.2)?
- How do the works in this dissertation fit together and what is the future of Iris-Wasm (§ 1.3)?

We finish the introduction by summarising the dissertation's contributions and giving an outline of the rest of the document (§ 1.4).

First, we give a general overview of WebAssembly. A reader familiar with WebAssembly can safely skip ahead to § 1.2.

1.1 WebAssembly

WebAssembly [41] is a low-level bytecode designed to be safe, fast, portable and compact. Its creation was motivated by the need for a language that would be both widely supported on the Web, and adapted as a compilation target. The original creators observed [41, §1] that Javascript was, by 'historical accident,' the 'only natively supported programming language on the Web,' but that it 'has inconsistent performance and a number of other pitfalls, especially as a compilation target,' hence the need for WebAssembly.

Today, WebAssembly is supported by 96% of all browser installations worldwide.[1] The scope of WebAssembly has also grown far beyond the web. In a September 2023 survey [119], 58% of respondents indicated using WebAssembly for web applications, 35% for data visualisation, 32% for the internet of things [111], 30% for artificial

[1] `https://caniuse.com/#search=WebAssembly`

intelligence, 28% for games, 27% for backend services, and 25% for edge computing. Other reported usecases include platform emulation, audio, video and image processing, augmented or virtual reality, and serverless computing [42].

Given WebAssembly was designed as a compilation target, it is unsurprising that there are a great number of compilers[2] targetting it, like WasmPack for Rust, Emscripten for C, Bolero for F♯, GHC for Haskell, j2wasm for Java, or Pyodide for Python. The length of this list is a testament to how wide the community of WebAssembly users is.

Another way to appreciate the importance WebAssembly has taken, is to look at the vibrant research ecosystem that surrounds it. The official standard has grown from WebAssembly 1.0 [95] to WebAssembly 2.0 [99] and version 3.0 is soon to come out; and numerous extensions have been proposed,[3] sometimes to enhance performance, sometimes to obtain additional safety guarantees. Some of these extensions include the addition of a garbage collector [92], multiple memories [97], exception handling [4], or tail call optimisation [93]. Out of all these proposals, we have identified three as particularly well-suited for formal reasoning, and have been studied using Iris-Wasm: Memory-Safe WebAssembly [24, 69] (§ 1.3.1), stack switching [66] (§ 1.3.2) and RichWasm [30] (§ 1.3.3). The first and the third of these achieve stronger safety properties than plain WebAssembly, which we thought would be interesting to verify formally; and the second introduces elements that make intuitive reasoning more complicated, hence we though formal tools could be useful. We describe all of these is more detail in § 1.3.

The developpers of WebAssembly define 5 phases[4] for all new proposals together with a set of requirements for each proposal to reach each of the 5 stages. These requirements include the usual reference implementation, example suites and prose descriptions of the semantics, but also the much less common requirement for a formal specification given in the form of an operational semantics together with a formal statement of safety and security properties. This is an outstanding opportunity for research to be done with a real-world impact, and ensures the new variations of WebAssembly are all built to the same strong standard.

The strengths of WebAssembly are:

- It is fast, running at near-native speeds using just-in-time or ahead-of-time compilation
- It is portable, allowing the same `.wasm` binaries to be used on most browsers and operating systems
- It is safe, providing a sandboxed environment to run code

[2] A full list of existing compilers can be found at `https://webassembly.org/getting-started/developers-guide/`

[3] A full list of official proposals can be found at `https://github.com/WebAssembly/proposals?tab=readme-ov-file`. There are also proposals that do not make an attempt at officially entering the standard and are therefore not included in this list.

[4] `https://github.com/WebAssembly/meetings/blob/main/process/phases.md`

- It is mostly deterministic: apart from a few instructions whose semantics must involve non-determinism, execution is entirely predictible

- It is compact, having a light-weight representation

- It is well-adapted as a compilation target, and most languages programmers use have a compiler to WebAssembly

- It has a well-defined semantics in an official standard [95, 99], allowing for formal reasoning (see § 1.2.1).

Safety and security are a key aspect of WebAssembly, and is crucial for most users, as Lehmann et al. [61] point out: 'on the client side, users run untrusted code from websites in their browser; on the server side in Node.js, WebAssembly modules operate on untrusted inputs from clients; in cloud computing, providers run untrusted code from users; and in smart contracts, programs may handle large sums of money'.

This security is achieved through several different design choices in the language, like separating the code and the call-stack from the memory, enforcing static typechecking, and not defining arbitrary go-to instructions and instead relying only on structured control flow.

In this dissertation, we also show formally that WebAssembly has a property called *local state encapsulation*. Code is organised in modules, each defining its own objects like functions, global variables, memories, etc. Modules can import objects defined by other modules, but only if those objects have been explicitly marked as exported. This property is widely relied upon in a lot of uses of WebAssembly.

The process of linking modules together is called *instantiation*, and is not performed by WebAssembly itself but rather by a *host language* that embeds the WebAssembly code. When the host language instantiates a module, it typechecks it, fetches all of its imports from modules already instantiated, and prepares the objects in the module that have been explicitely tagged as exported for subsequent imports by other modules. WebAssembly code can also invokes functions defined by the host language, and the host language can usually also run WebAssembly functions or inspect WebAssembly state. Javascript is often used as a host language to run WebAssembly. For simplicity, all three works in this dissertation consider the same simplified custom-built host language that emulates the key features of the Javascript-WebAssembly interface.

The large amount of tools, libraries and research surrounding WebAssembly illustrates how it is poised to be an influencial language for decades to come. WebAssembly offers a promising base to eliminate many safety risks, but in the wake of constantly evolving threats like Meltdown or Spectre [88], its vulnerabilities [61] have raised concerns and it is therefore important to continue studying it closely.

1.2 Formal Reasoning

Conventionally, the easiest way to show that a program or even a full programming language has a desired property is to test it on a wide variety of sample inputs. The size and variedness of the test suite then dictates what level of confidence one can have in the accuracy of the result. The disadvantage is obvious: unforeseen cases can go untested and lead to failures when the case arises organically. Instead, *formal* reasoning refers to the effort to reason mathematically about a language's operational semantics, often writing proofs of quantified statements where by definition all cases are considered.

While formal reasoning has been done by hand for centuries, the golden standard today is to write a computer-checked proof using a proof assistant like Rocq, Lean, Agda or Isabelle. Computer-assisted proofs require a very precisely and pedantically phrased theorem and an equally as pedantically conducted proof, which means both that the process is arduous, time-consuming and requires expertise, but also that the end result is water-tight and can be trusted with the greatest level of confidence.

In the domain of programming languages in particular, models quickly become enormous in size, meaning both that it can be difficult to assess the exhaustiveness of a test suite, and that hand-written proofs become prone to forgotten edge-cases or other accidental falacies. One of the points in defining WebAssembly with such a precise specification was precisely to facilitate formal reasoning, and as such a great amount of research has been done into mechanising and understanding the behaviour of WebAssembly's semantics. This thesis introduces the most ambitious case of formal reasoning for WebAssembly. In this section, we present the existing works that tackle formal reasoning about WebAssembly (§ 1.2.1), then we give a brief definition of a program logic and introduce Iris (§ 1.2.2), and finally we define logical relations (§ 1.2.3).

1.2.1 Reasoning about WebAssembly

Let us begin by giving an overview of works that pertain to reasoning formally about other low-level languages. RockSalt [71] is a checker, mechanised and verified in the Rocq proof assistant, that statically inspects code binaries and enforces a sandbox policy. A macro assembler has also been defined in Rocq [54], allowing to relate machine code to logic formulas.

The Certified Assembly Programming project [28, 77, 140, 141] defines frameworks for second-order Hoare logics for low-level assembly languages, supporting modular specifications and features including concurrency, dynamic thread creation and embedded code pointers; hence a focus on features not present in WebAssembly. CertiKOS [39] is an extensible architecture that can be used to establish specifications for concurrent operating system kernels, that has been used [39, 40] to define and verify a concurrent operating system kernel involving C and x86 code. The interaction of these two languages can be reasoned about by using CompCertX [38], assuming both languages have the same memory model. If the languages do not share a common

memory model, the DimSum [103] project has shown promising results in idealised language fragments.

Given WebAssembly's formally defined operational semantics, it is uniquely simple to mechanise in a proof assistant. Hence it is no surprise that WebAssembly too has had seen a great number of works tackling formal reasoning about the WebAssembly semantics. This includes mechanisations, most notably WasmCert in Rocq and Isabelle [8, 131, 133]. It is this Rocq mechanisation that the entire work of this thesis builds on. The mechanisation consists of a full description of the operational semantics and the typing system, as well as an interpreter, a pretty-printer, an automated type-checker (proven sound), and a proof of type soundness by progress and preservation. A first-order program logic for WebAssembly 1.0 had also been defined in Isabelle [132], but the first-orderness limits the expressivity and modularity. Hence this thesis introduces Iris-Wasm (see § 1.3), a full higher-order program logic and a logical relation, allowing for very general reasoning about behaviour of programs in WebAssembly.

Finally, let us mention SpecTec [139], a more general WebAssembly project. While designed for broader purposes than just formal reasoning, it will nonetheless likely have a huge impact on the ease of doing computer-assisted proofs about WebAssembly. SpecTec aims to simplify the process of defining WebAssembly and extension proposals, by defining a domain-specific language (DSL) in which the operational semantics and typing rules can be described. This description in that DSL then serves as a single source of truth when defining the interpreter, the examples, the prose description of the semantics, etc: all those elements are automatically generated by SpecTec. As long as one trusts SpecTec itself, one can be assured that all the different parts of the proposal are consistent with the DSL description and thus also with one another. For example, SpecTec has been used [62] to model the stack switching extension [66] (which we describe in § 1.3.2)

SpecTec is soon expected to also output mechanisations in different proof assistants like Rocq, Agda and Lean. While previous attempts [74] at automatically generating Rocq code have yielded results ill-adapted to mechanised soundness proofs, SpecTec is expected not to face similar issues specifically due to WebAssembly's design. Understanding the differences between SpecTec-generated code and the mechanisations this thesis uses/defines when defining our program logics is an interesting avenue for future research. Investigating whether SpecTec could also output at least partial fragments of a program logic is also an interesting avenue of future work.

1.2.2 Iris

Separation logic [80, 91] is a logical model, expanding on Hoare logic, that is used to reason about the programs. In this dissertiation, we focus on a specific framework for separation logic: Iris [49, 52], which has been mechanised in the Rocq proof assistant. It can be used to prove programs correct or more generally establish a wide range of language properties, including properties stronger than type soundness by the likes of contextual refinement and full functional correctness.

These properties are often phrased using *preconditions* (propositions describing the state before execution) and *postconditions* (propositions describing the state after execution). In this context, the word *proposition* means not simply a description of the state, but *ownership* of *resources*, the archetype of which is the points-to resource $\ell \mapsto v$ which corresponds to owning a singleton heap with only one location ℓ currently pointing to a value v. If a program owns the resource $\ell \mapsto v$ as part of its precondition, this means that it has the authority to do whatever it wants with this location in the heap: no other part of the program can meddle with location ℓ since the resource $\ell \mapsto v$ is exclusive. This makes separation logics like Iris very modular: one can reason about each fragments of the program in isolation.

One central notion in Iris is that of *weakest precondition*, noted $\mathsf{wp}\, e\, \{\Phi\}$, a predicate on programs and postconditions that means holding all necessary resources to run e safely and guarantee that in case of termination, the final value v satisfies $\Phi(v)$. With this notation,

$$P \mathrel{-\!\!*} \mathsf{wp}\, e\, \{Q\}$$

means that if we own precondition P, program e runs safely and if it terminates on a value v, postcondition $Q(v)$ hold.

To use Iris and its Rocq mechanisation, one need only describe the operational semantics of a programming language, and Iris then provides a logic in which one can define language-specific proof rules to reason about the behaviour of programs following the operational semantics given as an input. This means that the results proved in Iris are about the theoretical operational semantics, not any actual implementation of the language; one still needs to verify or trust that implementations faithfully correspond to the intended operational semantics. In this thesis, we instantiate Iris with WasmCert and reason about WebAssembly programs (see § 1.3).

While separation logic is famously a very powerful tool to reason about concurrent programs [10, 79], we show in Chapter 2 that this is also a useful concept when reasoning about a sequential language like WebAssembly. Furthermore, using a logic adapted to concurrent programming means that when dealing with WebAssembly extensions that emulate concurrency like stack-switching (see § 1.3.2), we automatically inherit the full strength of concurrent separation logic, as we illustrate in Chapter 4.

1.2.3 Logical Relations

A logical relation [85, 113] is a mathematical object that can be used as a proof strategy to prove deep program properties like termination, type soundness, contextual equivalence, or more. Its most famous and canonical example is the classic proof [113, §2.1] of normalisation of simply-typed lambda calculus. That proof illustrates well that some properties cannot be proved by a simple induction on a type derivation, showcasing the need for more creative mechanisms like logical relations.

In this section, we attempt to give an intuition of the general form logical relations usually take. For more precise details, see the sections associated with the logical relations defined in Iris-Wasm (Chapter 2) and Iris-MSWasm (Chapter 3).

A (unary) logical relation is often used to prove a language property of the type 'for all program e, if e has property P, then e has property Q,' where P is usually 'e typechecks syntactically'. The general strategy is then to define a predicate V roughly like this: $V(e)$ holds if all three conditions below hold:

1. e has property P
2. e has property Q
3. For all programs e' that e reduces to, $V(e')$

Then the proof structure is the following: we prove that for any program e that has property P, we have $V(e)$ (which then implies that e has property Q by virtue of point 2. in the definition). In the common case where P is the typechecking property, this is done by induction on the typing derivation; the presence of the third clause (stability under reduction) is what makes this proof work when a direct attempt at proving property Q by induction on the typing derivation would have failed.

Two of the works in this dissertation (Iris-Wasm and Iris-MSWasm) define a logical relation, in both case showing that programs that typecheck syntactically run safely (they *typecheck semantically*). This is powerful, as it allows us to reason about any code that typechecks syntactically. In practice, this is very useful when reasoning about code that invokes unknown functions (say, by doing library calls to an untrusted library). In that case, the program logic together with the logical relation allows for strong results even when reasoning about unknown code.

As we will see in those cases, the above rule of thumb for the structure of a proof by logical relation is often more complex in reality. The predicate V is actually a type-indexed family of predicates, and we define predicates for other components of the program like values, parts of the state, etc. Lastly, most of these predicates make use of the weakest precondition statement, instead of the 3-point rule-of-thumb definition shown above, as is common when defining logical relations in Iris [126]. The spirit of the proof remains the same, since the weakest precondition statement inherently contains a quantification on all program reductions, which respects the spirit of point 3. above.

In this context, when proving that syntactic typechecking implies semantic typechecking, the proof comes down to showing that for all instructions in the language, when given arguments that satisfy predicate V for the right type, the expression reduces to something that also satisfies precidate V for the right type. By definig V carefully, the proof thus boils down to showing that all instructions in the language satisfy some *universal contract* [47]. We give two examples of a universal contract in this dissertation: local state encapsulation in Iris-Wasm (see Chapter 2) and robust capability safety in Iris-MSWasm (see Chapter 3).

There also exists binary logical relations where predicates are defined on pairs of programs. These are useful to prove properties like contextual equivalence. The works in this dissertation only use unary logical relations.

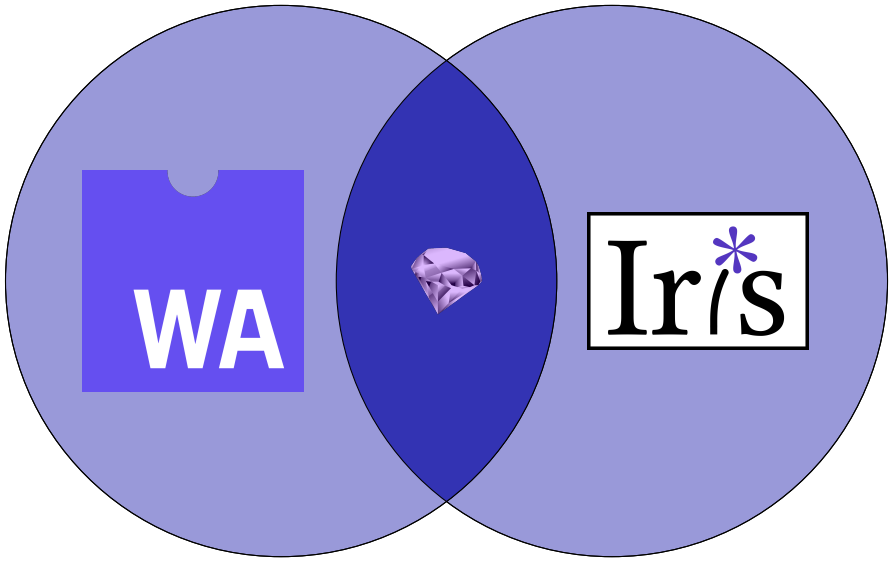


Figure 1.1: Roadmap to Iris-Wasm

1.3 Iris-Wasm

The first contribution of this thesis is Iris-Wasm, an instantiation of Iris with WebAssembly 1.0, which we describe in Chapter 2. Our program logic encompasses the entire operational semantics of WebAssembly 1.0, not just a idealised subset – with one purposeful departure from the official standard, which we describe in § 2.2.2.

Let us view Iris-Wasm as a little jewel at the intersection of WebAssembly and Iris, as is displayed in Fig. 1.1.

Iris-Wasm is the most ambitious project to date tackling formal reasoning about WebAssembly. Its strengths are:

Higher-orderness. The Iris logic is higher-order, so a user of Iris-Wasm can mention propositions, quantifications, functions etc in pre- and post-conditions of all specifications she defines. This means for example that when reasoning about a higher-order function, say, a $map function on stacks, the precondition of the specification can include a specification for the mapped function.

Modularity. The Iris-Wasm logic allows for giving standalone specifications to individual modules or even individual functions within module, and later link these specifications to reason about entire execution traces. Using the higher-orderness, one can also hide implementation details behind existential quantifying, further illustrating the isolation properties of WebAssembly.

Reentrancy. WebAssembly modules are instantiated by a host language, and WebAssembly code can call functions defined by this host language. Conversely, the host language can invoke WebAssembly functions or inspect or modify

WebAssembly state, hence opening the possibility for complex control flow back and forth between the WebAssembly level and the host level. In Iris-Wasm, we define a simple host language and a program logic for that host language, allowing for reasoning about those complex reentrant programs.

Robustness. On top of the program logic, we define a logical relation that formally proves that all code that typechecks syntactically runs safely when provided with exactly what resources it explicitely is allowed to import. This opens to door to reasoning about programs that invoke unknown, untrusted code, say, when making calls to functions from an untrusted library.

Local State Encapsulation. By defining the aforementioned program logic, we give the first formal proof that WebAssembly satisfies local state encapsulation: a module cannot arbitrarily meddle with the state of a different module, it can only interact with objects that that module explicitely tagged as exported.

But more importantly, Iris-Wasm can be used a starting point to reason about extensions to WebAssembly, like new versions or new variations that introduce new features. The WebAssembly research ecosystem is very dynamic and new extensions are proposed all the time, and these new extensions have not always been tested as rigourously as plain WebAssembly, hence formal results are a valuable asset in validating their models.

An extension to WebAssembly can often be thought of as the addition of a specific new feature. Because Iris also has a vibrant research community, it is often well understood how that feature can be reasoned about in theory. We therefore wondered if it might be possible to define an Iris program logic for these new extensions, using Iris-Wasm as a starting point, working with the diagram shown in Fig. 1.2.

What we found was that in most cases this scheme works and allows us to get a program logic for the new version of WebAssembly, albeit not quite as easily as taking the intersection of existing techniques as the diagram might suggest. The work involved in massaging the existing techniques to obtain the trophy at the centre of the diagram is non-trivial, but we argue it is worth the effort, as every side wins something:

- The WebAssembly community gains better understanding and trust in its new extension
- The Iris community gets to see the Iris framework exercised on a real-world industrial scale, highlighting the usefulness of individual projects and spotlighting areas where new needs exist
- The community for the new feature gets to see formal reasoning about that feature at a real-world industrial scale, where they previously often only had theoretical results or less formal results at the industrial scale

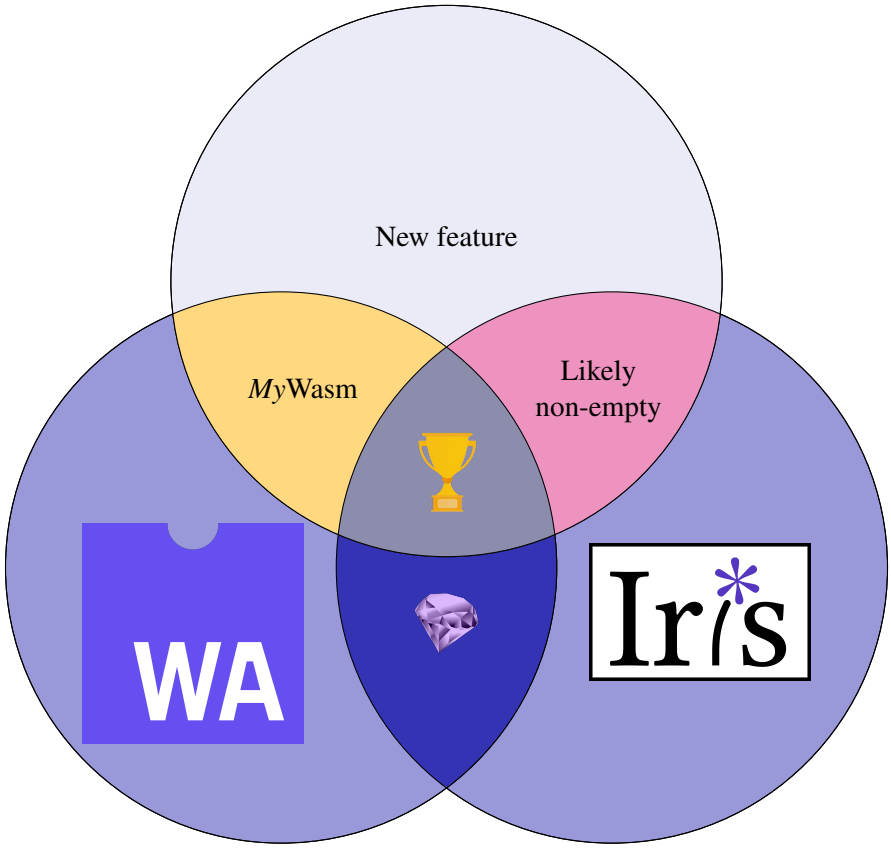


Figure 1.2: Roadmap to using Iris-Wasm

The two other contributions of this dissertation are Iris-MSWasm and Iris-WasmFX, program logics for two extensions of WebAssembly: MSWasm and stack switching. These were built atop Iris-Wasm and have been useful in refining and validating proposals as well as increasing confidence and thus encouraging adoption of both MSWasm and stack switching. In the rest of this section, we introduce the background to these two contributions, as well as to Iris-RichWasm, another work in progress that seeks to follow a similar roadmap to reason about RichWasm.

1.3.1 MSWasm

In MSWasm [24, 69], the new feature is the usage of hardware capabilities as a strategy to obtain memory safety, as displayed in Fig. 1.3.

Memory is a crucial source of vulnerability for many languages. About 70% of all security issues in the Google Chrome browser [15] and in Microsoft products [124] are related to memory safety.

In plain WebAssembly, memory is accessed using 32-bit integers as addresses. Similarly, conventional instruction set architectures like ARM, x86 or RISC-V also

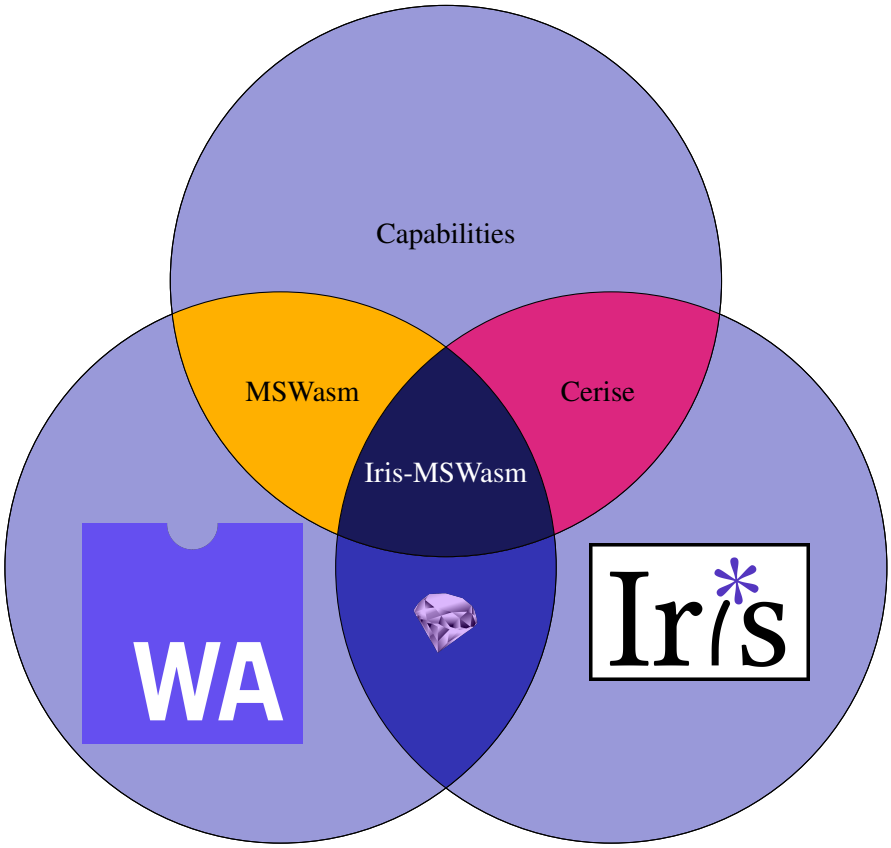


Figure 1.3: Roadmap to Iris-MSWasm

use integers as addresses. This means that all these languages are vulnerable to unsafe pointers being used to access areas of memory that were meant to be off limits. This is particularly unsafe in the case of compiled code, since most compilers implement control flow by defining a call-stack to bookkeep return pointers. Unsafe pointers can use this to hijack control flow, including in WebAssembly [61].

A number of techniques have been proposed [121] at the software level, but none of these completely removes the risks [7, 13, 110]. Georges [32] posits that 'in many ways, they seek to address the symptom, rather than the cause: that modern hardware architectures lack the necessary abstraction to enforce memory safety, and prevent memory corruption attacks.' Language-level solutions [104] have had success in providing memory safety. For instance, Rust provides strong memory safety properties that can be formally reasoned about [51, 53]. However, when compiled to a low-level assembly language, these safety properties do not hold and linking with unsafe assembly code results in all the same vulnerabilities as the target assembly language.

Hence the need for hardware-based solutions. WebAssembly has certainly im-

proved safety by limiting control flow to structured blocks and introducing a type system. Before it, Typed Assembly Language [72] also introduced static type system, and the stronger but more burdensome Proof Carrying Code (PCC) [76] requires code to come together with a proof of safety, both solutions representing static strategies to implement memory safety. Another promising static strategy is given by RichWasm [30], of which we say more in § 1.3.3. On the other hand, capabilities [14] are a hardware-level *dynamic* solution that has proven to be very efficient at guaranteeing memory safety by design; we focus on this solution in the rest of this section.

Capabilities are *unforgeable tokens of authority* [115], meaning they are a distinct type from other primitives like integers, they cannot be created by unprivileged actors, and they carry a certain degree of authority that can be restricted but cannot be increased. In practice, this can take many different meanings at different levels of abstraction. The capabilities we study in this dissertation are hardware capabities, which give access to a specific range of addresses in memory. When attempting to access memory, the semantics dynamically checks that the capability holds enough authority to access the specified address, and safely crashes if not. Here, crashing is considered a safe behaviour because no memory violation has occured when execution is halted.

While the topic of performance is largely orthogonal to the work in this dissertation, we still note that running a program that uses capabilities need not incur a high performance overhead, if the right hardware architecture is used. *Capability machines* like CHERI [129] have been designed to efficiently run capability code, and carefully chosen language design and calling conventions have been shown [32] to formally proven to allow strong memory safety guarantees and efficient performance.

In order to enjoy both WebAssembly's strong static guarantees and the finer-grained safety of capability-based dynamic techniques, Michael et al. [69] propose adding capabilities (which they call *handles*) to WebAssembly, defining Memory-Safe WebAssembly (MSWasm). While the proposal is encouraging, the implementation is incomplete and the safety guarantees are only expressed informally and backed by mere pen-and-paper proofs. In this dissertation, we bring the first formal results validating and improving the model of MSWasm, in the form of a mechanisation of the operational semantics in the Rocq proof assistant, a program logic that allows for manual verification of MSWasm libraries, and a logical relation that formally states and proves the fine-grained robust safety guarantees of MSWasm. We introduce all of these in Chapter 3.

When defining Iris-MSWasm, we take inspiration on existing works that reason about usage of hardware capabilities in Iris, the most important of which is Cerise [33–36]. While the techniques involving reasoning about capabilities are well understood, existing works focus on simplified examples that omit practical details like memory serialisation. The challenge in defining Iris-MSWasm was therefore one of scale: how to adapt the ideas of Cerise to the full industrial size of WebAssembly and the MSWasm proposed extension.

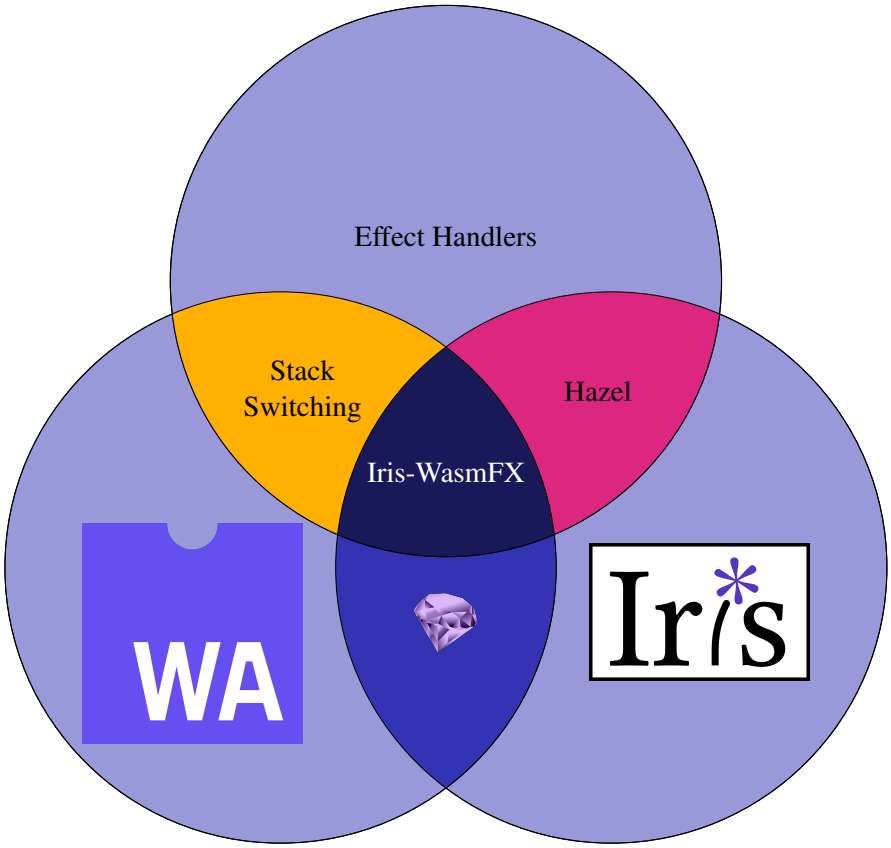


Figure 1.4: Roadmap to Iris-WasmFX

1.3.2 WasmFX and Stack Switching

In stack switching, the new feature is the presence of effect handlers as a new control flow mechanism, as displayed in Fig. 1.4.

Effect Handlers

User-defined computational effects [31] are a popular programming paradigm central in the design of many modern programming languages. These are known under different names depending on the language: call/cc, async/await, coroutines, generators/iterators, effect handlers, futures/promises, etc. These features are often crucial for performance, for example when implementing massively scalable concurrency.

While each version of effect handlers is slightly different [44, §V.A], the general idea is that a point in the program can halt execution and yield control flow to a different point in the program (the *handler*), creating a *continuation* representing the suspended execution. The handler can later decide to resume the continuation, which resumes the execution where the suspension occured. In this sense, an effect can be

viewed as a *resumable* exception. Just like for exceptions, the suspension site can hand over some values (the *payload* of the effect) to the handling site, and the handling site can give values back when resuming the continuation. This complex control flow structure allows for interesting coding patterns that have a good performance in many usecases.

When compiling code from an effectful language to WebAssembly, the absence of non-local control flow operators in WebAssembly means that a full-program transformation is necessary, e.g. in continuation-passing style. This encurs performance overheads and results in a program that cannot be linked with other native WebAssembly modules that have are not implementing the same continuation-passing style conventions. This is at odds with the WebAssembly values of efficiency and modularity, hence the need for new features in WebAssembly to support effect handling languages.

Given the vast array of different styles of effects, implementing all of them in WebAssembly would be at odds with the WebAssembly values of compactness and would make it challenging to achieve universal portability; it would also mean costly and time-consuming adaptation for inevitable future variations required when new non-local control features are introduced.

Instead, the stack switching extension [66] (building on the earlier WasmFX [83], on typed continuations [45, 96, 102], as well as several other projects) proposes a small number of new instructions that can emulate all existing variations of effect handlers and be forwards-compatible with future ones. This effort to be as general as possible, as well as the well-structuredness of WebAssembly programs in general, mean that the presentation of effect handlers in stack switching can look less intuitive to programmers acustomed to higher-level implementation styles; it does however mean that most effectful languages can be seamlessly compiled to stack switching.

The new instructions in stack switching implement *delimited continuations* identified by *named control tags*, inspired by the effect handlers of Plotkin and Pretnar [86, 87]. These effect handlers are well understood in literature [138], integrate well with WebAssembly's syntactic typing system, and can be implemented efficiently [112, 137].

The Hazel Logic

Reasoning about effect handlers is difficult, because of the complex control flow involved. This means that properties that often hold in programs, like well-bracketedness of function calls, no longer hold once effects are present. When defining a program logic for languages with effects, some classic features like the bind rule [52, Fig 13] no longer hold [125], hindering modular reasoning.

The Hazel program logic [18–20] was defined to bridge this gap and propose a way to maintain some degree of modularity when reasoning in the presence of effect handlers. The idea is to define a new *extended* weakest precondition, which adds a *protocol* Ψ to the postcondition Φ:

$$\mathsf{ewp}\, e \,\langle \Psi \rangle\, \{\Phi\}$$

which should be read as 'e runs safely, if it terminates on a value v, then $\Phi(v)$ holds, and if it performs an effect, then protocol Ψ is *followed*'. The role of the protocol is to specify what behaviour is expected when suspending and when resuming a continuation, hence establishing a contract between the suspension site and the handling site. The protocols used in practice are inspired by session types [46] and establish a form of conversation between suspender and handler, by requiring specific resources to be handed up when performing an effect, and specific resources to be handed back down when resuming the continuation. We introduce these in detail in Chapter 4 when we introduce or program logic Iris-WasmFX for stack switching.

When we define Iris-WasmFX, we adapt the ideas of Hazel to the specific case of WebAssembly. Another work that takes inspiration form Hazel is Osiris [17, 106, 107], a program language for OLang, a substantial subset of OCaml 5.3 featuring effect handlers. Given OCaml is designed to be coded in, their effect handlers are more human-readable than in stack switching, where the point is to facilitate compliation. The main challenge in defining Osiris is the loose evaluation order of OCaml, something which by design is not a difficulty in WebAssembly; hence Osiris and Iris-WasmFX face a very different set of challenges. An interesting avenue of future work could be to formally relate programs in OCaml and WebAssembly.

1.3.3 RichWasm

In RichWasm, the new feature is a rich syntactic type system, as displayed in Fig. 1.5.

Given WebAssembly's focus on being a compilation target, there are numerous languages that compile down to WebAssembly, making it an interesting platform for language interoperability. RichWasm [30] is an intermediate language created to facilitate this usage of WebAssembly as a confluence of other languages.

Using plain WebAssembly for this purpose is sometimes tricky for two main reasons. The first is that sharing memories between WebAssembly modules can only be done at a very coarse granularity: the module can share its entire memory, or share nothing. The multiple memories proposal [97] does a little better by allowing several memories to be defined in each module, but even so each memory must either be fully shared or fully private. Another solution is to use capabilities, as is done in MSWasm [24, 69] (see § 1.3.1); this method dynamically ensures memory safety by safely halting execution when a memory violation would occur, and therefore is not always helpful in identifying unsafeness during the typechecking process.

The second reason is the absence of expressive types in WebAssembly: since WebAssembly only defines four base types (**i32**, **i64**, **f32** and **f64**), it can be awkward to reason about exchanging values of higher-level types. Some of these higher-level types like characters, lists, records, variants, etc. are introduced in the interface types proposal [1] and completed in the Component Model porposal [122]. However, these extensions explicitely forbid sharing values and instead rely on time-consuming copying of values at language boundaries to ensure memory safety.

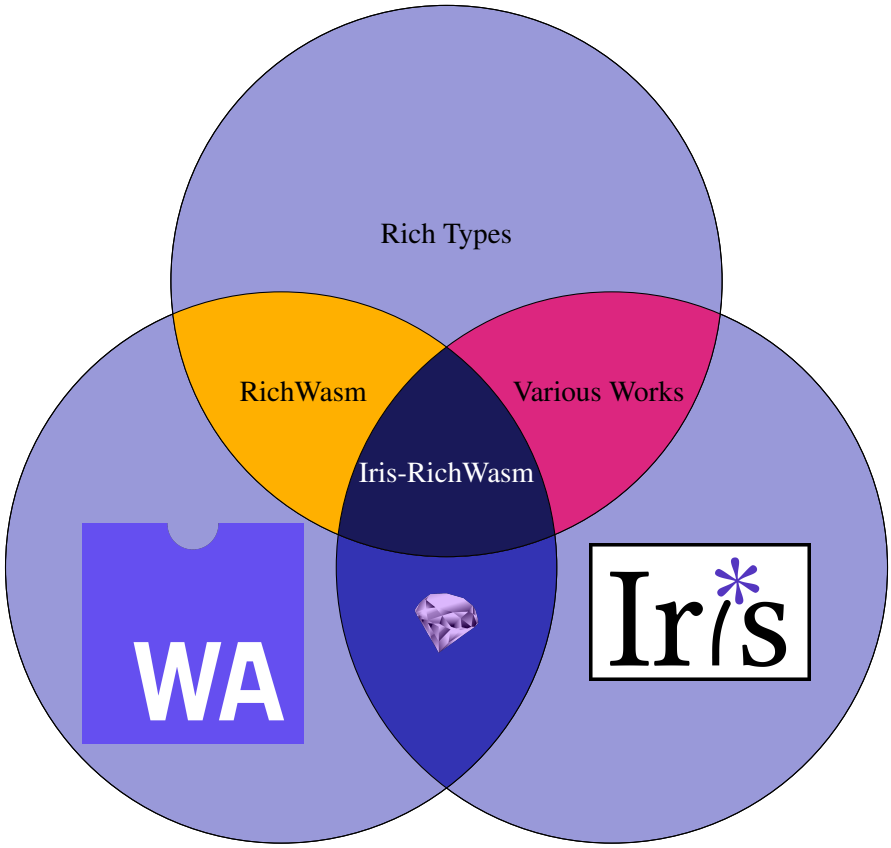


Figure 1.5: Roadmap to Iris-RichWasm

Hence the need for RichWasm. RichWasm can be thought of as a version of WebAssembly with added higher-level types and a richer syntactic type system. This means RichWasm can be compiled to WebAssembly by erasing the types and giving the high-level types a low-level implementation, making RichWasm easy to run. The idea is then that when a programmer wishes to have programs written in different programming languages interact, those programs can all be compiled to RichWasm, where the richer syntactic type system brings stronger memory safety properties.

This is most useful when reasoning about the interaction of garbage-collected language like OCaml and a language that uses a manually managed memory like Rust. In these usecases, passing references languages with conflicting memory management styles can constitute an unsafe information flow, which RichWasm's type system is strong enough to detect and reject safely. The programmer simply need annotate the source code where interaction occurs, to explicitely indicate places where the compiler should use a particular RichWasm type normally reserved for code compiled from the other source language.

A user wishing to make use of these strong guarantees can then write a type-

preserving compiler from the desired source languages to RichWasm, which is much less burdensome than what existing frameworks for safe foreign function interfaces [82] require. Fitzgibbons et al. [30] illustrate this by defining compilers from core ML (which is garbage-collected) and from L^3 [3, 70] (a subset of Rust, which uses a manually managed memory) to RichWasm.

In an exciting joint work in progress, we are helping a team of researchers at Northeastern University to augment Iris-Wasm to reason about the operational semantics of RichWasm. By mechanising a compiler from RichWasm to WebAssembly in the Rocq proof assistant, the team is then able to define a logical relation and prove formally that code output by the compiler always runs safely. In doing so, they have identified areas where RichWasm can be completed with new features like a kinding system.

This project is a good example of the kind of large-level verification projects made possible by Iris-Wasm. When completed, this will have validated and completed the model for RichWasm, provided tools to verify libraries written in RichWasm, and proved formally a strong memory safety result for RichWasm programs, increasing confidence in the language and encouraging its adoption at a wider scale. Since this is a work in progress, we do not include any further details in this dissertation and only include it in this introduction to illustrate other ways in which Iris-Wasm is useful.

1.4 Contributions and Structure

This dissertation consists of the following publications and manuscript, each of which having a corresponding Rocq development:

[89] Rao XiaoJia, Aïna Linn Georges, Maxime Legoupil, Conrad Watt, Jean Pichon-Pharabod, Philippa Gardner, Lars Birkedal
Iris-Wasm: Robust and Modular Verification of WebAssembly Programs
Proceedings of the ACM on Programming Languages (PLDI), 2023
`https://github.com/logsem/iriswasm`

[59] Maxime Legoupil, June Rousseau, Aïna Linn Georges, Jean Pichon-Pharabod, Lars Birkedal
Iris-MSWasm: Elucidating and Mechanising the Security Invariants of Memory-Safe WebAssembly
Proceedings of the ACM on Programming Languages (OOPSLA), 2024
`https://github.com/logsem/MSWasm`

[58] Maxime Legoupil, Jean Pichon-Pharabod, Sam Lindley, Lars Birkedal
Iris-WasmFX: Modular Reasoning for Wasm Stack Switching
To be submitted to PLDI 2026
`https://github.com/logsem/iris-wasmfx`

Each of these also corresponds to one key contribution of this dissertation:

Iris-Wasm: we provide a program logic and logical relation for the full semantics of WebAssembly 1.0, allowing for verification of WebAssembly libraries and formally stating and proving the crucial local state encapsulation property

Iris-MSWasm: we provide a mechanisation of MSWasm together with a program logic and logical relation for its full operational semantics, validating and completing the MSWasm proposal, creating a tool for verification of MSWasm libraries and formally stating and proving the stronger robust capability safety property

Iris-WasmFX: we provide a mechanisation of stack switching and a program logic for the key parts of its operational semantics, fully validating the model and completing its syntactic type system, establishing the first result for type safety, and creating a tool for verification of stack switching libraries

Within these works, my personal contributions have been the following:

- Iris-Wasm:
  - Establishing many helper lemmas about the operational semantics of WebAssembly
  - Defining the logical values in the program logic, in particular the break value `brV` which necessitated defining a dependent type to syntactically capture an environment being shallow enough for a break instruction to be stuck (see § 2.2.2)
  - Defining and proving select proof rules in the logical relation
  - Designing and verifying the stack module, which served as the running example in the paper
  - Substantial contributions in writing the paper
- Iris-MSWasm:
  - Extending the WasmCert mechanisation of WebAssembly to the proposed MSWasm extension
  - Identifying and correcting bugs in the original presentation of the proposal
  - Adding proof rules for all new instructions of MSWasm

  - Extending the logical relation with the new type and the new instructions of MSWasm
  - Designing and verifying all examples showcasing the new additions of MSWasm
  - Substantial contributions in writing the paper

- Iris-WasmFX:
  - Extending the WasmCert mechanisation of WebAssembly 1.0 with select instructions from WebAssembly 2.0, the exception handling suite, and the new instructions of stack switching
  - Identifying and correcting a bug in the proposal, and engaging with the group writing the proposal to get the change adopted
  - Defining syntactic typing rules for the runtime elements of stack switching
  - Proving type safety for stack switching
  - Defining our custom extended weakest precondition statement to define the program logic
  - Adding proof rules for key instructions of stack switching
  - Designing and verifying examples to showcase Iris-WasmFX
  - Substantial contributions in writing the paper

Part II

Publications

Chapter 2

Iris-Wasm: Robust and Modular Verification of WebAssembly Programs

This chapter consists of the following paper:

Rao XiaoJia, Aïna Linn Georges, Maxime Legoupil, Conrad Watt, Jean Pichon-Pharabod, Philippa Gardner, Lars Birkedal
Iris-Wasm: Robust and Modular Verification of WebAssembly Programs
Proceedings of the ACM on Programming Languages (PLDI), 2023

A few minor adjustments have been done to the article:

- The layout has been reformatted to fit this document's page size
- The headings have been recapitalised to fit ACM guidelines
- The appendices have been reorganised

Abstract

WebAssembly makes it possible to run C/C++ applications on the web with near-native performance. A WebAssembly program is expressed as a collection of higher-order ML-like modules, which are composed together through a system of explicit imports and exports using a host language, enabling a form of higher-order modular programming. We present Iris-Wasm, a mechanized higher-order separation logic building on a specification of Wasm 1.0 mechanized in Coq and the Iris framework. Using Iris-Wasm, we are able to specify and verify individual modules separately, and then compose them modularly in a simple host language featuring the core operations of the WebAssembly JavaScript Interface. Building on Iris-Wasm, we develop a logical relation that enforces robust safety: unknown, adversarial code can only affect other modules through the functions that they explicitly export. Together, the program logic and the logical relation allow

us to formally verify functional correctness of WebAssembly programs, even when they invoke and are invoked by unknown code, thereby demonstrating that WebAssembly enforces strong isolation between modules.

2.1 Introduction

WebAssembly (Wasm) is a new bytecode language, supported by all major Web browsers and designed primarily to be an efficient compilation target for low-level languages such as C/C++ and Rust. It is officially specified using a formal operational semantics in the W3C Wasm 1.0 standard [95]. The formal nature of the official Wasm standard and the existence of a well-exercized language mechanization give us a standout opportunity to define a higher-order program logic that covers the full definition of an industrial programming language. We introduce Iris-Wasm, a mechanized higher-order separation logic for Wasm 1.0 which builds on the WasmCert-Coq mechanized specification of the Wasm 1.0 language standard [133] and the Iris framework [49, 52]. In Iris-Wasm, we present an interactive formal verification framework that exactly reflects the Wasm semantics. The result is a semantic and compositional characterization of all Wasm definitions, which can be used to prove separation logic assertions about real Wasm programs, and which lays the foundation for rigorous investigations of the Wasm ecosystem.

A Wasm program is expressed as a collection of higher-order ML-like *modules*, which are composed together through a system of explicit imports and exports. This process of composing Wasm modules into a full program is not performed within Wasm itself. Instead, Wasm is embedded within a *host language*, which provides several important capabilities not available to core Wasm code, including a complex, inherently higher-order, *instantiation* operation in which the declared state of a WebAssembly module is allocated, the module's requested imports are satisfied, and the module's declared exports are registered for use in satisfying further imports requested during subsequent instantiations. The Wasm standard defines instantiation in a host-agnostic way, to be then satisfied by the specific host-language instantiation. For example, a typical Wasm program on the web will involve individual Wasm modules which are instantiated and composed together by a top-level JavaScript host script using the functions of the WebAssembly JavaScript Interface [25].

Iris-Wasm is a higher-order mechanized program logic for the W3C Wasm 1.0 industrial standard using the Iris framework, inspired by a previous Isabelle-mechanized first-order program logic for the language draft [132]. Our implementation of the Wasm run-time semantics, with its difficult constructs such as complex control-flow commands, is given directly in Iris, instead of being translated into an existing intermediate Iris language. This choice requires considerable Iris engineering, but provides more trust in our mechanization, as it is line-by-line close to the Wasm semantics, and should lead to the mechanization being comparatively straightforward to extend as the standard expands. We make a minor reformulation of the host function semantics (see §2.2.2), so that our core Wasm semantics and program logic are properly separate

from the host.

We provide a host-agnostic axiomatic characterization of Wasm module instantiation by establishing a lemma which lifts the complex W3C Wasm 1.0 instantiation predicate to our Iris-Wasm logic, describing the state before and after instantiation using our logical assertions. We illustrate this instantiation lemma on a simple host language designed to capture the core functionality of the WebAssembly JavaScript Interface [25], and corresponding host program logic, where the soundness of our host instantiation proof rule is established using our instantiation lemma. The Iris-Wasm program logic thus gives a semantic characterization of the host-agnostic instantiation operation.

By capturing the semantics of the full Wasm 1.0 industrial standard directly, Iris-Wasm lays the groundwork for a wide range of future analyses. Iris-Wasm can be used to validate proposed extensions to Wasm such as MSWasm, a memory safe extension of Wasm [69]. It can be used to rigorously investigate compilers that either target Wasm or compile Wasm down to some low-level assembly language. Jacobs et al. [48] demonstrate that Iris can be a useful tool to prove results such as full abstraction. Iris-Wasm sets the groundwork for similar results for realistic compilers involving Wasm.

We demonstrate our compositional higher-order reasoning about Wasm modules in our host language by developing a series of examples. Our main running example is a higher-order stack example comprising a stack module and a client module. The stack module defines and exports stack functions, including a higher-order map function for the stack. The client module imports and uses some of them, including map, in its main function. Using our Wasm program logic and a program logic for the simple host we introduce, we provide specifications for both modules: the stack module's specification contains specifications for all the stack functions, and the client module's specification depends on the stack module's specification. Finally, we verify a host program which instantiates the two modules in sequence, by modularly combining the proofs for the two module specifications. In addition, we demonstrate how to reason about *reentrancy* between the host and Wasm, by having the client module invoke a host function to modify the function table to provide a different input function for subsequent applications of map. The higher-order reasoning of the Iris framework provides an ideal environment to reason about Wasm modules. Nevertheless, it's a substantial task to apply Iris to a true industrial standard. Our implementation precisely follows the design decisions of the W3C Wasm 1.0 standard, and by using a rich logic such as Iris, we have laid the foundations for deep semantic investigations of WebAssembly and its future iterations.

In a case study, we investigate the intuitive coarse-grained encapsulation property of Wasm modules, stated in the standard: 'code from a module can arbitrarily affect its own state, but can only access the state of another module through the module's exports'. Several systems rely on this important property of Wasm to provide a form of sandboxing: for example, Fastly's 'Compute@Edge' [43] platform and the RLBox tool [75]. Both depend on the encapsulation property of a module, regardless of behaviour of other modules, which are validated but not necessarily trusted. Reasoning

about such modules necessarily involves the interaction between the known, verified code of one module against unknown, untrusted, and unverified code from other modules, something that cannot be done with a program logic. Building on top of Iris-Wasm, we define a relational interpretation of WebAssembly types through a unary logical relation, which is then used to verify specific *robust safety* properties of a known module, that hold even when composed with unknown modules. We demonstrate this by proving robust safety properties of our stack module composed with arbitrary clients. Our relational interpretation is entirely host agnostic, and can modularly be applied to any host language.

In summary, our contributions are:

1. Iris-Wasm, a Coq-mechanized higher-order program logic for the Wasm runtime semantics.

2. A host-agnostic module instantiation lemma, and a program logic for a simple example host language with the specific host instantiation rule proved using our general instantiation lemma.

3. A semantic interpretation of the Wasm type system, defined via a logical relations interpretation using our Wasm program logic.

4. Illustrative examples and case studies that demonstrate the expressiveness of Iris-Wasm; we show that an implementation of a higher-order stack module satisfies a very modular abstract specification; we verify a reentrant module that uses host language features to modify function tables dynamically; and we use Iris-Wasm to define and prove the properties of our logical relation, which we use to verify robust safety of higher-order examples.

All results, including soundness of the program logic and logical relations, are formalized in Coq. We hope this will prove useful to other researchers for further investigating the Wasm ecosystem.

2.1.1 Higher-Order Programming in WebAssembly and Reentrancy

Consider the WebAssembly snippet in Figure 2.1, which contains a module that works as a library implementing a stack of **i32**s (on the left), and a module that works as a client of that library (on the right). The library module, which the host language calls `"stack"` here, uses a **memory** (with initial size 0; some other function is in charge of allocating space for the stack) to implement a stack. The `"stack"` module exports a `"map"` function that maps a function over a stack. However, because WebAssembly is a first-order language, `"map"` does not take the function to map as an argument. Instead, `"map"` takes as argument an index, `$i`, into a table of 3 functions, `"tab1"`, that this module creates and exports, and calls the function at that index in the table using `call_indirect`. The client module imports the same shared table of functions, and uses the **elem** directive to populate it (from offset 0) with functions it defines: `$f0`, `$f1`, and `$f2`. It also imports the `"map"` function from the `"stack"` module as `$map`, and

```
stack_module ≜
(module ;;"stack"
  (type $t1 (func (param i32) (result i32)))
  (table (export "tab1") 3 funcref)
  (memory 0)
  (func (export "map")
        (param $i i32) (param $stk i32)
   ...
   loop
    ...
    local.get $i
    call_indirect  $t1
    ...
   end ...)
```

```
client_module ≜
(module ;;"client"
  (import "stack" "tab1" (table 3 funcref))
  (import "stack" "map"
   (func $map (param i32 i32)))
  (elem (i32.const 0) $f0 $f1 $f2)
  (func $f0 (param $n i32) (result i32)
   ...)
  (func (export "main")
        (param $stk i32) (result i32)
   i32.const 0
   local.get $stk
   call $map
   ... ;;Rest of the code))
```

Figure 2.1: A module implementing a stack library, and a client module. Module boundaries enforce isolation.
This example uses the Wasm text format; below, we work directly with the AST.

its "main" function then calls the $map function with function index 0 as argument, which makes it map $f0 on the stack.

In §2.2 we describe our program logic and we show in §2.2.2 how it can be used to give a modular specification of the stack module, and, in particular, in §2.2.3, the "map" function. A proof of the specification of the *instantiation* of the stack module is given at the end of §2.3. We emphasize that our logic supports verification of the client module relative to an abstract logical specification of the stack module; in other words, the encapsulation of the internal representation of the stack module is reflected in its specification.

We now consider a simple extension of this example to demonstrate the need for reasoning about reentrancy between WebAssembly and the host. To this end, we will let the "main" function, after the call to $map, dynamically modify the contents of the table to now contain a new function $f3 at index 0. Dynamic modification of the table cannot be performed in WebAssembly 1.0, as WebAssembly only has the **elem** directive available to statically provide an initial value for the elements of the table. WebAssembly code can, however, call functions defined by the host, and those may modify the state of the WebAssembly program. Thus we add an import (**import** "host" "mut" (func $mut (**param i32 i32**))) to the preamble of the client module and then complete the code of the "main" function with 6 more instructions: **i32.const** 0; **i32.const** $f3; call $mut; **i32.const** 0; local.get $stk; call $map. The first three of these call the host function $mut that we assume will modify the function table at address 0, replacing the previous value ($f0) by $f3. The last three instructions are a call to $map identical to the one at the beginning of the body of "main" function (see Figure 2.1), but this time, when mapping the 0th function from the table onto the stack, it maps function $f3 instead of $f0 like it did during the first call to $map. Thus calling "main" on a value that represents stack $[x_0, \ldots, x_n]$ will modify the stack so that the argument value now represents $[f_3(f_0(x_0)), \ldots, f_3(f_0(x_n))]$.

This example illustrates how programs may take advantage of the stronger expressive power of the host. In §2.2.2, we show how we deal with calls to host functions in Iris-Wasm, and in §2.3, we introduce a simple host language and a program logic for it and show how it can be used in combination with our WebAssembly program logic to reason about complex interaction between WebAssembly code and the host language code that embeds it, including this example.

2.2 Modular Reasoning for WebAssembly Modules

In this section, we introduce Iris-Wasm. We present our proof rules for WebAssembly language features, and outline how they are used to prove a specification for the stack module from the Introduction. For reasons of space, we only discuss selected proof rules; we stress that we have proved program logic rules for *all* of WebAssembly and used them to give full formal proofs of examples, including the stack module; see the accompanying Coq formalization for details. Then, in §2.3, we present the operational semantics and proof rules for our host language, and show how they are used to verify the interaction of a client module with the stack module; we focus on instantiation and reentrancy. Finally, in §2.4 we discuss how our program logic is defined within the Iris program logic framework, we overview some of the generic features and proof rules we inherit from Iris, and we state the soundness and adequacy of Iris-Wasm.

2.2.1 Proof rules for basic WebAssembly stack operations

WebAssembly is a stack language with structured control. Its dynamics is specified by a small-step operational semantics on *configuration tuples* of the form $(S;F;es)$, where es is a hybrid stack of values and instructions,[1] S is the global *store*, and F is the current function *frame*. The store S contains information about the global variables, the tables, the memories and the functions declared in all modules instantiated thus far, and the frame F contains the values of all local variables, as well as an *instance* that handles indirection, as will be explained progressively below. We recall the abstract syntax in Figure 2.2.

Reductions are structural: for any program fragment[2] es that reduces to es', the same reduction can occur under a context; for example, for any list vs of constants and es_2 of expressions, $vs +\!\!+ es +\!\!+ es_2$ reduces to $vs +\!\!+ es' +\!\!+ es_2$. We give the general meaning of contexts in §2.2.2.

The overall structure of the operational semantics is as expected for a stack language; for example, the stack $[t.\mathbf{const}\ c_1; t.\mathbf{const}\ c_2; t.\mathbf{binop}\ binop]$ reduces to $[t.\mathbf{const}\ c]$, where c is the result of applying $binop$ to c_1 and c_2. Let us introduce the corresponding proof rule in our program logic.

[1] The standard uses '*' to stand for 'a list of', but we prefer using s as a suffix to avoid confusion with the symbol for separating conjuction, so 'es' is a list of 'e's, 'vs' is a list of 'v's, etc.

[2] For simplicity, in this paper, we conflate what WebAssembly calls 'basic instructions' and 'administrative instructions'; see beginning of §2.2.2.

$$
\begin{array}{rl}
\text{(value type) } t & ::= \mathbf{i32} \mid \mathbf{i64} \mid \mathbf{f32} \mid \mathbf{f64} \\
\text{(value) } v & ::= t.\texttt{const}\ c
\end{array}
$$

$$
\begin{array}{rl}
\text{(function type) } ft & ::= ts \to ts \\
\text{(immediate) } i, min, max & ::= nat
\end{array}
$$

$$
\begin{array}{rl}
\text{(instructions) } e & ::= v \mid t.\texttt{add} \mid \mathit{other\ stackops} \mid \texttt{local.}\{\texttt{get/set}\}\ i \mid \texttt{global.}\{\texttt{get/set}\}\ i \mid \\
& \quad t.\texttt{load}\ \mathit{flags} \mid t.\texttt{store}\ \mathit{flags} \mid \texttt{memory.size} \mid \texttt{memory.grow} \mid \texttt{block}\ ft\ es \mid \\
& \quad \texttt{loop}\ ft\ es \mid \texttt{if}\ ft\ es\ es \mid \texttt{br}\ i \mid \texttt{br\_if}\ i \mid \texttt{br\_table}\ is \mid \texttt{call}\ i \mid \\
& \quad \texttt{call\_indirect}\ i \mid \texttt{return}
\end{array}
$$

$$
\begin{array}{rl}
\text{(functions) } \mathit{func} & ::= \texttt{func}\ i\ ts\ es \\
\text{(memories) } \mathit{mem} & ::= \texttt{mem}\ min\ max \\
\text{(elem segments) } \mathit{elem} & ::= \texttt{elem}\ i\ es_{\mathsf{off}}\ is
\end{array}
$$

$$
\begin{array}{rl}
\text{(tables) } \mathit{tab} & ::= \texttt{tab}\ min\ max \\
\text{(globals) } \mathit{glob} & ::= \texttt{glob}\ \mathsf{mutable}\ t\ \ e_{\mathsf{init}} \\
\text{(data segments) } \mathit{data} & ::= \texttt{data}\ i\ es_{\mathsf{off}}\ bytes
\end{array}
$$

$$
\begin{array}{rl}
\text{(import descriptions) } \mathit{importdesc} & ::= \texttt{func}_{\mathsf{i}}\ i \mid \texttt{tab}_{\mathsf{i}}\ min\ max \mid \texttt{mem}_{\mathsf{i}}\ min\ max \mid \\
& \quad \texttt{glob}_{\mathsf{i}}\ \mathsf{mutable}^{?}\ t \\
\text{(imports) } \mathit{import} & ::= \texttt{import}\ \mathit{string}\ \mathit{string}\ \mathit{importdesc} \\
\text{(export descriptions) } \mathit{exportdesc} & ::= \texttt{func}_{\mathsf{e}}\ i \mid \texttt{tab}_{\mathsf{e}}\ i \mid \texttt{mem}_{\mathsf{e}}\ i \mid \texttt{glob}_{\mathsf{e}}\ i \\
\text{(exports) } \mathit{export} & ::= \texttt{export}\ \mathit{string}\ \mathit{exportdesc} \\
\text{(start) } \mathit{start} & ::= \texttt{Some}\ i \mid \texttt{None}
\end{array}
$$

$$
\begin{array}{rl}
\text{(function instances) } \mathit{finst} & ::= \{(\mathit{inst}; ts); es\}^{\texttt{NativeCl}}_{tf} \mid \{\mathit{hidx}\}^{\texttt{HostCl}}_{tf} \\
\text{(table instances) } \mathit{tinst} & ::= \{\mathsf{elem} : is,\ \ \mathsf{max} : max^{?}\} \\
\text{(memory instance) } \mathit{minst} & ::= \{\mathsf{data} : bytes,\ \mathsf{max} : max^{?}\} \\
\text{(global instance) } \mathit{ginst} & ::= \{\mathsf{mut} : mutable^{?},\ \mathsf{value} : v\} \\
\text{(store) } S & ::= \{\ \mathsf{funcs} : \mathit{finsts},\ \mathsf{globs} : \mathit{ginsts}, \\
& \qquad \mathsf{mems} : \mathit{minsts},\ \mathsf{tabs} : \mathit{tinsts}\ \} \\
\text{(frame) } F & ::= \{\ \mathsf{locs} : vs,\ \ \mathsf{inst} : \mathit{inst}\ \} \\
\text{(module instance) } \mathit{inst} & ::= \{\ \mathsf{types} : \mathit{fts},\ \mathsf{funcs} : is,\ \mathsf{globs} : is, \\
& \qquad \mathsf{mems} : is,\ \mathsf{tabs} : is\ \}
\end{array}
$$

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

Figure 2.2: WebAssembly 1.0 Abstract Syntax

Weakest Preconditions Our proof rules are phrased using Iris' *weakest precondition*. Intuitively, $\mathsf{wp}\, es\, \{w, \Phi(w)\}$ states that the program fragment *es* computes safely, and, if it terminates with result w, predicate Φ holds of w (we discuss the formal meta-theory in §2.4). This construct is close to Hoare triples, as we have the following equality in Iris[3]:

$$\{P\}\ es\ \{w, \Phi(w)\} = \Box(P \mathrel{-\!\!*} \mathsf{wp}\, es\, \{w, \Phi(w)\})$$

Logical Values Because we reason about *fragments* of WebAssembly programs, execution does not always terminate with a stack of WebAssembly values, but more generally with a *logical value*:

$$\begin{array}{rcl} \mathit{LogVal} \ni w & ::= & \texttt{immV}\ \mathit{vs} \mid \texttt{trapV} \mid \texttt{brV}\ i\ \mathit{vh}_i \mid \texttt{retV}\ \mathit{lh}_k \mid \\ & & \texttt{call\_hostV}\ \mathit{tf}\ \mathit{hidx}\ \mathit{vs}\ \mathit{llh} \end{array}$$

which is one of the following:

- `immV` *vs*, the 'normal' result: a stack of WebAssembly values;
- a trap `trapV`, which represents that the program has encountered an error in its execution;
- a break (or branching) value `brV`, a return value `retV`, or a host call value `call_hostV`, which correspond to program fragments that are stuck as such, but can get unstuck when placed in an appropriate context; we explain their meaning, and the meaning of their arguments, in §2.2.2.

Accordingly, in our proof rules, the postcondition Φ takes a logical value w as an argument.

Proof Rule We prove the following Iris-Wasm proof rule for binary operators:

$$\texttt{wp\_binop}\quad \frac{[\![t.\mathit{binop}]\!](c_1, c_2) = c \ \ast\ \triangleright \Phi(\texttt{immV}\,[t.\textbf{const}\ c]) \ \ast\ \hookleftarrow^{\text{FR}} F}{\mathsf{wp}\,[t.\textbf{const}\ c_1; t.\textbf{const}\ c_2; t.\textbf{binop}\ \mathit{binop}]\left\{w, \Phi(w) \ast \hookleftarrow^{\text{FR}} F\right\}}$$

which states that, with two constants $t.\textbf{const}\ c_1$ and $t.\textbf{const}\ c_2$ on the value stack, and any function frame F, if an arbitrary predicate Φ holds *later* of the result c of the binop of type t on c_1 and c_2, then this program fragment executes safely, and if it terminates (which it does in this case), Φ holds of the execution result w, because it will be the value stack `immV` $[t.\textbf{const}\ c]$. The frame resource is a special resource which needs to be included in every proof rule where we 'take a reduction step'.

We merely require that Φ holds after one step of execution, as expressed by the later $\triangleright$ modality of Iris [52]. One may choose to ignore this, but it is necessary in the presence of Iris' higher-order features, to avoid cyclicity.

[3]The persistent modality $\Box$ indicates that the Hoare triple is a proposition that can be duplicated as many times as needed.

2.2.2 Control and function calls

Control and function calls in WebAssembly are intricate, but still feature locality, as expected; for example, blocks can be reasoned about in isolation, and function scope is still respected. We present an approach that allows us to reason about code fragments without needing knowledge of their environment; it improves over the approach taken in the earlier Wasm program logic [132] which does not scale to higher-order programs. In this section, we show how our rules capture this locality to make reasoning tractable.

Administrative Instructions

To define reduction of blocks and functions calls, WebAssembly adds an extra layer on top of the surface language, to represent intermediate states by *administrative instructions*, which are defined by the following grammar:

$$\begin{array}{rcl} AI & ::= & \texttt{basic}\ e \mid \texttt{trap} \mid \texttt{invoke}\ i \mid \texttt{label}_i\{es\}\ es\ \texttt{end} \mid \\ & & \texttt{local}_i\{F\}\ es\ \texttt{end} \mid \texttt{call\_host}\ tf\ hidx\ vs \end{array}$$

- A `basic` instruction is a plain WebAssembly expression, as described in Figure 2.2. When clear from the context, we conflate `basic` e and e, for example in weakest preconditions.
- A `trap` represents a program that has encountered an error in its dynamic execution.
- An `invoke` represents an intermediate step when reducing a `call` or `call_indirect` .
- A `label` represents a block or a loop that is being executed.
- A `local` represents a function call that is being executed.
- A `call_host` represents a program that performs a call to a function defined the host language.

We discuss the last four kinds of administrative instructions below, as we describe control flow and function calls in WebAssembly.

Blocks, Labels, and Breaks

WebAssembly is somewhat unusual as an assembly-like language in that it features only structured control, including labeled breaks. We show how we use the higher-order nature of Iris to ease reasoning about the control structure of WebAssembly.

WebAssembly has (aside from function calls) two core constructs for control flow: `block`, and `loop` (and the conditional **if**, which reduces immediately to a `block`). These take as arguments a function type, and a list of expressions constituting the body of the `block` or `loop`. This body will reduce until either it becomes a list of constants

and the `block` or `loop` is exited, or a `br` instruction is its first non-constant instruction. In a `block`, the body is then exited, and execution continues with whatever follows the block; and in a `loop`, the full original body of the `loop` is repeated from the beginning. The function type $ts_1 \to ts_2$ describes the $|ts_1|$ values[4] needed to enter the `block` or `loop`, and the $|ts_2|$ values that need to be on the stack if a `br` is encountered.

Because of the similarity between these two constructs, the WebAssembly semantics has them both reduce to a `label` administrative instruction. $\texttt{label}_n\{es_{\text{cont}}\}\ es_{\text{body}}\ \texttt{end}$ is a label with body es_{body} that will execute continuation expression es_{cont} if it encounters a `br` instruction preceeded by n values. We come back later to the exact semantics of `br`. When preceeded with $|ts_1|$ values vs of the right type, $\texttt{block}\ (ts_1 \to ts_2)\ es$ reduces to $\texttt{label}_{|ts_2|}\{[]\}\ vs +\!\!+\ es\ \texttt{end}$ and $\texttt{loop}\ (ts_1 \to ts_2)\ es$ reduces to $\texttt{label}_{|ts_1|}\{[\texttt{loop}\ (ts_1 \to ts_2)\ es]\}\ vs +\!\!+\ es\ \texttt{end}$.

Once the `block` or `loop` instruction has been reduced to a `label`, reduction steps can be taken in the body of the `label`. As this may happen under many nested labels, WebAssembly defines evaluation contexts lh_k, which describe stack environments consisting of k nested *labels* surrounding a *hole* $[\_]$ where the next step of execution takes place:

$$lh_0 ::= vs +\!\!+\ [\_] +\!\!+\ es \qquad lh_{k+1} ::= vs +\!\!+\ \texttt{label}_n\{es_{cont}\}\ lh_k\ \texttt{end} +\!\!+\ es$$

Note how only (constant) values vs can be on the left of the hole and label instructions: this enforces that we can only 'zoom in' on the next expression to reduce.

As expected, steps can be taken under an evaluation context: if es reduces to es', then $lh_k[es]$ reduces to $lh_k[es']$. Taking $k = 0$ yields the expected sequencing rule mentioned at the start of §2.2.1.

Correspondingly, we prove the following Iris-Wasm rule, which reduces reasoning about a program fragment that can be decomposed as $lh_i[es]$ to reasoning about $lh_i[vs]$, that is, the result vs of evaluating the expression to a list of constants, placed in the evaluation context.[5]

$$\texttt{wp\_ctx\_bind}\quad \frac{\mathsf{wp}\, es\, \{w, \mathsf{wp}\, lh_i[w]\, \{w', \Phi(w')\}\}}{\mathsf{wp}\, lh_i[es]\, \{w', \Phi(w')\}}$$

This rule leverages the fact that in Iris, weakest preconditions are propositions themselves, and can therefore be nested. Notice how we have implicitly cast w, a logical value, into an expression when plugging it into lh_i. This is done in the intuitive way: `immV` vs is cast into vs, `trapV` is cast into the single administrative instruction [`trap`], etc.

While control flow in WebAssembly is structured, the presence of labelled breaks makes it slightly involved. A break targets a particular level of the evaluation context,

[4] In WebAssembly 1.0, ts_1 is always empty.

[5] The version we show here is meant for evaluation contexts with at least one label constructor; in our Coq formalization, we prove more intricate variations of this rule, to be applied for sequencing, with for instance $lh_i[es]$ replaced with $lh_i[es_1 +\!\!+\ es_2]$.

and skips the rest. As a result, the default evaluation context rules provided by Iris are inadequate, and we have to build our own reasoning principles for contexts.

The `br` i instruction targets the i^{th} label from the context. Crucially, breaking relies on the instruction `br` i being in an evaluation context lh_k with $i = k$: the break index indicates what context depth is targeted. If $i > k$, the expression $lh_k[\texttt{br}\ i]$ is stuck and can only reduce if placed in a deeper context. Correspondingly, we introduce a new type of logical values: `brV` i vh_i, representing the program fragment $vh_i[\texttt{br}\ i]$. The *breaking context* vh_i is similar to an evaluation context lh_i, except that the meaning of the subscript i is that the context has depth *at most* i, instead of exactly i. If $i < k$, a `br` i nested in context lh_k will only break out of the i first `labels`, and the result will be in the form $lh_{k-i}[vs +\!\!+\ es]$. The break value `brV` allows to bind into any number of `labels` without needing to worry about getting stuck at a `br` i statement: when encountering such a statement, we simply bind back $i+1$ times to get a wp in a form where our rule for `br` can be applied.

Functions

There are two ways to call a function in WebAssembly: statically with `call`, or by dynamically fetching a function from a table, with `call_indirect`. We focus on the simpler direct `call` here, and explain `call_indirect` in §2.2.3.

The instruction `call` n calls the nth function declared in the current module. Indexing starts at 0 with the imported functions, followed by the functions defined in the module itself. The store S keeps a list of the *function closures* (which we describe below) of *all* the instantiated modules. This means the nth function in the current module will not always be the nth function in the store: the *instance* in the function frame F is in charge of remembering that indirection. The instance also contains this indirection information for global variables, memories, and tables.

A `call` i retrieves the address *addri* of the relevant closure in the store from the frame's instance, and reduces to `invoke` *addri*. We prove the corresponding Iris-Wasm rule:

$$\frac{(F.\mathsf{inst.funcs}[i] = addri) * \xhookrightarrow{\text{FR}} F * \triangleright \left(\xhookrightarrow{\text{FR}} F \mathrel{-\!\!*} \mathsf{wp}\,[\texttt{invoke}\ addri]\,\{w, \Phi(w)\}\right)}{\mathsf{wp}\,[\texttt{call}\ i]\,\{w, \Phi(w)\}}\ \texttt{wp\_call}$$

which requires ownership of the frame, not only because we are taking a reduction step, but also to know where to look up index *addri*.

The function closures *cl* (also called function instances *finst* in Figure 2.2) stored in the store S are of two kinds: native and host. Let us focus first on native closures, and come back to host closures at the end of this section. The closure $\{(inst; ts); es\}^{\texttt{NativeCl}}_{ts_1 \to ts_2}$ describes a *native* function that was defined in a WebAssembly module with instance *inst* (this is the environment for the closure), which expects arguments of type ts_1, defines additional local variables of type ts for the computation of its body, yields results of type ts_2, and has body es. When reducing `invoke`, we look up the closure in the store, and check that the stack contains the appropriate

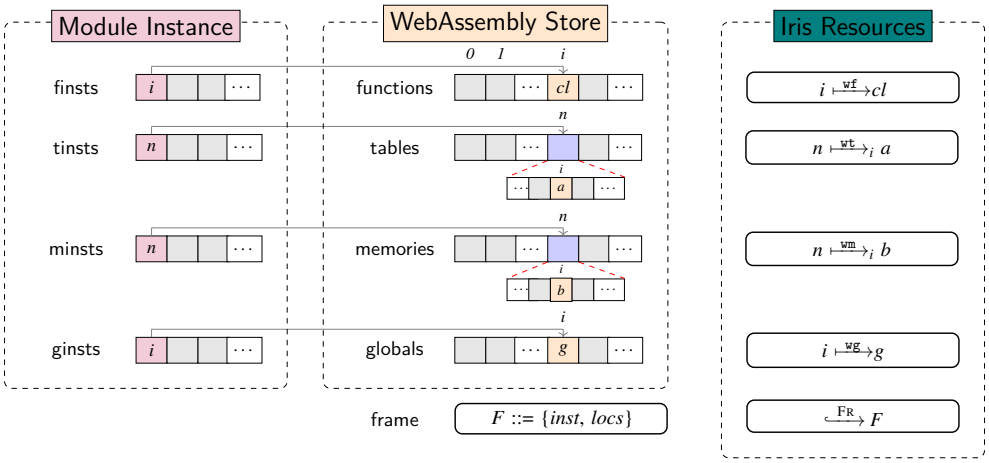


Figure 2.3: Points-to predicates for the store and the frame

number of values to be passed as parameters to the function. If the closure is native, `invoke` is replaced with the body of the function. In order to properly encapsulate the function call, WebAssembly places the function body inside a `local` administrative instruction, and inside a `block`, as captured by the following Iris-Wasm proof rule (we say more about `local` and the meaning of F' further down):

`wp_invoke_native`

$$\frac{|vs| = |ts_1| * cl = \{(inst; ts); es\}^{\texttt{NativeCl}}_{(ts_1 \to ts_2)} * F' = \{\mathsf{locs} := vs +\!\!+ \texttt{zeros}(ts); \mathsf{inst} := inst\} * i \overset{\texttt{wf}}{\mapsto} cl * \overset{\text{FR}}{\hookleftarrow} F * \triangleright \left[\begin{array}{c} (i \overset{\texttt{wf}}{\mapsto} cl * \overset{\text{FR}}{\hookleftarrow} F) \mathrel{-\!\!*} \\ \mathsf{wp}\ \left[\texttt{local}_{|ts_2|}\{F'\}\ (\texttt{block}\ ([] \to ts_2)\ es)\ \texttt{end}\right]\ \{w, \Phi(w)\} \end{array}\right]}{\mathsf{wp}\ (vs +\!\!+ \texttt{invoke}\ i)\ \{w, \Phi(w)\}}$$

Unlike for the function frame F, we do not assert ownership of the whole store S. Instead, we rely on points-to predicates to assert ownership of specific components: for instance, the predicate $i \overset{\texttt{wf}}{\mapsto} cl$ asserts ownership of $S.\mathsf{funcs}[i]$ in the store.

In general, we define points-to predicates for each component of the Wasm store. Fig. 2.3 illustrates all the points-to predicates used in this paper, and how they relate to the physical Wasm store. Functions and globals are referred to directly via their indices, while function tables and linear memories can be viewed as two dimensional structures, where an index is used to refer to a particular table or memory, and another index is used to refer to a particular cell within that table or memory. For example, $n \overset{\texttt{wm}}{\mapsto}_i b$ asserts that the i^{th} byte of memory n is b. The WebAssembly frame F tracks the scope of the currently executing function, namely its enclosing instance and local variables. The enclosing instance collects indices of all the entities of the Wasm store that the module may access, and is crucial for enforcing the encapsulation properties of Wasm modules.

Encapsulation Let us return to why the function body is placed inside a `local` and inside a `block`. The first of these is to provide proper encapsulation, as reduction of

an expression nested in a `local` takes place with respect to the nested frame of the `local`: when reducing $[\texttt{local}_n\{F_1\}\ es\ \texttt{end}]$, one reduces es with respect to frame F_1 rather than the current function frame F.

For our native invocation, the frame used will be F'. Note that the inst field of F' is the instance that was declared in the closure (to enforce static scoping), and that the local variables in F' are the function parameters from the stack, followed by a list of zeros corresponding to the types of local variables required by the function. We prove the corresponding proof rule for `local`:

`wp_local_bind`

$$\frac{\xhookrightarrow{\text{FR}} F * \left(\xhookrightarrow{\text{FR}} F_1 \mathrel{-\!\!*} \mathsf{wp}\, es \left\{ w, \exists F_1', \xhookrightarrow{\text{FR}} F_1' * \left(\xhookrightarrow{\text{FR}} F \mathrel{-\!\!*} \mathsf{wp}\,[\texttt{local}_n\{F_1'\}\ w\ \texttt{end}]\left\{w', \Phi(w')\right\}\right)\right\}\right)}{\mathsf{wp}\,[\texttt{local}_n\{F_1\}\ es\ \texttt{end}]\left\{w', \Phi(w')\right\}}$$

which is reminiscent of `wp_ctx_bind`; the only reason this rule looks like more of a mouthful, is that the frame changes. As discussed above, this frame change is necessary for proper encapsulation.

Finally, the reason WebAssembly puts the function body in a `block` is to allow the function body to contain a `br` (with the right index) to exit the function-body's execution. Alternatively, a **return** instruction will work like a `br`, but target the closest `local` instruction. The **return** instruction also has an associated logical value `retV` lh_k, representing the expression $lh_k[\textbf{return}]$.

Example Consider the increment function with body $es_{incr} = [\textbf{i32}.\texttt{local.get}\ 0; \textbf{i32}.\textbf{const}\ 1; \textbf{i32}.\texttt{add}]$ of type $[\textbf{i32}] \to [\textbf{i32}]$. We show that calling it on input 3 returns 4.

Define es as $[\textbf{i32}.\textbf{const}\ 3; \texttt{call}\ \$\text{incr}]$, and let $F.\text{inst.funcs}[\$\text{incr}] = i$. We prove that

$$i \xmapsto{\texttt{wf}} \{(inst; []); es_{incr}\}^{\texttt{NativeCl}}_{[\textbf{i32}]\to[\textbf{i32}]} \mathrel{-\!\!*} \xhookrightarrow{\text{FR}} F \mathrel{-\!\!*} \mathsf{wp}\, es \{w, w = \texttt{immV}\ [\textbf{i32}.\textbf{const}\ 4]\}$$

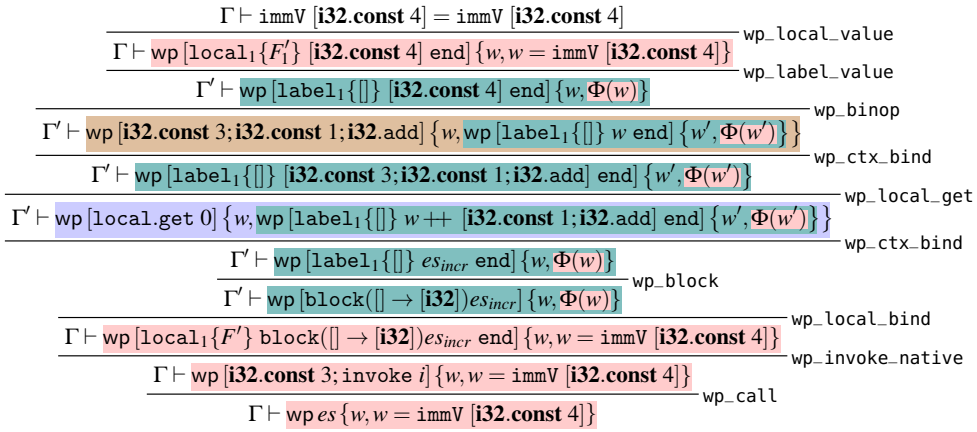
Here, the first precondition asserts that we know that function number i in the store is the increment function (we denote by *inst* the instance of the module where the increment function was defined), and the second precondition is ownership of the frame F.

We introduce the two preconditions by moving them to a proof environment Γ. For the first step of derivation, we apply the `wp_call` rule[6]. To fulfill the premises of the `wp_call` rule, the resource $\xhookrightarrow{\text{FR}} F$ from Γ is consumed, and it remains to prove

$$\triangleright(\xhookrightarrow{\text{FR}} F \mathrel{-\!\!*} \mathsf{wp}\,[\textbf{i32}.\textbf{const}\ 3; \texttt{invoke}\ i]\{w, w = \texttt{immV}\ [\textbf{i32}.\textbf{const}\ 4]\})$$

Now we introduce the $\triangleright$, move the frame resource back to our proof environment Γ, and are left with a new weakest precondition to prove. This first proof step corresponds to the bottom-most rule of the following simplified proof-tree:

[6]Some structural rules, which we have omitted here, allow it to be applied despite the constant preceding the `call` instruction.



As illustrated, we proceed by applying rule `wp_invoke_native`, leaving us with a new
weakest precondition to prove with the same environment Γ. In the figure, F' is defined as
$\{\mathsf{locs} := [\mathbf{i32.const}\ 3]; \mathsf{inst} := inst\}$, which is the frame where the call to the increment function needs to be executed in. Next we apply the rule `wp_local_bind` to bind the contents of the `local`. We give up the $\xhookrightarrow{\text{FR}} F$ resource to fulfill one premise. In its last premise, the new frame resource $\xhookrightarrow{\text{FR}} F'$ is introduced back to the context, and will be the frame we use to reason within the call to the increment function. We denote by Γ' this new proof environment where we own frame F' instead of F, and let $\Phi(w) = \exists F_1', \xhookrightarrow{\text{FR}} F_1' * \left(\xhookrightarrow{\text{FR}} F \mathrel{-\!\!*} \mathtt{wp}\,[\mathtt{local}_1\{F_1'\}\ w\ \mathtt{end}]\,\{w', w' = \mathtt{immV}\ [\mathbf{i32.const}\ 4]\}\right)$, which corresponds to the postcondition in the premise of the rule `wp_local_bind`.

The next few steps are mechanical, and we omit the details of some rules for brevity. We apply `wp_block` followed by `wp_ctx_bind` to focus on the first instruction of es_{incr}, `local.get`. We resolve it by applying rule `wp_local_get`, which inspects the locs field of the frame, and leaves us to prove the post-condition for 3. We apply `wp_ctx_bind` again to bind the binary operation **i32**.add, resolve it by applying `wp_binop`[7], and then `wp_label_value` to exit the label. It now remains to show $\Phi(\mathtt{immV}\ [\mathbf{i32.const}\ 4])$, which expands to

$$\exists F_1', \xhookrightarrow{\text{FR}} F_1' * \left(\xhookrightarrow{\text{FR}} F \mathrel{-\!\!*} \mathtt{wp}\,[\mathtt{local}_1\{F_1'\}\ [\mathbf{i32.const}\ 4]\ \mathtt{end}]\,\{w, w = \mathtt{immV}\ [\mathbf{i32.const}\ 4]\}\right)$$

We satisfy the existential with F', give up the resource $\xhookrightarrow{\text{FR}} F'$ from the context Γ' to satisfy the first part of the separating conjunction, and obtain $\xhookrightarrow{\text{FR}} F$ back, making our proof environment Γ again. We exit the `local` instruction (which is the function call context) by applying `wp_local_value`, and are left with our original postcondition to prove, which is now trivial when substituted with the value we obtained inside `local`. This completes the detailed proof.

[7] Formally, to use the rule as it was presented earlier, one must first frame in the resource $\xhookrightarrow{\text{FR}} F'$ in order to have the postcondition be of the right form. This means that, just like for every rule we have applied so far, even though we give up ownership of $\xhookrightarrow{\text{FR}} F'$ to fulfill one premise, we still get to use it to prove the other premise.

Example Coming back to the stack module from §2.1.1, we now outline what specifications for functions look like and, how they can be used by client modules. Take any function f. We write its specification in the general form:

$$\square\exists cl\ P, \forall i\ vs\ xs, \quad \Psi(P, vs, xs) \mathrel{-\!\!*} (i \overset{\texttt{wf}}{\longmapsto} cl) \mathrel{-\!\!*} \mathsf{wp}\ vs +\!\!+ [\texttt{invoke}\ i]\{w, \Phi(P, w, xs)\}$$

with Φ and Ψ some predicates specific to the function f. The persistence modality $\square$ simply indicates this specification can be duplicated as many times as needed;[8] we omit this modality in every specification that follows, for simplicity. Note the existential quantifiers. The first one, cl, abstracts over the actual closure of function f; because it is hidden behind an existential, it is hidden from clients. The second one, P, allows the specification to reference some abstract representation predicate. In the case of the functions from the "stack" module, we will have an existentially quantified predicate `isStack`, which hides the data representation from clients. We put all specifications under one large existential $\exists cl_{\text{push}}\ cl_{\text{pop}}\ cl_{\text{map}}\ \ldots\ \texttt{isStack}$, so that all specifications can share the predicate `isStack`.

The specification is thus a weakest precondition[9] on an `invoke`, with some precondition Ψ on the arguments vs given and some postcondition Φ. Both Ψ and Φ can mention the existentially quantified predicate P, as well as some universally quantified variables xs. The invocation address i is linked to the function f by the condition $i \overset{\texttt{wf}}{\longmapsto} cl$, that asserts that the function body is stored at address i. Let us give the concrete Φ and Ψ used for function "push":

$$\exists cl_{\text{push}}\ cl_{\text{pop}}\ cl_{\text{map}}\ \ldots\ \texttt{isStack}, \Big(\forall i\ v\ x\ s, \texttt{isStack}(v, s) \mathrel{-\!\!*} (i \overset{\texttt{wf}}{\longmapsto} cl_{\text{push}}) \mathrel{-\!\!*} \mathsf{wp}\,[\textbf{i32.const}\ x; v; \texttt{invoke}\ i]\{w, w = \texttt{immV}\ [] * \texttt{isStack}(v, x :: s)\}\Big) * \ldots \text{(other specs)}$$

To present the corresponding Φ and Ψ predicates for the "map" function, we need first to introduce some aspects about higher-order code in WebAssembly, which we do in §2.2.3.

Given a specification written in this form, and given the resource $i \overset{\texttt{wf}}{\longmapsto} cl_{\text{map}}$,[10] a client can verify its code in the presence of a call to the imported map function: when arriving at the instruction `call` $map, `wp_call` reduces `call` to `invoke`, and now the specification shown above can be applied.

Host Functions WebAssembly is meant to be defined independently of the host language in which it is embedded. However, the way the WebAssembly standard is phrased assumes that it is given some operational semantics of the host language as input, and embeds it in the operational semantics of WebAssembly. This phrasing suffices for defining the semantics of WebAssembly alone, which is what the WebAssembly standard does. However, when providing the first formal integration of WebAssembly with a separately-defined host language, we identified that this phrasing

[8] As a counterpart, proving this specification cannot rely on usage of any non-duplicable resource.

[9] In practice, we use the host weakest precondition $\mathsf{wp}_{\mathbf{HOST}}\ -\ \{-\}$ that we introduce in §2.3, as to allow functions to interact with the host via host calls. For functions that do not interact with the host, this makes no difference.

[10] The name of the index i and ownership of this resource are provided by instantiation when the client does the import.

is limiting, because it prevents formally giving the semantics of the combined host and embedded language as the integration of two concrete, separately defined language.

To account for this, we modify the presentation of the WebAssembly semantics (this is our only point of departure from the Coq formalization of Watt et al. [133]) so that the `invoke` of a host function reduces to a new `call_host` administrative instruction:

$$\texttt{invoke\_host}\ \frac{(S.\mathsf{funcs}[i] = \{hidx\}^{\texttt{HostCl}}_{ts_1 \to ts_2}) * (|ts_1| = |vs|)}{(S;F;vs +\!\!+ [\texttt{invoke}\ i]) \hookrightarrow (S;F;[\texttt{call\_host}\ (ts_1 \to ts_2)\ hidx\ vs])}$$

The closure $\{hidx\}^{\texttt{HostCl}}_{ts_1 \to ts_2}$ represents a *host* function imported from the host language that expects arguments of type ts_1 and yields results of type ts_2. The argument $hidx$ is an identifier that the host will use to determine what the desired function is. The `call_host` instruction remembers the function type tf, the 'host identifier' $hidx$ that allows the host language to identify which function is being called, and the function arguments vs. A `call_host` is stuck, and can only be unstuck by the host language, which typically replaces it by the return value of the call, possibly changing the frame or the store in doing so. We say more about the host interaction in §2.3.

We prove the following Iris-Wasm proof rule:

$$\texttt{wp\_invoke\_host}\ \frac{|vs| = |ts_1| * cl = \{hidx\}^{\texttt{HostCl}}_{(ts_1 \to ts_2)} * i \overset{\texttt{wf}}{\mapsto} cl * \overset{\text{FR}}{\hookrightarrow} F * \triangleright \left[\begin{array}{c} (i \overset{\texttt{wf}}{\mapsto} cl * \overset{\text{FR}}{\hookrightarrow} F) \mathrel{-\!\!*} \\ \mathsf{wp}\,(\texttt{call\_host}\ (ts_1 \to ts_2)\ hidx\ vs)\,\{w, \Phi(w)\} \end{array} \right]}{\mathsf{wp}\,(vs +\!\!+ [\texttt{invoke}\ i])\,\{w, \Phi(w)\}}$$

We introduce the `call_hostV` tf $hidx$ vs llh logical value, representing the stuck value
$llh[\texttt{call\_host}\ tf\ hidx\ vs]$. This allows for seamless binding rules when we introduce the host language's logical rules in §2.3. Since a `call_host` instruction is also stuck if it is under a `local` or under a `label`, we remember the context llh around the `call_host` as the fourth argument of `call_hostV`. This context llh is a generalized version of lh_k, that has a hole in nested `local`s and `label`s. In the rule above, $\mathsf{wp}\,(\texttt{call\_host}\ (ts_1 \to ts_2)\ hidx\ vs)\,\{\Phi\}$ is thus a weakest precondition on a value, and it thus suffices to show that $\Phi(\texttt{call\_hostV}\ (ts_1 \to ts_2)\ hidx\ vs\ [\_])$.

For example, when specifying the `"main"` function of the extended client module from §2.1.1, one intermediate goal, when verifying the part of the code corresponding to the call to the host function `$mut`, would have the form $\mathsf{wp}\ vs +\!\!+ \texttt{call \$mut}\,\{\Phi\}$, where vs represents the constant arguments we have pushed onto the stack prior to making the call. To prove this, one can simply apply rule `wp_call` to reduce `call` to `invoke`, and then rule `wp_invoke_host` to reduce the `invoke` to a `call_hostV` value. The computation is now reduced to a logical value, thus we now must prove that the postcondition Φ holds of the host call value. We cannot carry on to the rest of the code of the reentrant example if we stick at the WebAssembly level; this is in line with the nature of this call: it is a host call and needs interaction with the host to be unstuck. We will see in §2.3 how to reason about interaction with the host to prove the full specification of the reentrant example.

2.2.3 Higher-order code with `call_indirect`

As explained in §2.1.1, one can use `call_indirect` to implement higher-order functions with the help of the host language. The instruction `call_indirect` i, where i is an index into the types field of the module instance in the function frame, takes one argument k from the stack, and uses it as an index to look up the function to call in the table. The table itself is located in the store. Like for function invocation, the instance in the frame F finds the store-index ta of the correct table (i.e. the one at the head of the tables field). Now the kth element a of the table indexed ta can be looked up, and used as the index in the function closures component of the store, to find the closure cl to execute. As a side condition, the type of the closure must match the one declared by index i (that `call_indirect` takes as an immediate). Finally, $[\texttt{call\_indirect}\ i]$ reduces to $[\texttt{invoke}\ a]$, setting cl to be invoked in the next reduction step.

We prove the following program logic rule:

$$\frac{\begin{array}{l}\texttt{wp\_call\_indirect\_success}\\ \xhookrightarrow{\text{FR}} F * (F.\mathsf{inst.tabs}[0] = ta) * (ta \overset{\texttt{wt}}{\mapsto}_k a) * (a \overset{\texttt{wf}}{\mapsto} cl) * (F.\mathsf{inst.types}[i] = \mathsf{typeof}\ cl) *\\ \quad \triangleright \left((ta \overset{\texttt{wt}}{\mapsto}_k a) \mathrel{-\!\!*} (ta \overset{\texttt{wf}}{\mapsto} cl) \mathrel{-\!\!*} (\xhookrightarrow{\text{FR}} F) \mathrel{-\!\!*} \mathsf{wp}\,[\texttt{invoke}\ a]\,\{w, \Phi(w)\} \right)\end{array}}{\mathsf{wp}\,[\textbf{i32.const}\ k; \texttt{call\_indirect}\ \ i]\,\{w, \Phi(w)\}}$$

Here, we use the points-to predicate for elements of the table: only ownership of the relevant kth element of the table is required. Notice how the rule passes the ownership of all three points-to predicates (frame ownership, table element ownership and function closure ownership) to the continuing weakest precondition.

Example The higher-order `"map"` function of our stack module in §2.1.1 calls its argument function on each element in the stack by using **call_indirect**. We have now introduced enough logical machinery to present our modular specification of `"map"`:

$$\begin{array}{ll}
\exists cl_{\text{map}}\ \texttt{isStack}, \forall \Phi\ \Psi\ a\ v\ s\ F\ \ j\ k\ i, & (1)\\
\Box(\forall u.\, \Phi\ u \mathrel{-\!\!*} \ldots \mathrel{-\!\!*} \mathsf{wp}\ (\textbf{i32.const}\ u; \texttt{invoke}\ a)\,\{v, \Psi\ u\ v * \ldots\}) \mathrel{-\!\!*} & (2)\\
\quad \texttt{isStack}\ v\ s \mathrel{-\!\!*} \mathsf{stack_all}\ s\ \Phi \mathrel{-\!\!*} & (3)\\
\quad\quad (\xhookrightarrow{\text{FR}} F) \mathrel{-\!\!*} (F.\mathsf{inst.tabs}[0] = j) \mathrel{-\!\!*} (j \overset{\texttt{wt}}{\mapsto}_k a) \mathrel{-\!\!*} \ldots \mathrel{-\!\!*} (i \overset{\texttt{wf}}{\mapsto} cl_{\text{map}}) \mathrel{-\!\!*} & (4)\\
\quad\quad\quad \mathsf{wp}\ [\textbf{i32.const}\ k; v; \texttt{invoke}\ i]\,\{w, \exists s'.\, \texttt{isStack}\ v\ s' * \mathsf{stack_all2}\ s\ s'\ \Psi * \ldots\} & (5)
\end{array}$$

Let us describe the specification line by line: 1. As explained in §2.2.2, we existentially quantify over a closure cl_{map} and a predicate `isStack`, to hide our implementation of the stack and the body of the `"map"` function. We then universally quantify over many variables, including notably Φ and Ψ used in the specification of the mapped function, stressing this specification can be as general as needed. 2. The first precondition is a specification for the mapped function; it uses two predicates Φ and Ψ to express that for any **i32** input u that satisfies Φ, the mapped function returns an **i32** result v such that Ψ relates u with v. We have used '…' to elide some predicates, which are simply a copy of some of the resources from line 4, so as to allow usage of those resources (like frame ownership) in the proof of the specification of the mapped function. 3. Next, we describe the argument value v: it must represent a mathematical

stack s, all elements of which satisfy Φ. This is captured by the `isStack` v s predicate. 4. A points-to predicate for table j links the argument value k to the function index a (from the `invoke` in line 2). For brevity, we elide other side-conditions pertaining to typechecking the mapped function. At the end of the line, we have the function closure points-to predicate that links the index i of the invocation on line 5 to the `"map"` function closure. 5. After running `"map"`, we have a stack with logical state s' at location v, whose elements are related one-to-one to that of the previous logical state s by Ψ. For readability, we omit the second part of the postcondition, which simply gives back all of the resources from line 4.

To prove the above specification, the $stack module, who has access to the actual code of the `"map"` function, simply fills in the existential quantifiers with the actual closure of `"map"` and the definition of `isStack` reflecting the actual implementation. Then all that remains is a weakest precondition to prove, which is done by applying the rules in §2.2.2: `wp_invoke_native` using hypothesis $i \overset{\texttt{wf}}{\longmapsto} cl_{\text{map}}$, then `wp_local_bind`, to enter the local etc.

Note that we rely on the fact that our ambient logic, Iris, is a higher-order separation logic, in which weakest preconditions are just usual propositions. We stress again that the user of `"map"` does not need to know how `isStack` is defined (and in fact, we hide it with an existential quantifier surrounding the specification of the stack module, again exploiting the higher-order logic of Iris) or the physical state of the stack representation in memory: they only need to reason about the mathematical state, s; for example, stack_all only refers to s.

This example demonstrates that Iris-Wasm can be used to prove specifications for modules that cleanly hide the heavy indirection and low-level details of WebAssembly.[11] The use of `call_indirect` for higher-order programming, to call an arbitrary client function, goes beyond the 'encapsulated' fragment of WebAssembly of Watt et al. [132], and yet is captured modularly in the first line of our specification. Our accompanying Coq formalization contains a formal proof that a simple implementation of the stack module meets the specification. We can then apply the specification to different clients. In this paper, we focus on the reentrant client introduced in §2.1.1, see §2.3, and a client that applies `"map"` to an unknown and potentially malicious imported function (see §2.5). The code for these examples, and a few more, can be found in our Coq development.

2.3 Host Language and Proof Rules

In this section, we define a minimal host language featuring the core operations of the WebAssembly JavaScript Interface. The host fulfils two important roles; first, it embeds WebAssembly and defines the interoperability between WebAssembly and the host; and, second, it implements *module instantiation*, in which the host language

[11] Indeed, the specification shown here is akin to the specification for a stack module implemented in an ML-like programming language in standard Iris [6].

(import variable) $vi ::= nat$ (module variable) $vm ::= nat$ (host action id) $hidx ::= nat$

(declaration)	δ	::=	`inst_decl` $vis\ vm\ vis$ | `get_global` i
(host action)	a	::=	`nop` | `print` | `instantiate` δ | `call_wasm` | `table.set`
(import variable store)	I	::=	$vi \hookrightarrow export$
(host state)	H	::=	$\{\mathsf{store} : S, \mathsf{frame} : F, \mathsf{imports} : I, \mathsf{modules} : ms, \mathsf{actions} : as\}$
(host expression)	he	::=	$(es; \delta s)$ (host value) $hw ::= (vs; []) \mid (\texttt{trap}; [])$

Figure 2.4: Host Syntax (definitions reference the grammar in Fig. 2.2)

handles the allocation of WebAssembly states. Our minimal host language also has the ability to mutate WebAssembly function tables.

We begin by introducing the syntax of the host language and selected proof rules, with a focus on the interoperability with WebAssembly. We then detail the rules for module instantiation.

The syntax of the host language is shown in Fig. 2.4. Host expressions are pairs of WebAssembly expressions and host-specific declarations; host values are pairs of WebAssembly values, and an empty list of declarations. Finally, the host state is a record of the WebAssembly store and frame, as well as host-specific state. Host specific state has three components. First, it includes a store of export objects, to store the exports of an instantiated module, and to feed the imports of future instantiations. Note that while we call them import variables, they are used both for imports and exports. Subsequently, an *export* object refers to any object passed from one module to another, either as import or export. Second, it keeps track of a list of WebAssembly modules. Finally, to maintain the generality of host calls, host actions are indirectly referenced by indices into a list of available host actions.

To illustrate the expressive power of a host, our minimal host language includes five different host actions. `nop`, `print` and `instantiate` δ are pure operations that do not depend on host or WebAssembly store. More noteworthy are the `call_wasm` and `table.set` operations: `call_wasm` reduces to a WebAssembly call instruction, which opens up the possibility of reentrancy between the host and WebAssembly; `table.set` displays the expressive power of the host over the WebAssembly store, by mutating a given function table with a function from the WebAssembly store.

Declarations are either 1. instantiations `inst_decl` $vis\ vm\ vis$, which consist of a list of import/ export variables to feed into the imports of a module (referenced indirectly by its index into the module store), whose exports are stored in the subsequent list of import/export variables, or 2. load declarations for WebAssembly globals, to load the final output of a Wasm module's main function. The host operational semantics prioritises the reduction of WebAssembly expressions over that of instantiation declarations. We refer to the Coq formalization for a full account of the host operational semantics.

In the remainder of this section, we will discuss the proof rules of our new program logic for the host. We define our host logic using a weakest precondition predicate $\mathsf{wp}_{\mathbf{HOST}}\,(es; \delta s)\,\{hw, \Phi(hw)\}$, which intuitively means that the host expression $(es; \delta s)$

does not get stuck and, if it terminates with the host value hw, then the predicate Φ holds for hw.

While the host weakest precondition is not to be confused with the Wasm weakest precondition, it shares some similarities in its memory model. The memory model of the host program logic extends the memory model of the Wasm program logic, as it includes the Wasm store. We reason about the host-specific part of the host state using three new predicates: 1. $vi \overset{\texttt{vis}}{\mapsto} export$: a points-to predicate for the export object store; 2. $vm \overset{\texttt{mod}}{\hookrightarrow} m$: a points-to predicate for the module store; 3. $hidx \overset{\texttt{ha}}{\hookrightarrow} a$: a points-to predicate for the host action store. We present the host program logic in two parts: first we discuss the rules that implement interoperability between WebAssembly and the host, and second we discuss module instantiation.

Interoperability The first key to WebAssembly and host interoperability is the WebAssembly lifting step. Any reduction in the WebAssembly part of a host expression corresponds to a step in the host expression, as captured by the following bind rule:

$$\frac{\texttt{wp\_lift\_wasm}\quad \mathsf{wp}\, es\, \{w, \mathsf{wp}_{\mathbf{HOST}}\, (w; \delta s)\, \{hw, \Phi(hw)\}\}}{\mathsf{wp}_{\mathbf{HOST}}\, (es; \delta s)\, \{hw, \Phi(hw)\}}$$

Note that w may be a logical value, in particular a suspended host call from Wasm to the host, which can now be resolved via the host proof rules for `call_host`. Recall the definition of a stuck host call: the `call_host` *tf hidx vs* administrative instruction is considered stuck in any nested WebAssembly context *llh*, and is interpreted as the logical value `call_hostV` *tf hidx vs llh*, in which *hidx* refers to the host action identifier which is storing the executing host action, *tf* refers to its type, and *vs* refers to the parameters of the invocation. Each host action is resolved via a different proof rule.

In particular, one such host action is a call in the other direction, from the host to Wasm. In that case, the inner `call_wasm` action, performed by the host function *hidx*, reduces to the WebAssembly instruction `call` as follows.

$$\frac{\begin{array}{l}\texttt{wp\_host\_action\_call\_wasm}\\ \qquad hidx \overset{\texttt{ha}}{\hookrightarrow} \texttt{call\_wasm}\ *\\ \triangleright\, (hidx \overset{\texttt{ha}}{\hookrightarrow} \texttt{call\_wasm} \mathrel{-\!\!*} \mathsf{wp}_{\mathbf{HOST}}\, (llh[\texttt{call}\ i]; \delta s)\, \{hw, \Phi(hw)\})\end{array}}{\mathsf{wp}_{\mathbf{HOST}}\, (llh[\texttt{call\_host}\ tf\ hidx\ [\texttt{i32.const}\ i]]; \delta s)\, \{hw, \Phi(hw)\}}$$

Reentrant Example We now have all we need to prove a specification for the extended (reentrant) client introduced in §2.1.1. This specification will be parametrized with specifications for all the functions from the stack module (and thus with all the existentials of those specifications, most importantly the `isStack` predicate), and can be *modularly* combined with a specification for the stack module.

Our specification could look like this:

$$\begin{array}{c}\exists cl_{\text{main}}, \forall v\ x_1\ \ldots\ x_n\ i\ hidx, \quad \texttt{isStack}\ v\ [x_1, \ldots, x_n] \mathrel{-\!\!*} i \overset{\texttt{wf}}{\mapsto} cl_{\text{main}} \mathrel{-\!\!*}\\ \texttt{OwnClosures}([\$\text{f0}; \$\text{f3}; \$\text{map}]) \mathrel{-\!\!*} \$\text{mut} \overset{\texttt{wf}}{\mapsto} \{hidx\}^{\texttt{HostCl}}_{[\textbf{i32};\textbf{i32}]\to[]} \mathrel{-\!\!*}\\ hidx \overset{\texttt{ha}}{\hookrightarrow} \texttt{table.set} \mathrel{-\!\!*}\\ \mathsf{wp}_{\mathbf{HOST}}\, ([\textbf{i32.const}\ v; \texttt{invoke}\ i], [])\, \{hw, \texttt{isStack}\ hw\ [f_3(f_0(x_1)), \ldots, f_3(f_0(x_n))] * \ldots\}\end{array}$$

The elided postconditions give back all the preconditions; $\texttt{OwnClosures}(fs)$ asserts ownership, for all functions $f \in fs$, of a closure cl_f. For the function $map imported from the stack module, the closure is the one referenced in the specification of the stack module. In order to carry out our proof, we assume we are given specifications for functions $f0 and $f3 that reference cl_{f_0} and cl_{f_3}.

To prove this specification, we fill in the existential quantifier for cl_{main} with the actual code of the "main" function. Now we apply `wp_lift_wasm` to bring ourselves to proving a WebAssembly weakest preconditon: the postcondition now becomes $w, \mathsf{wp}_{\mathbf{HOST}}\, w\, \{hw, \Phi(hw)\}$ where Φ is the postcondition in the weakest precondition shown above. We can now begin the proof just like we proved all the specifications for the functions in the stack module: we apply `wp_invoke_native`, then `wp_local_bind`, etc.

As showcased in §2.2.2, the WebAssembly weakest precondition gets stuck on a value when it arrives at the host call: we now need to show that the postcondition holds of the `call_hostV` value, i.e. that

$$\mathsf{wp}_{\mathbf{HOST}}\, llh[\texttt{call\_host}\ tf\ hidx\ vs]\, \{hw, \Phi(hw)\}$$

where *llh* is the context in which the host call was, containing for instance all the code that follows the host call. To prove this, we have a rule `wp_host_action_table_set` similar to rule `wp_host_action_call_wasm` shown above, that, given our knowledge of $n \overset{\texttt{wt}}{\longmapsto}_0$ $f0, gives back $n \overset{\texttt{wt}}{\longmapsto}_0$ $f3, and brings us to prove a (host) weakest precondition statement on the code that follows the host call, with this new function at the 0th place in the table. We can prove this by lifting to WebAssembly and carrying out the proof in the WebAssembly program logic until the end.

Module Instantiation While WebAssembly 1.0 does not depend on any particular host language, it does define a *specification* for module instantiation. Any host language is tasked with implementing instantiation according to that specification. We thus conceptually distinguish between the parts of module instantiation pertaining to the official WebAssembly specification, and the parts that deal with the host language. $\texttt{Instantiate}(S, m, exportdescs, ((S', inst, exports), start))$ defines the specification for module instantiation. The full definition is quite elaborate; we refer to the Coq mechanization for all details, and provide an intuitive overview here. In essence, it states that *inst* is the result of instantiating module *m* while importing *exportdescs*, *exports* are the resulting exports, and S' is the resulting WebAssembly store, in which all the relevant state has been allocated.

The specification enforces various side conditions. First, the module must be well typed according to a list of relevant import and export types. Next, it asserts the necessary operational conditions on the allocated state and created instance; that all the fields of the instance are properly initialized (e.g. any function table is initialized with the proper elements as defined by the module), that all the initialized values are within the bounds of the initialized object, and finally that the start function is either empty, or refers to a function of the module of type $[] \to []$.

The instantiation specification specifies the outcome of module instantiation on the WebAssembly store. Note that the specification is host language agnostic. The semantic outcome of instantiation on the WebAssembly store ought likewise to be independent of the host language that implements it. The following lemma captures the effects of instantiation on the interpretation of the WebAssembly store as Iris resources, according to the host agnostic instantiation specification. The lemma is thus independent of any host language definition.

Lemma 1 (Module Instantiation Resource Allocation)**.**

$$
\begin{array}{ll}
\textit{If} & \vdash m : \mathit{timps} \to \mathit{texps} \;\wedge\; \texttt{constInits}(m) \\
\textit{and} & \texttt{Instantiate}(S, m, \mathit{imports}, ((S', \mathit{inst}, \mathit{exports}), \mathit{start})) \\
\textit{then} & \texttt{resourcesImports}(m, \mathit{imports}, \mathit{timps}, \mathit{wfs}, \mathit{wts}, \mathit{wms}, \mathit{wgs}) * \\
& \texttt{stateInterp}(S) \\
& \Rrightarrow \texttt{resources}(m, \mathit{imports}, \mathit{timps}, \mathit{wfs}, \mathit{wts}, \mathit{wms}, \mathit{wgs}, \mathit{start}, \mathit{inst}) * \\
& \texttt{stateInterp}(S')
\end{array}
$$

For readability, we omit the technical details behind some of the above predicates. It suffices to know the following: `constInits` limits the global initializers and offsets to be constants, `resourcesImports` defines the points-to predicate associated with each import in `imports`, and `resources` defines all the points-to predicates associated to the created instance *inst*, including those that were previously imported. The variables *wfs*, *wts*, *wms* and *wgs* are maps that summarise the values of functions, tables, memories and globals of the created instance. (The $\Rrightarrow$ modality is used in Iris to update ghost resources [52].) Using Lemma 1, we can then prove a host weakest precondition rule for host instantiation, that we will refer to as `wp_host_instantiate`.

Example The complete stack module is an instantiation declaration, which exports closures for push, pop, new_stack, is_empty, is_full, stack_length and map, as well as the function table invoked by map. We recall that exports are passed via indices into the import variable store.

$$
\mathit{vm} \triangleq \{0 \mapsto \mathsf{stack\_module}\} \qquad \mathit{host\_program} \triangleq ([], [\texttt{inst\_decl}\ []\ 0\ [0,1,2,3,4,5,6,7]])
$$

The Iris-Wasm specification of the complete stack module from §2.1.1 is as follows (we elide the exporting of the table, for simplicity):

$$
\begin{array}{l}
\exists \mathsf{stack\_module}, \forall i\ \mathit{js}, (i \xhookrightarrow{\texttt{mod}} \mathsf{stack\_module}) \mathrel{-\!\!*} \mathop{\Large *}_{j \in \mathit{js}} j \xmapsto{\texttt{vis}} - \mathrel{-\!\!*} \\
\mathsf{wp}_{\mathbf{HOST}}\,([]; [\texttt{inst\_decl}\ []\ i\ \mathit{js}])
\left\{
\begin{array}{r}
\exists \mathit{cl}_{\mathrm{push}}\ \mathit{cl}_{\mathrm{pop}}\ \ldots\ \mathit{cl}_{\mathrm{map}}, \texttt{isStack}, \mathsf{spec\_push} * \mathsf{spec\_pop} \\
* \ldots * \mathsf{spec\_map} * \mathop{\Large *}_{j \in \mathit{js}} j \xmapsto{\texttt{vis}} \mathsf{function\_export}\ \mathit{cl}_j
\end{array}
\right\}
\end{array}
$$

spec_push is the specification of the "push" method shown earlier. Likewise for the other specifications mentioned in the postcondition. Both the contents of the $stack module and the implementations of the stack operations are hidden from clients because of the existential quantifiers.

Figure 2.5: Lines of code of the Iris development, as given by cloc

helpers	language	rules	instantiation	host
11836	3685	7123	6828	2339

examples	logrel	stack	total
2754	8145	8787	51497

This stack module specification is proven by applying rule `wp_host_instantiate`, which populates the value import stores and gives ownership of all the resources necessary for the stack module operations, and then we apply the specifications for the stack operations shown in §2.2.3.

With this specification for the stack module and a similar one for the client module (parametrized by the specification of the stack), we verify the complete stack program (a sequence of instantiations) in our Coq formalization.

2.4 Mechanization in the Iris Framework

We implement and prove the Iris-Wasm proof rules in this paper in the Iris framework in the Coq proof assistant. Iris was originally developed to reason about programs with complex concurrency; however, the same mechanisms have proven useful to reason about complex sequential programs such as the awkward example, as demonstrated for example by Georges et al. [33]. In this paper, we focus our presentation on the novel, language-specific proof rules we introduce and prove, but our program logic also inherits many other logical constructs and proof rules from Iris which we make use of in our development. We have already mentioned the 'later' modality, $\triangleright$, which avoids circularities in the presence of the higher-order features of Iris, and which can be used to define guarded recursive predicates in Iris, as well as the 'persistence' $\square$ and 'update' $\Rightarrow$ modalities. Other features we use include the frame rule, non-atomic invariants, ghost state, and other proof rules like Löb induction; for a thorough introduction to those, see Jung et al. [52].

We prove all our proof rules in Iris, with respect to the default definition of the weakest precondition predicate (with an extra requirement that the frame resource holds for every step of reduction) instantiated to refer to the Coq formalization of the official WebAssembly 1.0 operational semantics by Watt et al. [133].

The adequacy theorem of Iris [52, §6.4] then yields the final desired soundness theorem, which intuitively says that if a weakest precondition for a WebAssembly or host program has been proved in Iris-Wasm, then it does indeed mean that the program runs safely, according to the official WebAssembly 1.0 operational semantics, or the host language that embeds it. An example of the latter can be found in the Coq mechanization.

The size of the full Iris development is summarized in Fig. 2.5. The `logrel` folder

contains a case study presented in the next section, and `stack` contains the full stack module and associated clients.

The stack module, with a binary size of 637 bytes, is defined in around 200 lines code in Coq, with the module type checking done in 300 lines of code using the type checker from Watt et al. [133]. The module specification is fully verified using the Iris-Wasm logic in around 3800 lines of code in Coq, where 2100 lines are used to verify each of the module function specifications, and the remaining code is used to prove the top-level instantiation specification and auxiliary lemmas. Such a ratio between program and proof size may hint at a substantial verification effort. However, it's important to note that it reflects a version of Iris-Wasm without a bespoke proof mode; an interesting line of future work is to extend Iris-Wasm with various automation techniques, such as the proof search strategy of Mulder et al. [73], and use it to prove specifications of large real-world programs.

2.5 Case Study

We showcase the utility of our program logic through a case study[12]. The goal is to leverage the coarse-grained encapsulation guarantees of WebAssembly modules to prove robust safety of two scenarios involving some interaction between a known module and an unknown, potentially malicious, module. While the coarse-grained encapsulation properties granted by modules are relatively shallow (one module cannot interact with the internals of another), the reasoning principles are not: not only are we reasoning about unknown code, the desired robust safety property can be subtle, and highly specific to the particular implementation of a robustly safe module. We emphasize that we do not seek to either define or prove encapsulation as a meta-property, rather, we define and apply a methodology to prove robust safety of specific modules.

WebAssembly's modules are designed to allow trusted code to encapsulate its local state (e.g. variables and memory), by limiting what is shared with untrusted modules via imports and exports. This encapsulation is meant to hold no matter what other modules do, either by accident or by malice, and thus does not rely on compliance. Modules can take advantage of this encapsulation to guarantee various safety properties. To prove those properties formally, we may need to reason about the interaction between known, trusted code and unknown, untrusted code. We have thus far presented a program logic to reason about known code only. In this case study, we use the program logic to build a method to reason about the instantiation of unknown code, and use it to prove the *robust* safety of known code, that is, safety even when composed with adversarial code.

The methodology is based on a relational interpretation of WebAssembly types, built on top of our Iris-Wasm program logic, by defining logical relations for each WebAssembly type. The key idea is to interpret the types of primitives, functions,

[12]Our Coq mechanization also includes another case study of a program that uses recursion through the store, by applying a host call to mutate the function table, known as Landin's Knot.

```
1 m_client ≜ (module ;; Another Stack Client
2 (import "adv" "f" (func $f (param i32) (result i32)))
3 (import "stack" "map" (func $map (param i32 i32)))
4  ... ;; import global g and the remaining stack module
5 (elem (i32.const 0) $f) ;; populate table with imported
      function
6 (func $main (local $i i32)
7  call $new_stack; ... ; const 4; call $push;
8  local.get $i; const 2; call $push;
9  local.get $i; call $map;
10 local.get $i; call $stack_length; global.set $g))
11
```

```
stack_client ≜
inst_decl [] "stack" ["tab";...;"pop"]
inst_decl [] "adv" ["f"]
inst_decl ["f";"g";"tab";...;"pop"] "client" []
```

Figure 2.6: Robust safety example:
applying map on an imported function

etc., all the way to module types, as propositions in Iris-Wasm. The methodology of defining logical relations in Iris is well known [33, 51, 57, 120], but here it is for the first time applied to the type system of a full industrial standard, namely the WebAssembly type system. We define semantic interpretations for all WebAssembly types. That includes all the internals of a module, and in particular it includes the types of exports and imports. We say that an import object is safe to share, or valid, if it is in the appropriate relation. All the results in this section have been formally proved in Coq. We give an overview here, and refer the reader to the accompanying Coq code for the full definition of the relational interpretation of WebAssembly types.

The interpretation of module types via the instance relation, denoted $\mathscr{I}[\![C]\!]$, is the keystone to derive specifications for unknown functions. The following key theorem states that the result of instantiating a well-typed module $\vdash m : timps \to texps$ produces a valid instance, given that all imports are valid according to *timps*.

Theorem 1 (Valid Instance Allocation)**.** *If* $\vdash m : timps \to texps$*, and inst is the result of instantiating module m with imports imps, then*

$$\texttt{resources}(m, imps, timps, \cdots, inst) \mathrel{-\!\!*} \texttt{valid}[\![timps]\!](imps) \mathrel{-\!\!*} \mathscr{I}[\![C]\!](inst)$$

where C is the module type, determined syntactically, $\texttt{resources}(\cdots, inst)$ *corresponds to the ghost resources allocated by module instantiation as depicted by Lemma 1, and* $\texttt{valid}[\![timps]\!](imps)$ *unfolds the list of imports, and applies the relevant relation on each import object.*

Proof. By unfolding the definition of module typing, inferring properties about the result of instantiating *m*, and component-wise proving the instance relation. Validity

of imported types is established by the $\texttt{valid}[\![\mathit{timps}]\!](\mathit{imps})$ assumption, while the rest are established using the fundamental theorem of logical relations (FTLR). The FTLR, which roughly states that all well-typed programs are semantically well-typed, is a key non-trivial language property, and is proved by induction over the full type system. □

Applications of the Logical Relation Next we describe two scenarios, each involving our stack module interacting with some unknown function. In each case, the two modules interact via imported closures. We will therefore employ the closure relation $\mathscr{C}\mathit{los}$ as the principal logical relation in our reasoning.

The two applications highlight a conceptual distinction between two kinds of scenarios in which known code interacts with unknown code. In the first example, known code imports functions from an unknown module, and has a certain amount of control over how these are applied. The second example exports known code to an unknown module, and in that case, exported closures must carefully guard against misuse.

Fig. 2.6 depicts a client of the stack module, which imports a closure `"f"` of type $[\mathbf{i32}] \to [\mathbf{i32}]$ from an unknown module. The client creates a new stack, pushes two values, then applies `map` using the imported unknown function, and finally computes the length of the stack by calling a function from the stack module. The stack module hides its internal representation from the context. Likewise, the host makes sure to hide the stack module operations from the unknown module. WebAssembly's coarse grained encapsulation thus guarantees that the integrity of the allocated stack is maintained, no matter what the unknown imported function does: as long as it does not trap, the final `length` operation succeeds and returns the original size of the stack, namely 2. We refer to imports and modules via names rather than indices, for the sake of readability. The following theorem expresses robust safety formally:

Theorem 2 (Top-level Host Specification)**.** *If* $\vdash m_{\texttt{adv}} : [] \; [\texttt{func}_{\texttt{e}} \; ([\texttt{i32}] \to [\texttt{i32}])]$ *and the syntactic restrictions on* $m_{\texttt{adv}}$ *hold, then*

$$\left\{\begin{array}{l}\texttt{"stack"} \xhookrightarrow{\texttt{mod}} m_{\texttt{stack}} * \texttt{"adv"} \xhookrightarrow{\texttt{mod}} m_{\texttt{adv}} * \\ \texttt{"client"} \xhookrightarrow{\texttt{mod}} m_{\texttt{client}} * \texttt{"g"} \xmapsto{\texttt{vis}} \$g * \\ \$g \xmapsto{\texttt{wg}} - * [\mathit{NaInv} : \top] * \\ \texttt{"f"}, \texttt{"tab1"}, \texttt{"map"}, \ldots, \texttt{"pop"} \xmapsto{\texttt{vis}} -\end{array}\right\} \texttt{stack\_client} \left\{ hw, \begin{array}{c}(hw = ([];[]) \wedge \\ \$g \xmapsto{\texttt{wg}} 2) \vee \\ hw = (\texttt{trap};[])\end{array} \right\}$$

Proof. Once the host has allocated the unknown module, we apply Theorem 1 to conclude that its instance is valid, which guarantees that each of its components, including the exported closure of type $[\mathbf{i32}] \to [\mathbf{i32}]$, is valid. As a result, we know that the unknown import of our client is in the closure relation $\mathscr{C}\mathit{los}$, which by definition of the relational interpretation includes a specification for the unknown function. Crucially, this specification does not depend on the stack internals, and thus we are able to prove that the stack size is maintained. □

Next we consider a scenario in which an unknown module imports operations from the stack module, namely `new_stack`, `push` and `pop`. The encapsulation of

the stack module's internal state, alongside careful checks at the boundaries of each operation, which we will elaborate on below, should guarantee that the stack module memory indeed stores and maintains stacks, as defined by the **isStack** predicate, irrespectively of what the unknown module does. Henceforth we will refer to this as the representation invariant, denoted by $\texttt{stackInvariant}(m)$, where m is the index of the encapsulated memory. Roughly, the representation invariant is an Iris (non-atomic) invariant containing a big separation of **isStack** predicates, one for each allocated stack.

The basic type system of WebAssembly guarantees that the adversary code does not get stuck. However, our goal is to reason about integrity of the data representation enforced by the module system. While the type system defines the typing of an individual module, it does not consider interweaving of module instantiations, since instantiation is handled by a host, typically written in untyped JavaScript. Therefore, the type system is too weak to capture the data abstraction enforced by the module system, which we are relying on here. As such, our interpretation of the type system does not capture the refined interpretation (with the representation invariant) of the stack module.

We use the standard type interpretation of the adversary module to reason about its execution. However, we want this interpretation to depend on the *refined representation invariant* of the stack module internals, rather than the *default interpretation granted by the logical relation*. Since each import must be valid when applying Theorem 1, we *manually* prove that, given the representation invariant, each exported function (`new_stack`, `push`, and `pop`) is in the closure relation.

As a result, we must now consider the case where a stack operation is applied on an arbitrary input value. Consider, for instance, `push` – it takes two arguments, one of which is a stack value, which is interpreted as a memory address. A malicious adversary could apply `push` to a masked stack value (a bogus memory address), thus breaking the expected internal behavior of the stack module. `push` must thus guard against such a situation by dynamically checking the validity of all safety-critical parameters. These dynamic checks ensure that no stack gets corrupted. Relying on those dynamic checks, we can then prove specifications that maintain the representation invariant:

Theorem 3 (Validity of Select Stack Module Operations)**.** *If* $inst.\texttt{mems} = [m]$ *then,*

$$\begin{aligned}\texttt{stackInvariant}(m) \to\ & \mathscr{C}los[\![[\mathbf{i32};\mathbf{i32}] \to []]\!](\{(inst, [\mathbf{i32}]);\texttt{push}\}^{\texttt{NativeCl}}) \\ & * \mathscr{C}los[\![[\mathbf{i32}] \to [\mathbf{i32}]]\!](\{(inst, [\mathbf{i32}]);\texttt{pop}\}^{\texttt{NativeCl}}) \\ & * \mathscr{C}los[\![[] \to [\mathbf{i32}]]\!](\{(inst, [\mathbf{i32}]);\texttt{new\_stack}\}^{\texttt{NativeCl}})\end{aligned}$$

The representation invariant is allocated upon instantiation of the stack module, at which point there are no allocated stacks. Theorem 3 is then applied on each of the relevant stack module exports, such that we can apply Theorem 1, and conclude with the standard type interpretation of the adversary module, while maintaining the now allocated representation invariant.

2.6 Related Work

Watt et al. [132] develop a mechanized first-order separation logic for what they call "encapsulated" WebAssembly, that is, code limited to a single module, with no exports or imports, and no uses of the **call_indirect** instruction or the host, and they do not handle instantiation. For their subset of the language, our proof rules are similar up to presentational details, except for the handling of breaks, where, as mentioned in §2.2.2, we use a novel approach with a bind rule which scales to higher-order programs, unlike the approach taken by Watt et al. [132].

WebAssembly provides coarse-grained memory safety, at the boundary of memory objects, and coarse-grained isolation, at the boundary of modules. Lehmann et al. [61] show that many of the classical attacks against memory unsafe languages, targeting a finer granularity, also work against Wasm programs that not specifically written to take advantage of module isolation. We show in our examples that, when Wasm programs are written with module isolation in mind, the language specification does indeed enforce expected isolation guarantees.

MSWasm [24, 69] (Memory-Safe Wasm) is a proposed extension of WebAssembly that adds first-class support for CHERI-like [129] fine-grained runtime-checked memory capabilities. The logical relation of Cerise [33, 35–37], mechanized in Iris, captures encapsulation for hardware capabilities in an idealized assembly model and may be used as a starting point to formalize the guarantees of MSWasm on top of Iris-Wasm.

CapableWasm [29] is a (work-in-progress) extension of the type system of WebAssembly to support compositional compilation from different languages. They rely on their type system to enforce finer-grained encapsulation than at the module boundary.

Kolosick et al. [56] use a logical relation to show that WebAssembly programs naturally compile to unsafe platform assembly in such a way that the compiled code obeys a safe calling convention and certain isolation properties with respect to the rest of the system. Narayan et al. [75] rely on this result to implement a sandboxing technique whereby C code is first compiled to WebAssembly which is then ultimately compiled to native assembly for linking. They use this technique to sandbox a number of Firefox libraries.

Many related works deal with the mechanized formalization of low-level languages. RockSalt [71] is a verified checker that validates code binaries against a sandbox policy, similar to that of Google's Native Client (NaCl). RockSalt is mechanically verified using a formalization of a subset of x86 in Coq. Kennedy et al. [54] use Coq to build a macro assembler for x86, while relating machine code to separation logic formulas suitable for program verification.

The Certified Assembly Programming (CAP) family of frameworks [28, 77, 140, 141] support the definition of second-order Hoare logics for verifying modular specifications of low-level assembly programs, using expressive features such as embedded code pointers, concurrency, and dynamic thread creation. As such, CAP focuses on features that are abstracted away by Wasm. Gu et al. [39] presents CertiKOS, an

extensible architecture for certifying concurrent OS kernels. Using CertiKOS, Gu et al. [39, 40] develop and verify a concurrent OS kernel consisting of both C and x86 assembly code. By leveraging CompCertX [38], CertiKOS is able to reason about interactions between C and x86 assembly. As is the case with Iris-Wasm, the setup assumes that the two languages share the same memory model. The recent DimSum [103] framework supports reasoning about multilingual programs between languages with different memory models. However, while Iris-Wasm focuses on mechanizing the full language of a real industrial standard, the DimSum approach has so far only been applied to a simple high-level imperative language and an idealized assembly language.

The W3C have announced a Public Working Draft for WebAssembly 2.0. It includes several features orthogonal to our focus on security, such as extra numeric operations. The two relevant features are: the lifting of the artificial restriction to one table per module (we have done this too), which corresponds to a simple update to the relation on instances; and the addition of opaque reference types to objects of the host language, which adds new WebAssembly values, but no actual complexity because of their opacity (this is trivial to do).

2.7 Conclusion

We have presented Iris-Wasm, a practical higher-order, mechanized program logic for the W3C WebAssembly 1.0 official language standard [95], building on the mechanized WasmCert-Coq specification [133]. We show how the reasoning of Iris-Wasm can handle the intricacies of WebAssembly, including interaction with its host language and the higher-order programs and reentrancy that it enables, going far beyond the 'encapsulated' fragment of WebAssembly in previous work [132]. We then leverage our program logic to build a logical relation which enforces robust safety, demonstrating that we can prove properties of encapsulation at module boundaries. This example illustrates the potential of what can be done with formal methods. We hope other researchers will use our formalization to further investigate the WebAssembly ecosystem, and that industrial language communities will thereby be further enticed to embrace the formalization of language specifications.

Acknowledgements

This work was supported in part by a Villum Investigator grant (no. 25804), Center for Basic Research in Program Verification (CPV), from the VILLUM Foundation. Conrad Watt is supported by a Research Fellowship from Peterhouse, University of Cambridge. Rao is supported by a Doctoral Scholarship Award from Department of Computing, Imperial College London. Gardner is supported by the EPSRC fellowship VeTSpec: Verified Trustworthy Software Specification (EP/R034567/1).

Artifact Availability

The artifact [90] of this paper, containing the full Coq development, is available on Zenodo. Detailed instructions of usage are provided within the artifact itself.

Appendices

In these appendices, we present a full picture of the Iris-Wasm program logic and logical relation. This full picture can also be found in the Coq development.

2.A The Full Iris-Wasm Program Logic

2.A.1 Proof Rules

Here we present the Iris-Wasm proof rules for the whole of WebAssembly. Note that in the Coq formalization, additional rules exist for convenience. Notably, many rules like `wp_call` have a corresponding `wp_call_ctx` rule that (almost) corresponds to successively applying the bind rule and the `wp_call` rule.

Let us begin by recalling this general Iris rule:

$$\frac{\text{WP\_VALUE}}{\dfrac{\text{isLogVal}(es, w) * \Phi(w)}{\mathsf{wp}\, es\, \{w, \Phi(w)\}}}$$

We give all other rules in Figures 2.7 to 2.11.

wp_unop

$$\frac{[\![t.unop]\!](c) = c' * \triangleright \Phi(\texttt{immV}\ [t.\textbf{const}\ c']) * \xhookrightarrow{\text{FR}} F}{\mathsf{wp}\ (t.\textbf{const}\ c; t.\textbf{unop}\ unop) \left\{ w, \Phi(w) * \xhookrightarrow{\text{FR}} F \right\}}$$

wp_binop

$$\frac{[\![t.binop]\!](c_1, c_2) = c \quad * \quad \triangleright \Phi(\texttt{immV}\,[t.\textbf{const}\ c]) \quad * \quad \xhookrightarrow{\text{FR}} F}{\mathsf{wp}\ [t.\textbf{const}\ c_1; t.\textbf{const}\ c_2; t.\textbf{binop}\ binop] \left\{ w, \Phi(w) * \xhookrightarrow{\text{FR}} F \right\}}$$

wp_binop_failure

$$\frac{[\![t.binop]\!](c_1, c_2) = \bot * \ \triangleright \Phi(\texttt{trapV}) * \xhookrightarrow{\text{FR}} F}{\mathsf{wp}\ (t.\textbf{const}\ c_1; t.\textbf{const}\ c_2; t.\textbf{binop}\ binop) \left\{ w, \Phi(w) * \xhookrightarrow{\text{FR}} F \right\}}$$

(Other numerical rules are similar)

wp_unreachable

$$\frac{\triangleright \Phi(\texttt{trapV}) * \xhookrightarrow{\text{FR}} F}{\mathsf{wp}\ (\textbf{unreachable}) \left\{ w, \Phi(w) * \xhookrightarrow{\text{FR}} F \right\}}$$

wp_drop

$$\frac{\triangleright \Phi(\texttt{immV}\ []) * \xhookrightarrow{\text{FR}} F}{\mathsf{wp}\ (t.\textbf{const}\ c; \textbf{drop}) \left\{ w, \Phi(w) * \xhookrightarrow{\text{FR}} F \right\}}$$

wp_nop

$$\frac{\triangleright \Phi(\texttt{immV}\ []) * \xhookrightarrow{\text{FR}} F}{\mathsf{wp}\ (\textbf{nop}) \left\{ \lambda\ w \Phi(w) * \xhookrightarrow{\text{FR}} F \right\}}$$

wp_select_true

$$\frac{n \neq 0 * \triangleright \Phi(\texttt{immV}\ [t.\textbf{const}\ c_1]) * \xhookrightarrow{\text{FR}} F}{\mathsf{wp}\ (t.\textbf{const}\ c_1; t.\textbf{const}\ c_2; i32.\textbf{const}\ n; \textbf{select}) \left\{ w, \Phi(w) * \xhookrightarrow{\text{FR}} F \right\}}$$

wp_select_false

$$\frac{n = 0 * \triangleright \Phi(\texttt{immV}\ [t.\textbf{const}\ c_2]) * \xhookrightarrow{\text{FR}} F}{\mathsf{wp}\ (t.\textbf{const}\ c_1; t.\textbf{const}\ c_2; i32.\textbf{const}\ n; \textbf{select}) \left\{ w, \Phi(w) * \xhookrightarrow{\text{FR}} F \right\}}$$

Figure 2.7: Pure rules

wp_local_bind

$$\frac{\xhookrightarrow{\text{FR}} F * \left[\xhookrightarrow{\text{FR}} F_1 \mathrel{-\!\!*} \left(\mathsf{wp}\, es \left\{w, \exists F_1', \xhookrightarrow{\text{FR}} F_1'\right\} * \left(\xhookrightarrow{\text{FR}} F \mathrel{-\!\!*} \mathsf{wp}\,[\mathbf{local}_n\{F_1'\}\ w\ \mathbf{end}]\,\{w, \Phi(w)\}\right)\right)\right]}{\mathsf{wp}\,[\mathbf{local}_n\{F_1\}\ es\ \mathbf{end}]\,\{w, \Phi(w)\}}$$

wp_ctx_bind

$$\frac{\mathsf{wp}\, es \left\{w, \mathsf{wp}\, lh_i[w]\left\{w', \Phi(w')\right\}\right\}}{\mathsf{wp}\, lh_i[es]\left\{w', \Phi(w')\right\}}$$

Figure 2.8: Structural bind rules

wp_call

$$\frac{(F.\mathsf{inst.funcs}[i] = addri) * \xhookrightarrow{\text{FR}} F * \triangleright \left(\xhookrightarrow{\text{FR}} F \mathrel{-\!\!*} \mathsf{wp}\,[\texttt{invoke}\ addri]\,\{w, \Phi(w)\}\right)}{\mathsf{wp}\,[\texttt{call}\ i]\,\{w, \Phi(w)\}}$$

wp_call_indirect_success

$$\frac{\begin{array}{c}\xhookrightarrow{\text{FR}} F * (F.\mathsf{inst.tabs}[0] = ta) * \\ (ta \overset{\texttt{wt}}{\longmapsto}_k a) * (a \overset{\texttt{wf}}{\longmapsto} cl) * (F.\mathsf{inst.types}[i] = \mathsf{typeof}\ cl) * \\ \triangleright \left((ta \overset{\texttt{wt}}{\longmapsto}_k a) \mathrel{-\!\!*} (ta \overset{\texttt{wf}}{\longmapsto} cl) \mathrel{-\!\!*} (\xhookrightarrow{\text{FR}} F) \mathrel{-\!\!*} \mathsf{wp}\,[\texttt{invoke}\ a]\,\{w, \Phi(w)\}\right)\end{array}}{\mathsf{wp}\,[\mathbf{i32.const}\ k; \texttt{call\_indirect}\ i]\,\{w, \Phi(w)\}}$$

wp_invoke_native

$$\frac{\begin{array}{c}|vs| = |ts_1| * cl = \{(inst; ts); es\}^{\texttt{NativeCl}}_{(ts_1 \to ts_2)} * \\ F' = \{\mathsf{locs} := vs +\!\!+ \texttt{zeros}(ts); \mathsf{inst} := inst\} * i \overset{\texttt{wf}}{\longmapsto} cl * \xhookrightarrow{\text{FR}} F * \\ \triangleright \left[\begin{array}{c}(i \overset{\texttt{wf}}{\longmapsto} cl * \xhookrightarrow{\text{FR}} F) \mathrel{-\!\!*} \\ \mathsf{wp}\,\left(\texttt{local}_{|ts_2|}\{F'\}\,(\texttt{block}\,([] \to ts_2)\ es)\ \texttt{end}\right)\,\{w, \Phi(w)\}\end{array}\right]\end{array}}{\mathsf{wp}\,(vs +\!\!+ \texttt{invoke}\ i)\,\{w, \Phi(w)\}}$$

wp_invoke_host

$$\frac{\begin{array}{c}|vs| = |ts_1| * cl = \{hidx\}^{\texttt{HostCl}}_{(ts_1 \to ts_2)} * \\ i \overset{\texttt{wf}}{\longmapsto} cl * \xhookrightarrow{\text{FR}} F * \triangleright \left[\begin{array}{c}(i \overset{\texttt{wf}}{\longmapsto} cl * \xhookrightarrow{\text{FR}} F) \mathrel{-\!\!*} \\ \mathsf{wp}\,(\texttt{call\_host}\ (ts_1 \to ts_2)\ hidx\ vs)\,\{w, \Phi(w)\}\end{array}\right]\end{array}}{\mathsf{wp}\,(vs +\!\!+ [\texttt{invoke}\ i])\,\{w, \Phi(w)\}}$$

Figure 2.9: Function call rules

wp_get_local

$$\frac{F.\mathsf{locs}[i] = v * \triangleright \Phi(\texttt{immV}\ [v']) * \overset{\text{FR}}{\hookrightarrow} F}{\mathsf{wp}\,(\texttt{local.get}\ i)\left\{w, \Phi(w) * \overset{\text{FR}}{\hookrightarrow} F\right\}}$$

wp_set_local

$$\frac{i < |F.\mathsf{locs}| * \triangleright \Phi(\texttt{immV}\ []) * \overset{\text{FR}}{\hookrightarrow} F}{\mathsf{wp}\,(v; \texttt{local.set}\ i)\left\{w, \Phi(w) * \overset{\text{FR}}{\hookrightarrow} \{\mathsf{inst} := F.\mathsf{inst}; \mathsf{locs} := \langle [i := v] \rangle F.\mathsf{locs}\}\right\}}$$

wp_tee_local

$$\frac{\triangleright(\overset{\text{FR}}{\hookrightarrow} F \mathrel{-\!\!*} \mathsf{wp}\,(v; v; \texttt{local.set}\ i)\{w, \Phi(w)\}) * \overset{\text{FR}}{\hookrightarrow} F}{\mathsf{wp}\,(v; \texttt{local.tee}\ i)\{w, \Phi(w)\}}$$

wp_get_global

$$\frac{F.\mathsf{inst.globs}[i] = n * n \overset{\texttt{wg}}{\mapsto} \{\mathit{mutability}, v\} * \triangleright \Phi(\texttt{immV}\ [v]) * \overset{\text{FR}}{\hookrightarrow} F}{\mathsf{wp}\,(\texttt{global.get}\ i)\left\{w, \Phi(w) * n \overset{\texttt{wg}}{\mapsto} g * \overset{\text{FR}}{\hookrightarrow} F\right\}}$$

wp_set_global

$$\frac{F.\mathsf{inst.globs}[i] = n * n \overset{\texttt{wg}}{\mapsto} \{\texttt{mut}, -\} * \triangleright \Phi(\texttt{immV}\ []) * \overset{\text{FR}}{\hookrightarrow} F}{\mathsf{wp}\,(v; \texttt{global.set}\ i)\left\{w, \Phi(w) * n \overset{\texttt{wg}}{\mapsto} \{\texttt{mut}, v\} * \overset{\text{FR}}{\hookrightarrow} F\right\}}$$

wp_load

$$\frac{\begin{array}{c} F.\mathsf{inst.mems}[0] = n * n \overset{\texttt{wms}}{\mapsto}_{(i+\mathit{off})} bs * \\ \mathsf{deserialise}(bs) = v * \mathbf{typeof}(\mathsf{v}) = t * \triangleright \Phi(\texttt{immV}\ [v]) * \overset{\text{FR}}{\hookrightarrow} F \end{array}}{\mathsf{wp}\,(\mathbf{i32.const}\ i; \mathbf{load}\ t\ \mathit{off})\left\{w, \Phi(w) * n \overset{\texttt{wms}}{\mapsto}_{(i+\mathit{off})} bs * \overset{\text{FR}}{\hookrightarrow} F\right\}}$$

wp_store

$$\frac{\begin{array}{c} F.\mathsf{inst.mems}[0] = n * n \overset{\texttt{wms}}{\mapsto}_{(i+\mathit{off})} bs * \\ \mathbf{typeof}(\mathsf{v}) = t * |bs| = |t| * \mathsf{serialise}(v) = bv * \triangleright \Phi(\texttt{immV}\ []) * \overset{\text{FR}}{\hookrightarrow} F \end{array}}{\mathsf{wp}\,(\mathbf{i32.const}\ i; v; \mathbf{store}\ t\ \mathit{off})\left\{w, \Phi(w) * n \overset{\texttt{wms}}{\mapsto}_{(i+\mathit{off})} bv * \overset{\text{FR}}{\hookrightarrow} F\right\}}$$

wp_current_memory

$$\frac{\begin{array}{c} F.\mathsf{inst.mems}[0] = n * n \overset{\texttt{mlen}}{\mapsto} len * \lfloor len/64\mathsf{Ki} \rfloor = c * \\ \triangleright \Phi(\texttt{immV}\ [\mathbf{i32.const}\ c]) * \overset{\text{FR}}{\hookrightarrow} F \end{array}}{\mathsf{wp}\,\mathbf{current\_memory}\left\{w, \Phi(w) * n \overset{\texttt{mlen}}{\mapsto} len * \overset{\text{FR}}{\hookrightarrow} F\right\}}$$

wp_grow_memory

$$\frac{\begin{array}{c} F.\mathsf{inst.mems}[0] = n * n \overset{\texttt{mlen}}{\mapsto} len * \lfloor len/64\mathsf{Ki} \rfloor = c * \\ \triangleright \Phi(\texttt{immV}\ [\mathbf{i32.const}\ c]) * \triangleright \Psi(\texttt{immV}\ [\mathbf{i32.const}\ (-1)]) * \overset{\text{FR}}{\hookrightarrow} F \end{array}}{\mathsf{wp}\,(\mathbf{i32.const}\ c; \mathbf{grow\_memory})\left\{w, \begin{array}{c} \Big((\Phi(w) * n \overset{\texttt{mlen}}{\mapsto} (len + c \cdot 64\mathsf{Ki}) * \\ n \overset{\texttt{wms}}{\mapsto}_{len} (0\texttt{x}00)^{64\mathsf{Ki}}) \vee \\ (\Psi(w) * n \overset{\texttt{mlen}}{\mapsto} len)\Big) * \overset{\text{FR}}{\hookrightarrow} F \end{array}\right\}}$$

Figure 2.10: Resource-related rules

wp_br

$$\frac{(lh_k[vs +\!\!+ \texttt{br}\ k] = les) * (|vs| = n) * \xhookrightarrow{\text{FR}} F * \triangleright\left(\xhookrightarrow{\text{FR}} F \mathrel{-\!\!*} \texttt{wp}\,(vs +\!\!+ es)\,\{w,\Phi(w)\}\right)}{\texttt{wp}\,\texttt{label}_n\{es\}\ les\ \texttt{end}\,\{w,\Phi(w)\}}$$

wp_return

$$\frac{(lh_i[(vs +\!\!+ \texttt{return})] = les) * (|vs| = n) * \triangleright\left[\texttt{wp}\,(vs +\!\!+ es)\left\{w,\Phi(w) * \xhookrightarrow{\text{FR}} F\right\}\right]}{\texttt{wp}\,\texttt{local}_n\{F_0\}\ les\ \texttt{end}\left\{w,\Phi(w) * \xhookrightarrow{\text{FR}} F\right\}}$$

wp_block

$$\frac{|vs| = |ts_1| * \xhookrightarrow{\text{FR}} F * \triangleright\left[\xhookrightarrow{\text{FR}} F \mathrel{-\!\!*} \texttt{wp}\,\texttt{label}_{|ts_2|}\{[]\}\ (vs +\!\!+ es)\ \texttt{end}\,\{w,\Phi(w)\}\right]}{\texttt{wp}\,(vs +\!\!+ \texttt{block}\ (ts_1 \to ts_2)\ es)\,\{w,\Phi(w)\}}$$

wp_loop

$$\frac{|vs| = |ts_1| * \xhookrightarrow{\text{FR}} F * \triangleright\left[\xhookrightarrow{\text{FR}} F \mathrel{-\!\!*} \texttt{wp}\,(\texttt{label}_{|ts_1|}\{[\texttt{loop}\ (ts_1 \to ts_2)\ es]\}\ (vs +\!\!+ es)\ \texttt{end})\,\{w,\Phi(w)\}\right]}{\texttt{wp}\,(vs +\!\!+ \texttt{loop}\ (ts_1 \to ts_2)\ es)\,\{w,\Phi(w)\}}$$

wp_if_true

$$\frac{c \neq 0 * \xhookrightarrow{\text{FR}} F * \triangleright\left[\xhookrightarrow{\text{FR}} F \mathrel{-\!\!*} \texttt{wp}\,[\texttt{block}\,ft\ es_1]\,\{w,\Phi(w)\}\right]}{\texttt{wp}\,([\textbf{i32.const}\ c; \textbf{if}\ ft\ es_1\ es_2]\,\{w,\Phi(w)\}}$$

wp_if_false

$$\frac{c = 0 * \xhookrightarrow{\text{FR}} F * \triangleright\left[\xhookrightarrow{\text{FR}} F \mathrel{-\!\!*} \texttt{wp}\,[\texttt{block}\,ft\ es_2]\,\{w,\Phi(w)\}\right]}{\texttt{wp}\,([\textbf{i32.const}\ c; \textbf{if}\ ft\ es_1\ es_2]\,\{w,\Phi(w)\}}$$

wp_br_if_true

$$\frac{c \neq 0 * \xhookrightarrow{\text{FR}} F * \triangleright\left[\xhookrightarrow{\text{FR}} F \mathrel{-\!\!*} \texttt{wp}\,[\texttt{br}\ \ i]\,\{w,\Phi(w)\}\right]}{\texttt{wp}\,([\textbf{i32.const}\ c; \texttt{br\_if}\ \ i]\,\{w,\Phi(w)\}}$$

wp_br_if_false

$$\frac{c = 0 * \xhookrightarrow{\text{FR}} F * \triangleright\Phi(\texttt{immV}\ [])}{\texttt{wp}\,([\textbf{i32.const}\ c; \texttt{br\_if}\ \ i]\left\{w,\Phi(w) * \xhookrightarrow{\text{FR}} F\right\}}$$

wp_br_table

$$\frac{iss[c] = j\ * \xhookrightarrow{\text{FR}} F * \triangleright\left[\xhookrightarrow{\text{FR}} F \mathrel{-\!\!*} \texttt{wp}\,[\texttt{br}\ \ j]\,\{w,\Phi(w)\}\right]}{\texttt{wp}\,([\textbf{i32.const}\ c; \texttt{br\_table}\ iss\ i]\,\{w,\Phi(w)\}}$$

wp_br_table_length

$$\frac{|iss| \leq c * \xhookrightarrow{\text{FR}} F * \triangleright\left[\xhookrightarrow{\text{FR}} F \mathrel{-\!\!*} \texttt{wp}\,[\texttt{br}\ \ i]\,\{w,\Phi(w)\}\right]}{\texttt{wp}\,([\textbf{i32.const}\ c; \texttt{br\_table}\ iss\ i]\,\{w,\Phi(w)\}}$$

Figure 2.11: Control flow rules

2.A.2 Host Language Semantics and Proof Rules

$$\frac{(S;F;es) \hookrightarrow (S';F';es')}{(S;F;I;ms;as;(es;\delta s)) \hookrightarrow_h (S';F';I;ms;as;(es';\delta s))}\ \texttt{lifting}$$

$$\frac{llh_k[\texttt{call\_host}\ ft\ hidx\ vs] = lles \quad as[hidx] = a \quad (S;\texttt{innermostFrame}(F,llh_k);vs;a) \rightsquigarrow_a (S';es) \quad llh_k[es] = lles'}{(S;F;I;ms;as;(lles;\delta s)) \hookrightarrow_h (S';F;I;ms;as;(lles',\delta s))}\ \texttt{host action}$$

$$\frac{llh_k[\texttt{call\_host}\ ([] \to [])\ hidx\ []] = lles \quad as[hidx] = \texttt{instantiate}\ \delta \quad llh_k[[]] = lles'}{(S;F;I;ms;as;(lles;\delta s)) \hookrightarrow_h (S;F;I;ms;as;(lles',\delta :: \delta s))}\ \texttt{host action: instantiate}$$

$$\frac{\begin{array}{c}ms[vm] = m \quad I[vi_0] = \texttt{export}\ s_0\ exportdesc_0 \cdots I[vi_n] = \texttt{export}\ s_n\ exportdesc_n \\ \texttt{Instantiate}(S,m,[exportdesc_0;\cdots;exportdesc_n],((S',inst,exports),start)) \\ I' = I[vi'_0 \leftarrow exports[0]][\cdots][vi'_m \leftarrow exports[m]] \\ es = [] \ if\ start = \texttt{None} \quad es = [\texttt{invoke}\ i]\ if\ start = \texttt{Some}\ a \wedge inst.\texttt{funcs}[a] = i\end{array}}{\begin{array}{c}(S;F;I;ms;as;(vs;\texttt{inst\_decl}\ [vi_0;\cdots;vi_n]\ vm\ \ [vi'_0;\cdots;vi'_m] :: \delta s)) \\ \hookrightarrow_h (S';F;I';ms;as;(es;\delta s))\end{array}}\ \texttt{instantiation}$$

$$\frac{F.\mathsf{inst.globs}[i] = addr * S.\mathsf{globs}[addr] = \{mutability; v\}}{(S;F;I;ms;as;(vs;\texttt{get\_global}\ \ i)) \hookrightarrow_h (S;F;I;ms;as;(v :: vs;[]))}\ \texttt{get\_global}$$

Figure 2.12: Full picture of the host operational semantics. $\texttt{innermostFrame}(F, llh_k)$ looks through llh_k to find the innermost frame, and returns F if there is none.

$$(S;F;[];\texttt{nop}) \rightsquigarrow_a (S;[])\ \texttt{nop} \qquad (S;F;[\texttt{i32.const}\ i];\texttt{call\_wasm}) \rightsquigarrow_a (S;[\texttt{call}\ i])\ \texttt{call\_wasm}$$

$$(S;F;[v];\texttt{print}) \rightsquigarrow_a (S;[])\ \texttt{print}$$

$$\frac{F.\texttt{inst.funcs}[fidx] = func \qquad S' = S[\texttt{tabs} \leftarrow S.\texttt{tabs}[0][tidx \leftarrow func]]}{(S;F;[\texttt{i32.const}\ tidx;\texttt{i32.const}\ fidx];\texttt{table.set}) \rightsquigarrow_a (S';[])}\ \texttt{table.set}$$

Figure 2.13: Host actions

wp_lift_wasm

$$\frac{\mathsf{wp}\, es\, \{w, \mathsf{wp}_{\mathbf{HOST}}\, (w; \delta s)\, \{hw, \Phi(hw)\}\}}{\mathsf{wp}_{\mathbf{HOST}}\, (es; \delta s)\, \{hw, \Phi(hw)\}}$$

wp_host_action_table_set

$$\frac{\begin{array}{l} \xhookrightarrow{\text{FR}} F \;\ast\; \text{innermost\_frame}\; llh_k = F_0 \;\ast \\ F_0.\mathsf{inst.tabs}[0] = tab \;\ast\; F_0.\mathsf{inst.funcs}[fidx] = func \;\ast\; hidx \xhookrightarrow{\texttt{ha}} \texttt{table.set} \\ \ast\; tab \overset{\texttt{wt}}{\longmapsto}_{tidx} - \;\ast \\ \triangleright \left(\begin{array}{c} \left(\xhookrightarrow{\text{FR}} F \;\ast\; hidx \xhookrightarrow{\texttt{ha}} \texttt{table.set} \;\ast\; tab \overset{\texttt{wt}}{\longmapsto}_{tidx} func \right) \mathrel{-\!\!\ast} \\ \mathsf{wp}_{\mathbf{HOST}}\, (\delta s; llh_k[[]])\, \{v, \Phi(v)\} \end{array} \right) \end{array}}{\mathsf{wp}_{\mathbf{HOST}}\, (\delta s; llh_k[\texttt{call\_host}\ ([\mathbf{i32}; \mathbf{i32}] \to [])\ hidx\ \left[\begin{array}{l} \mathbf{i32.const}\ tidx; \\ \mathbf{i32.const}\ fidx \end{array} \right]])\, \{v, \Phi(v)\}}$$

wp_host_action_instantiate

$$\frac{\begin{array}{l} hidx \xhookrightarrow{\texttt{ha}} \texttt{instantiate}\ \delta \;\ast \\ \triangleright\, (hidx \xhookrightarrow{\texttt{ha}} \texttt{instantiate}\ \delta \mathrel{-\!\!\ast} \mathsf{wp}_{\mathbf{HOST}}\, (\delta :: \delta s; llh_k[[]])\, \{v, \Phi(v)\}) \end{array}}{\mathsf{wp}_{\mathbf{HOST}}\, (\delta s; llh_k[\texttt{call\_host}\ ([] \to [])\ hidx\ []])\, \{v, \Phi(v)\}}$$

wp_host_action_no_state_change

$$\frac{\begin{array}{l} hidx \xhookrightarrow{\texttt{ha}} f \;\ast\; (\forall S\, F, (S; F; vs; f) \rightsquigarrow_a (S; res)) \;\ast \\ \triangleright\, (hidx \xhookrightarrow{\texttt{ha}} f \mathrel{-\!\!\ast} \mathsf{wp}_{\mathbf{HOST}}\, (\delta s; llh_k[res])\, \{w, \Phi(w)\}) \end{array}}{\mathsf{wp}_{\mathbf{HOST}}\, (\delta s; llh_k[\texttt{call\_host}\ tf\ hidx\ vs])\, \{w, \Phi(w)\}}$$

Figure 2.14: Host program logic rules (Part 1)

wp_host_instantiate

$$\frac{\vdash m : timps \to texps \;*\; \texttt{constInits}(m) \;*\; \texttt{resourcesImports}(m, imports, timps, wfs, wts, wms, wgs) \;*\; vm \xhookrightarrow{\texttt{mod}} m \;*\; \ast_{\substack{(vi,imp) \in \\ (vis_i, imports)}} vi \overset{\texttt{vis}}{\mapsto} imp \;*\; \ast_{vi \in vis_e} vi \overset{\texttt{vis}}{\mapsto} -}{\texttt{wp}_{\mathbf{HOST}}\,(vs; [\texttt{inst\_decl}\ vis_i\ vm\ vis_e])\left\{ \begin{array}{c} \_,\ vm \xhookrightarrow{\texttt{mod}} m \;* \\ \ast_{\substack{(vi,imp) \in \\ (vis_i, imports)}} vi \overset{\texttt{vis}}{\mapsto} imp \ast \\ \exists inst, \texttt{resources} \\ \left(\begin{array}{c} m, imports, timps, wfs, wts, \\ wms, wgs, m.\texttt{start}, inst \end{array} \right) * \\ \ast_{\substack{(vi,exp) \in \\ (vis_e, m.\texttt{exports})}} vi \overset{\texttt{vis}}{\mapsto} \\ \texttt{constructExport}(exp, inst) \end{array} \right\}}$$

WP_HOST_GET_GLOBAL

$$\frac{i \overset{\texttt{wg}}{\mapsto} \{mutability, v\} * \Phi(\texttt{immV}\ (v :: vs))}{\texttt{wp}_{\mathbf{HOST}}\,([\texttt{get\_global}\ i], vs)\left\{w, \Phi(w) * i \overset{\texttt{wg}}{\mapsto} \{mutability, v\}\right\}}$$

WP_HOST_GET_GLOBAL_TRAP

$$\frac{\Phi(\texttt{trapV})}{\texttt{wp}_{\mathbf{HOST}}\,([\texttt{get\_global}\ i], [\texttt{trap}])\,\{w, \Phi(w)\}}$$

Figure 2.15: Host program logic rules (Part 2)

2.A.3 Logical Resources

Points-to predicates for components of the WebAssembly store

$i \overset{\texttt{wf}}{\mapsto} cl$	Ownership of a function closure
$i \overset{\texttt{wt}}{\mapsto}_{idx} addr$	Ownership of a function address in the table
$i \overset{\texttt{tlen}}{\hookrightarrow} n$	Ownership of the length of a table
$i \overset{\texttt{tlim}}{\hookrightarrow} \texttt{option}\ n$	Ownership of the maximum length of a table
$i \overset{\texttt{wm}}{\mapsto}_{addr} b$	Ownership of a byte in memory
$i \overset{\texttt{wms}}{\mapsto}_{addr} bv$	Ownership of a list of bytes in memory
$i \overset{\texttt{mlen}}{\mapsto} n$	Ownership of size of a memory
$i \overset{\texttt{wg}}{\mapsto} \{mutability; v\}$	Ownership of a global variable

Points-to predicate for ownership of the WebAssembly frame

$$\overset{\text{FR}}{\hookrightarrow} F$$

Points-to predicates for components of the host state

$vi \overset{\texttt{vis}}{\mapsto} export$	Ownership of an import value store
$vm \overset{\texttt{mod}}{\hookrightarrow} m$	Ownership of a module
$hidx \overset{\texttt{ha}}{\hookrightarrow} a$	Ownership of a host action

Figure 2.16: The points-to assertions corresponding to the various components of the state

2.A.4 Instantiation Lemma

Lemma 2 (Module Instantiation Resource Allocation)**.**

$$\begin{array}{ll} \textit{If} & \vdash m : timps \to texps \ \ \wedge \ \ \texttt{constInits}(m) \\ \textit{and} & \texttt{Instantiate}(S, m, imports, ((S', inst, exports), start)) \\ \textit{then} & \texttt{resourcesImports}(m, imports, timps, wfs, wts, wms, wgs) \ * \\ & \quad \texttt{stateInterp}(S) \\ & \quad\quad \Rrightarrow \texttt{resources}(m, imports, timps, wfs, wts, wms, wgs, start, inst) \ * \\ & \quad\quad\quad \texttt{stateInterp}(S') \end{array}$$

$\texttt{Instantiate}(S, m, imports, ((S', inst, exports), start))$ is the full instantiation predicate in the WebAssembly 1.0 specification, specifying the instantiation giving a relation between the instantiation result and the input parameters. Its definition consists of multiple components, which we only briefly describe here, and refer to the official specification for the detailed definition. The `Instantiate` predicate does the following:

1. Asserts that the module is well-typed with respect to the module imports provided, which were looked up from the store S at addresses specified by *imports*;

2. Allocate into store S, in order, the functions, tables, memories and globals declared in the module, by converting the static declarations into runtime representations and append to the corresponding components of the store. During the procedure, create module *instance* record that collects all the addresses of the allocated WebAssembly states. Each component of the instance if prepended by the addresses of the module imports. The resulting instance will be *inst* in the result tuple;

3. For each module export declared by m, lookup in *inst* the addresses of them, and create a module export with export name as specified by the export declaration itself and export description being the address from the above lookup. This list of all module exports will be *exports* in the result tuple;

4. If the module declares a start function, looks up the correct address of it from the instance *inst* and put it into *start* of the resulting tuple. Otherwise, *start* will be empty;

5. Let S_1 be the store resulting from the above. For each elem segment of the module, check if its offset and size are within the bound of its targeted table. If this succeeds, initializes the corresponding table segment with a list of function references specified by the elem segment. Otherwise, the instantiate fails. Formally, this is expressed in stating that no resulting tuple of $(S', inst, exports, start)$ will be able to satisfy the `Instantiate` relation. Note that this has to be done in runtime, as the size of imported tables and memories are unknown prior to instantiation.

6. Let S_2 be the store resulting from the above elem segment initialization. Now for each data segment, perform the similar set of checks and initialization to the targeted memories likewise. This may also fail in the same way as above.

7. Let S' be the resulting store from above data segment initialization, which is the resulting store in the result tuple.

The `resourceImports` characterises the resources required to perform an instantiation:

$$\texttt{resourcesImports}(m, imports, timps, wfs, wts, wms, wgs) \triangleq \left(\begin{array}{l} \texttt{functionImports}(imports, timps, wfs) * \\ \texttt{tableImports}(imports, timps, wts) * \\ \texttt{memoryImports}(imports, timps, wms) * \\ \texttt{globalImports}(imports, timps, wgs) * \\ \texttt{elemBoundCheck}(wts, imports, m) * \\ \texttt{dataBoundCheck}(wms, imports, m) * \\ |import| = |timps| \end{array} \right)$$

where

$$\texttt{functionImports}(imports, timps, wfs) \triangleq \left(\begin{array}{l} \ast_{n \in \mathrm{dom}(wfs)}\, n \xmapsto{\texttt{wf}} wfs[n] * \\ \mathrm{dom}(wfs) = \{k \mid \texttt{func}_\texttt{e}\ k \in \mathsf{exportdesc} \cdot imports\} * \\ \ast_{i<|imports|} \left[(imports[i].\mathsf{exportdesc} = \texttt{func}_\texttt{e}\ k) \mathrel{-\!\!*} \right. \\ \left. (\exists cl, wfs[k] = cl \wedge \texttt{type\_agree}(cl, timps[i])) \right] \end{array} \right)$$

The exportdesc field is looked up from *imports*, as the imports were provided by the host instantiate declaration, which came from some *exports* produced by other modules.

`tableImports`, `memoryImports` and `globalmports` are defined similarly.

$\texttt{elemBoundCheck}(wts, imports, m)$ and $\texttt{dataBoundCheck}(wms, imports, m)$ respectively contain conditions that ensure all the each element and data segments are not out of bound of the import tables and memories.

The `resources` predicate characterises the resulting state after instantiation:

$$\texttt{resources}(m, imports, timps, wfs, wts, wms, wgs, start, inst) \triangleq \left(\begin{array}{l} \exists\ g_{inits}\ t_{allocs}\ m_{allocs}\ g_{allocs}\ wts'\ wms', \\ wts' = \texttt{module\_import\_init\_tabs}(m, inst, wts) * \\ wms' = \texttt{module\_import\_init\_mems}(m, inst, wms) * \\ t_{allocs} = \texttt{module\_inst\_build\_tables}(m, inst) * \\ m_{allocs} = \texttt{module\_inst\_build\_mems}(m, inst) * \\ g_{allocs} = \texttt{module\_inst\_global\_init} \\ \quad (\texttt{module\_inst\_build\_globals}(m, g_{inits}) * \\ module\_glob\_init\_values(m, g_{inits}) * \\ instance\_import\_agree(imports, inst) * \\ start\_agree(m, inst, start) * \\ \texttt{module\_inst\_resources\_wasm}(m, inst, t_{allocs}, m_{allocs}, g_{allocs}) * \\ \texttt{functionImports}(imports, timps, wfs) * \\ \texttt{tableImports}(imports, timps, wts') * \\ \texttt{memoryImports}(imports, timps, wms') * \\ \texttt{globalImports}(imports, timps, wgs) * \\ |import| = |timps| \end{array} \right)$$

We explain its components briefly:

- The list of existential variables at the start is merely a specification device; they are decided in the later predicates completely deterministically;
- `module_import_init_tabs` and `module_import_init_mems` are pure assertions that evaluate the effect of elem and data initializer on the imported tables and memories. The resulting expected contents are obtained in wts' and wms';
- `module_inst_build_tables`, `module_inst_build_mems`, and `module_inst_build_globals` are pure assertions that evaluate the expected content of the newly allocated tables, memories, and globals after the effect of initializers. Note that functions do not need to be computed separately, since once the function closure is evaluated, there is no corresponding initializer that could further modify it in the store.
- `module_global_init_values`(m, g_{inits}) evaluates the result of the global initializer expressions declared in the module. This corresponds to the specification device in the official specification to break the ostensible circularity in the definition of the `instantiate` predicate.
- `instance_import_agree` is a pure assertion stating that each component of the instance correctly starts with the import addresses, as required by the official specification. Note that we do not constrain the addresses in which the allocated WebAssembly entities created by instantiation (aside from that they are different from the existing ones since we're using separation logic). This is a deliberate design to make the concrete addresses of these allocated entities abstract.
- `start_agree` is a pure assertion stating that the created *start* variable contains the correct address for the start function of the module, as given by the *instance*.
- `module_inst_resources_wasm` is the resource assertion that include all the points-to assertions for the allocated WebAssembly resources in the store, stating that we have the expected allocated resources (as evaluated above) at the corresponding locations given by the instance *inst*.
- The last few resources predicates are the same as that from the import, although the tables and memory predicates take the update variables wts' and wms', describing the potentially updated imported tables and memories after instantiation.

2.B Logical Relation

$\boxed{\mathscr{V}[\![ts]\!] : LogVal \to iProp}$

$$
\begin{array}{rcl}
\mathscr{V}_0[\![t]\!](v) & \triangleq & \exists c, v = t.\texttt{const}\ c \\
\mathscr{V}[\![[t_1, \cdots, t_n]]\!](w) & \triangleq & w = \texttt{trapV} \lor \\
 & & \exists v_1, \cdots, v_n, w = \texttt{immV}\ \ [v_1, \cdots, v_n] \land \\
 & & \mathscr{V}_0[\![t_1]\!](v_1) \land \cdots \land \mathscr{V}_0[\![t_n]\!](v_n)
\end{array}
$$

Figure 2.17: A logical relation for values

$\boxed{\mathscr{C}los[\![ts \to ts']\!]_{hfs} : Closure \to iProp}$

$$\mathscr{C}los[\![ts \to ts']\!]_{hfs}\big(\{(inst, tlocs); e\}^{\texttt{NativeCl}}_{ts \to ts'}\big) \triangleq$$
$$\Box \forall vs, f, \mathscr{V}[\![ts]\!](\texttt{immV}\ vs) * [\text{NaInv} : \top] * \xhookrightarrow{\text{FR}} F \mathrel{-\!\!*}$$
$$\mathscr{E}[\![ts']\!]_{(F,hfs)}(\mathbf{local}_{\texttt{len}(ts')}\{inst; vs +\!\!+ \texttt{zeros}(tlocs)\}$$
$$[\mathbf{label}_{\texttt{len}(ts')}\{\varepsilon\}\ e\ \mathbf{end}]\ \mathbf{end})$$

$$\mathscr{C}los[\![ts \to ts']\!]_{hfs}\big(\{h\}^{\texttt{HostCl}}_{ts \to ts'}\big) \triangleq (h, ts \to ts') \in hfs$$

$\boxed{\mathscr{F}unc[\![ts \to ts']\!]_{hfs} : \mathbb{N} \to iProp}$

$$\mathscr{F}unc[\![ts \to ts']\!]_{hfs}(n) \triangleq \exists cl, \boxed{n \overset{\texttt{wf}}{\longmapsto} cl}^{\mathscr{N}_{wf}.n}_{\text{NaInv}} * \triangleright \mathscr{C}los[\![ts \to ts']\!]_{hfs}(cl)$$

$\boxed{\mathscr{T}able[\![\{min; max\}]\!] : \mathbb{N} \to iProp}$

$$\mathscr{T}able[\![\{min; max\}]\!](t) \triangleq \exists n, m, t \xhookrightarrow{\texttt{tlim}} m * \ m \leq max * \ t \xhookrightarrow{\texttt{tlen}} n *$$
$$\ast_{i \in [0..n]} \exists \tau s, \eta s, fe, \boxed{t \overset{\texttt{wt}}{\longmapsto}_i fe}^{\mathscr{N}_{wt}.(t,i)}_{\text{NaInv}} *$$
$$\mathscr{F}unc[\![\tau s \to \eta s]\!](fe)$$

$\boxed{\mathscr{M}em[\![\{min; max\}]\!] : \mathbb{N} \to iProp}$

$$\mathscr{M}em[\![\{min; max\}]\!](n) \triangleq \exists (mem : \texttt{memory}),$$
$$\boxed{\begin{array}{l} n \overset{\texttt{mlen}}{\longmapsto} \texttt{MemLen}(mem) \\ \ast_{i \mapsto b \in \texttt{MemData}(mem)} n \overset{\texttt{wm}}{\longmapsto}_i b \end{array}}^{\mathscr{N}_{wm}.n}_{\text{NaInv}} *$$
$$min \leq \texttt{MemLen}(mem) \wedge max = \texttt{MemMax}(mem)$$

$\boxed{\mathscr{G}lob[\![\{\mu; \tau\}]\!] : \mathbb{N} \to iProp}$

$$\mathscr{G}lob[\![\{\mathbf{mut}; \tau\}]\!](n) \triangleq \boxed{\exists w, n \overset{\texttt{wg}}{\longmapsto} \{\mathbf{mut}; w\} * \mathscr{V}_0[\![\tau]\!](w)}^{\mathscr{N}_{wg}.n}_{\text{NaInv}}$$
$$\mathscr{G}lob[\![\{\mathbf{immut}; \tau\}]\!](n) \triangleq \exists (P : \texttt{valuetype} \to \texttt{value} \to iProp),$$
$$(\Box \forall w, P(\tau)(w) \mathrel{-\!\!*} \mathscr{V}_0[\![\tau]\!](w)) *$$
$$\boxed{\exists w, n \overset{\texttt{wg}}{\longmapsto} \{\mathbf{immut}; w\} * P[\![\tau]\!](w)}^{\mathscr{N}_{wg}.n}_{\text{NaInv}}$$

Figure 2.18: Our logical relation for instance elements

$$\boxed{\mathscr{I}[\![C]\!]_{hfs} : Instance \to iProp}$$

$$\mathscr{I}\left[\!\!\left[\left\{\begin{array}{l}\mathsf{types} = ts,\ \ \mathsf{func} = fts,\\ \mathsf{global} = gts,\ \ \mathsf{table} = [tt,\cdots],\\ \mathsf{memory} = [mt,\cdots],\\ \mathsf{locals},\mathsf{labels},\mathsf{return} = \cdots\end{array}\right\}\right]\!\!\right]_{hfs}\left(\left\{\begin{array}{l}\mathsf{types} = ts',\\ \mathsf{funcs} = fs,\\ \mathsf{globs} = gs,\\ \mathsf{tabs} = [t,\cdots],\\ \mathsf{mems} = [m,\cdots]\end{array}\right\}\right) \triangleq$$
$$ts = ts' * \mathop{\ast}_{f\in fs; ft\in tfs} \mathscr{F}unc[\![ft]\!]_{hfs}(f) *$$
$$\mathop{\ast}_{g\in gs; gt\in gts} \mathscr{G}lob[\![gt]\!](g) * \mathscr{T}able[\![tt]\!](t) * \mathscr{M}em[\![mt]\!](m)$$

$$\boxed{\mathscr{C}tx[\![C]\!]_{(inst,hfs)} : Lholed \to iProp}$$

$$\mathscr{C}tx[\![\{\cdots;\tau l;\tau_{\mathtt{lbs}};\tau_{\mathtt{ret}}\}]\!]_{(inst,hfs)}(lh) \triangleq \mathtt{StructuralCond}(\tau_{\mathtt{lbs}}, lh) * \mathop{\ast}_{j\mapsto ts\in\tau_{\mathtt{lbs}}} \mathscr{K}[\![ts]\!]^{\tau_{\mathtt{lbs}},\tau_{\mathtt{ret}}}_{(\tau l,inst,hfs)}(lh,j)$$

Figure 2.19: Our logical relation for instances

$\boxed{\mathscr{R}et[\![ts]\!]_{(\tau l,inst)} : Val \to iProp}$

$$\begin{array}{rcl}\mathscr{R}et[\![ts]\!]_{(\tau l,inst)}(w) & \triangleq & \exists lh, v, w = lh[\mathbf{return}] * lh(0) = v * \\ & & \exists \tau s', \mathscr{V}[\![\tau s']\!](v) * \\ & & \forall F, F', \xhookrightarrow{\text{FR}} F' \mathrel{-\!\!*} \\ & & \mathtt{wp}\,[\mathbf{local}_n\{F\}\ w\ \mathbf{end}]\left\{w, \mathscr{V}[\![ts]\!](w) * \xhookrightarrow{\text{FR}} F'\right\}\end{array}$$

$\boxed{\mathscr{B}r[\![[ts_1;\cdots;ts_l]]\!]^{\tau_{ret}}_{(\tau l,inst,hfs)} : Val \times Lholed \to iProp}$

$$\begin{array}{l}\mathscr{B}r[\![[ts_1;\cdots;ts_l]]\!]^{\tau_{ret}}_{(\tau l,inst,hfs)}(w, lh_n) \triangleq \\ \quad \exists lh'''_p, j, w = lh'''_p[\mathbf{br}\ \mathtt{j}] * \\ \quad \exists vs', vs, k, es, lh'_m, es', lh''_{n-S(j-p)}, ts', \\ \quad lh_n(n - S(j-p)) = (vs, k, es, lh'_m, es') * \\ \quad \mathtt{outer}_{n-S(j-p)}(lh_n) = lh''_{n-S(j-p)} * \\ \quad lh'''_p(0) = (vs', -) * \\ \quad \mathscr{V}[\![ts' +\!\!+ ts_{j-p}]\!](vs') * \\ \quad \forall F, \mathscr{F}rame[\![\tau l]\!]_{inst}(F) \mathrel{-\!\!*} \\ \quad \mathtt{wp}\,(vs +\!\!+ \mathtt{label}_k\{es\}\ lh'_m\ \mathtt{end} +\!\!+ es')[\mathtt{drop}\ (\mathtt{len}(ts'))\ vs' +\!\!+ [\mathtt{br}\ (j-p)]] \\ \quad \left\{ w,\ \begin{array}{l}\left(\begin{array}{l}(\exists \tau s, \mathscr{V}[\![\tau s]\!](w)) \vee \\ \left(\exists \tau s, \mathscr{H}[\![\tau s]\!]^{[ts_{S(S(j-p))},\cdots,ts_l],\tau_{ret}}_{(\tau l,inst,hfs)}(w)\right) \vee \\ \triangleright \mathscr{B}r[\![[ts_{S(S(j-p))},\cdots,ts_l]]\!]^{\tau_{ret}}_{(\tau l,inst,hfs)}\left(w, lh''_{n-S(j-p)}\right) \vee \\ \mathscr{R}et[\![\tau_{ret}]\!]_{(\tau l,inst)}(w)\end{array}\right) * \\ \exists F, \mathscr{F}rame[\![\tau l]\!]_{inst}(F)\end{array} \right\}\end{array}$$

Figure 2.20: Our logical relation for control flow (Part 1)

$$\boxed{\mathscr{K}[\![\tau s]\!]^{\tau_{\mathtt{lbs}},\tau_{\mathtt{ret}}}_{(\tau l,inst,hfs)} : Lholed \times \mathbb{N} \to iProp}$$

$$\begin{array}{l}\mathscr{K}[\![\tau s]\!]^{\tau_{\mathtt{lbs}},\tau_{\mathtt{ret}}}_{(\tau l,inst,hfs)}(lh_n, j) \triangleq \\ \quad \exists vs, k, es, lh'_m, es', lh''_{n-S(j)}, \\ \qquad lh_n(n-S(j)) = (vs,k,es,lh',es') * \\ \qquad \mathtt{outer}_{n-S(j)}(lh_n) = lh_{n-S(j)} * \\ \qquad \square \forall v, F, \mathscr{V}[\![\tau s]\!](v) \mathrel{-\!\!*} \\ \qquad\quad \mathscr{F}rame[\![\tau l]\!]_{inst}(F) \mathrel{-\!\!*} \\ \qquad\qquad \exists \tau s_2, \mathscr{E}[\![\tau s_2]\!]^{\mathtt{drop}\ S(j)\ \tau_{\mathtt{lbs}},\tau_{\mathtt{ret}}}_{\tau l,inst,hfs)} \left(lh''_{n-S(j)}, vs +\!\!+ v +\!\!+ es +\!\!+ es'\right)\end{array}$$

$$\boxed{\mathscr{H}[\![ts]\!]^{\tau_{\mathtt{lbs}},\tau_{\mathtt{ret}}}_{(\tau l,inst,hfs)} : Val \times Lholed \to iProp}$$

$$\begin{array}{lll}\mathscr{H}[\![ts]\!]^{\tau_{\mathtt{lbs}},\tau_{\mathtt{ret}}}_{(\tau l,inst,hfs)}(w, lh) & \triangleq & \exists llh, vs, ft, hidx, ts_1, ts_2, \\ & & \quad w = \mathtt{call\_hostV}\ ft\ hidx\ vs\ llh * \\ & & \quad ft = ts_1 \to ts_2 * \\ & & \quad (hidx, ft) \in hfs * \\ & & \quad \mathtt{allBasicInstr}(llh) * \\ & & \quad \mathscr{V}[\![ts_1]\!](vs) * \\ & & \quad \square\ \forall vs', F, \left(\begin{array}{l} \mathscr{V}[\![ts_2]\!](vs') \mathrel{-\!\!*} \\ \quad \mathscr{F}rame[\![\tau l]\!]_{inst}(F) \mathrel{-\!\!*} \\ \quad \mathscr{E}[\![ts]\!]^{\tau_{\mathtt{lbs}},\tau_{\mathtt{ret}}}_{(\tau l,inst,hfs)}(lh, llh[vs']) \end{array}\right)\end{array}$$

Figure 2.21: Our logical relation for control flow (Part 2)

$$\boxed{\mathscr{F}rame[\![ts]\!]_{inst} : Frame \to iProp}$$

$$\mathscr{F}rame[\![ts]\!]_{inst}(F) \quad \triangleq \quad [\mathrm{NaInv} : \top] * \xhookrightarrow{\mathrm{FR}} F * \exists vs, F = \{inst; vs\} * \ \mathscr{V}[\![ts]\!](\mathtt{immV}\ vs)$$

$$\boxed{\mathscr{E}_0[\![ts]\!]_* : Expr \to iProp} \qquad \boxed{\mathscr{E}[\![ts]\!]^*_* : Lholed \times Expr \to iProp}$$

$$\begin{array}{rcl}\mathscr{E}_0[\![ts]\!]_{(F,hfs)}(es) & \triangleq & \mathsf{wp}\, es \left\{ w,\ \begin{array}{l}(\mathscr{V}[\![ts]\!](w) \vee \mathscr{H}[\![ts]\!]_{hfs}(w)) * \\ [\mathrm{NaInv} : \top] * \xhookrightarrow{\mathrm{FR}} F\end{array} \right\} \\ \mathscr{E}[\![ts]\!]^{\tau_{\mathtt{lbs}},\tau_{\mathtt{ret}}}_{(\tau l,inst,hfs)}(lh, es) & \triangleq & \mathsf{wp}\, es \left\{ w,\ \begin{array}{l}\left(\begin{array}{l}\mathscr{V}[\![ts]\!](w) \vee \mathscr{H}[\![ts]\!]^{\tau_{\mathtt{lbs}},\tau_{\mathtt{ret}}}_{(\tau l,inst,hfs)}(w) \\ \vee \mathscr{B}r[\![\tau_{\mathtt{lbs}}]\!]^{\tau_{\mathtt{ret}}}_{(\tau l,inst,hfs)}(w, lh) \\ \vee \mathscr{R}et[\![\tau_{\mathtt{ret}}]\!]_{(\tau l,inst)}(w)\end{array}\right) * \\ \exists F, \mathscr{F}rame[\![\tau l]\!]_{inst}(F)\end{array} \right\}\end{array}$$

Figure 2.22: Our logical relation for frames and expressions

2.C Case Study: Landin's Knot

```
1 m_factorial ≜
2 (module ;; Factorial Module
3  (import "host" "g" (global $g i32))
4  (import "host" "mut" (func $mut (param i32 i32)))
5  (table 1 funcref)
6  (func $setup (param $f i32)
7   const 0; local.get $f; call $mut)
8  (func $F (param $n i32) (result i32)
9   local.get $n; ... )
10 (func $fact (param $n i32) (result i32)
11  local.get $n; const $F; call $setup; const 0; call_indirect)
12 (func $main
13  global.get $g; call $fact; global.set $g))
14
```

Figure 2.23: Factorial module implemented using Landin's Knot

Fig. 2.23 depicts a factorial function implemented using recursion through the store; the so-called Landin's Knot. Rather than doing a recursive call, the body of the factorial function $F does an indirect call to some unspecified function in the function table. The knot is tied by storing a reference to $F into the function table, prior to invoking it. The idea is to dynamically update the function table, to allow for multiple recursive functions, all using recursion through the store. Since WebAssembly programs cannot mutate function tables, this is done by invoking an imported function from the host. The specification of the function shows that the imported function is indeed the modify table host action.

The specification can be phrased as a host triple around a host that instantiates the factorial module, and invokes its main function, which in turn calls $fact on the imported global variable $g. The postcondition asserts that the returned value, passed through $g, matches the result of the mathematical factorial operation. Note that we refer to imports and modules via names rather than indices, for the sake of readability.

The precondition details the imports of the factorial module, namely the global variable $g, and the host closure h. Additionally, the host closure must refer specifically to the `table.set` host action.

Theorem 4 (Factorial Module Specification)**.**

$$\left\{\begin{array}{l}\texttt{"fact"} \overset{\texttt{mod}}{\hookrightarrow} m_{factorial} * \texttt{"g"} \overset{\texttt{vis}}{\mapsto} \$g * \texttt{"mut"} \overset{\texttt{vis}}{\mapsto} h * \\ \$g \overset{\texttt{wg}}{\mapsto} n * [NaInv : \top] * h \overset{\texttt{wf}}{\longmapsto} \{a\}^{\texttt{HostCl}}_{[\texttt{i32};\texttt{i32}]\to[]} * a \overset{\texttt{ha}}{\hookrightarrow} \texttt{table.set}\end{array}\right\}$$
$$\texttt{inst\_decl}\,[\texttt{"mut"};\texttt{"g"}]\ \texttt{"fact"}\ []$$
$$\left\{hw,\ hw = ([],[]) * \$g \overset{\texttt{wg}}{\mapsto} n!\right\}$$

Proof. The specification is proved by interweaving the host and WebAssembly program logics. The pattern for host calls is applied to reason about the host call in the `$setup` function. □

Chapter 3

Iris-MSWasm: Elucidating and Mechanising the Security Invariants of Memory-Safe WebAssembly

This chapter consists of the following paper:

Maxime Legoupil, June Rousseau, Aïna Linn Georges, Jean Pichon-Pharabod, Lars Birkedal

Iris-MSWasm: Elucidating and Mechanising the Security Invariants of Memory-Safe WebAssembly

Proceedings of the ACM on Programming Languages (OOPSLA), 2024

It has been reformatted to fit this document's page size.

Abstract

WebAssembly offers coarse-grained encapsulation guarantees via its module system, but does not support fine-grained sharing of its linear memory. MSWasm is a recent proposal which extends WebAssembly with fine-grained memory sharing via handles, a type of capability that guarantees spatial and temporal safety, and thus enables an expressive yet safe style of programming with flexible sharing. In this paper, we formally validate the pen-and-paper design of MSWasm. To do so, we first define MSWasmCert, a mechanisation of MSWasm that makes it a fully-defined, conservative extension of WebAssembly 1.0, including the module system. We then develop Iris-MSWasm, a foundational reasoning framework for MSWasm composed of a separation logic to reason about known code, and a logical relation to reason about unknown, potentially adversarial code. Iris-MSWasm thereby makes explicit a key aspect of the implicit universal contract of MSWasm: robust capability safety. We apply Iris-MSWasm to reason about key use cases of handles, in which the effect of calling

an unknown function is bounded by robust capability safety. Iris-MSWasm thus works as a framework to prove complex security properties of MSWasm programs, and provides a foundation to evaluate the language-level guarantees of MSWasm.

3.1 Introduction

WebAssembly (abbreviated Wasm) is the current industry standard to run applications efficiently in the browser [41], and is increasingly adopted in cloud computing (for example, Fastly's Compute@Edge [26, 43] and Fermyon's Spin [12]), in part thanks to its well-defined semantics and the high-performance implementations it enables. To rise up to the stringent security requirements of the web, Wasm promises not only sandboxing, but also several language-level security guarantees, including control flow integrity and coarse-grained memory safety at the level of its units of code distribution, *modules*. Each module can define a *linear memory* (or several, in Wasm 2.0), which is private by default, but which the module can explicitly export. In that case, any other module can import it, and thereby access it unrestrictedly. This unusually strong encapsulation guarantee that a non-exported memory cannot be affected by other modules [89] makes edge computing practical and lightweight [16]: one can safely compose a module with untrusted, potentially adversarial library modules to perform tasks (image compression, etc.) on separate memories. However, sharing is an all-or-nothing affair: a linear memory is either completely private, or all of it is shared with every module. As pointed out by Lehmann et al. [61], this means that many of the classical attacks against memory unsafe languages, targeting a finer granularity, also work against programs that are not specifically written to take advantage of module isolation of WebAssembly.

Thus, to take advantage of the memory isolation guarantees of Wasm, programs require either invasive changes to fit WebAssembly's module system even though programs are typically not written directly in WebAssembly, or rely on extensive copying (which is the approach taken by the Component Model [122, 123]).

To address this lack of flexibility, Disselkoen et al. [24] and Michael et al. [69] propose Memory-Safe WebAssembly (abbreviated MSWasm), a conservative extension of WebAssembly with a mechanism for fine-grained memory sharing in the form of capabilities [21, 134], which it calls *handles*, and which embody authority over ranges of a new kind of memory: *segment memory*. This design is inspired by the capability-enhanced CHERI hardware architecture [135], which has been shown to be targetable from C with lightweight code changes by relying on reasonable patches to production compilers [68, 143]. The expectation is that MSWasm programs respect much finer memory safety invariants than plain Wasm. However, as illustrated during the development of the CHERI capability hardware architecture, these security invariants are very brittle: a mistake in a single detail can invalidate all encapsulation guarantees [5, 78], and prose specifications backed by mere testing do not provide the required level of assurance.

Contributions In this paper, we complete the pen-and-paper definition of MSWasm to be a conservative extension of WebAssembly 1.0, and mechanise it in the Coq proof assistant as MSWasmCert, building on WasmCert [133]. On top of this precise language definition, we develop Iris-MSWasm, a program logic that extends Iris-Wasm [89] with capability reasoning. Using the assertion language of Iris-MSWasm, we formulate an unstated yet key part of the *universal contract* [127] of MSWasm: that all instructions respect *robust capability safety*. Robust capability safety, as demonstrated for object capabilities [22, 120] and capability hardware architectures [32–36, 114, 116–118], makes it tractable to reason about the combination of known code with unknown, potentially adversarial code. As such, it refines the original *memory safety* guarantee of MSWasm, which does not directly lend itself to prove integrity properties of local state.

With our definition in hand, we identify cases where the original prose description is imprecise, as well as a handful of minor typos. We then show that MSWasm satisfies robust capability safety, and illustrate it on key representative examples capturing fine-grained memory invariants, thereby validating the design of MSWasm to the level of rigour that it deserves. To our knowledge, this is the first proof of robust capability safety for an industrial language, and for a language of this size. Moreover, because our formulation of robust capability safety captures the behaviour of an arbitrary MSWasm module given the exports that the module has access to, we expect that it can be used to reason about the combination of WebAssembly code compiled from a higher-level language with unknown code compiled to MSWasm.

In showing robust capability safety for a complete definition of MSWasm, we make the case that, in addition to the extensional behaviour of a formally defined operational semantics, *industrial-scale language definitions can and therefore should come with a formally stated universal contract backed by a machine-checked proof.*

In summary, our contributions are:

- MSWasmCert, a mechanised language definition of MSWasm.
- Iris-MSWasm, a mechanised program logic covering all the language constructs of MSWasm, and with a complete proof of soundness.
- A mechanised statement and proof of robust capability safety using a logical relation.

All the technical results have been proved in the Coq proof assistant, and our Coq development is available online (see Data Availability Statement).

Outline In the rest of this section, we present capabilities/handles, illustrate their use on a running example (§3.1.1), and describe the attacker model that we consider (§3.1.2). We then present our precise semantics of MSWasm (§3.2), focusing on the differences to plain WebAssembly, and highlight how we complete the original prose semantics. We then describe our program logic and its assertion language (§3.3), which we then use for the main contribution of this paper: the definition and proof

of robust capability safety of MSWasm (§3.4). We illustrate this property on a larger example (§3.5), and we finish with a discussion (§3.6).

3.1.1 Introduction to MSWasm via a Running Example

We illustrate MSWasm on the classic capability 'buffer' example [136], adapted to our setting. We give the code (using the formal syntax we present later in Figures 3.2 and 3.3) and depict it visually in Fig. 3.1, and describe it informally below.

```
0 [ i32.const 8;                 } Allocate a handle
1   segalloc;                    } with room for two i32s
2   local.set $h;                  Call it $h
3   local.get $h;                }
4   i32.const 42;                } Store private value 42
5   i32.segstore;                } at h[0..3]
6   local.get $h;                }
7   i32.const 4;                 } Create a sub-handle for
8   i32.const 4;                 } the rest of the buffer, h[4..]
9   slice;                       }
10  local.set $hpub;               Call it $hpub
11  local.get $hpub;             } Call adversary function $adv
12  call $adv;                   } with argument $hpub
13  local.get $h;                }
14  i32.segload     ]            } Read from $h
```

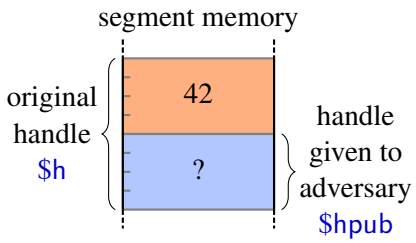


Figure 3.1: The buffer example

The known code starts (lines 0–2) by allocating a handle that has authority over a 'buffer': a part of segment memory. It stores (3–5) a private value, 42, in the first four bytes. The intent is to call an untrusted function, $adv, with access to the rest of the buffer, but not to the private value. To do so safely, the known code *slices* (6–9) the handle to get a sub-handle that has authority only over the rest of the buffer. The known code then calls $adv, sharing only the sub-handle by passing it as an argument (11–12), and finally reads back the private value (13–14).

MSWasm guarantees that the handle $h has not been freed and the private value is unchanged after the call to $adv returns. In general, MSWasm guarantees fine-grained memory safety: unless explicitly given access to a handle with authority over a part of segment memory, a module cannot read or write to that part of segment memory.

In the rest of the paper, we show how to prove that this program's return value is either the `trap` failure value (in case the allocation or adversary call traps), or 42. We use a program logic to reason about the known code, and a logical relation to reason about the unknown code.

In §3.5, we also illustrate this approach on a stack module that showcases MSWasm and demonstrates that our approach scales to complex invariants about practical data structures.

3.1.2 Attacker Model and TCB

Wasm modules are linked together via *instantiation*. Instantiation does not take place within a Wasm program, but in a *host* — in the browser, this is typically JavaScript code. Instantiation enforces that all the modules are well-typed and have consistent exports and imports. The attacker model that we consider is one where one or more 'friendly' modules with known code are instantiated with one or more unknown, potentially adversarial Wasm modules. We assume that the host does not affect memories, locals, control flow, etc.; in our formalisation, we do this by restricting the host language. This attacker model fits the context of cloud computing (microservices, edge computing, etc.), where one client's module should be isolated from the third-party libraries it imports.

Our results concern the language specification, not a particular implementation in term of a Wasm runtime, which we still have to trust. We prove integrity, but not confidentiality — this could be tackled using a binary logical relation expanding our unary logical relation [32, §4.5], but it is outside of our scope to define an operational semantics that faithfully captures confidentiality in the setting of WebAssembly. We also have to trust the host language to match the assumptions stated above. On the mechanisation side, we have to trust the soundness of the 'kernel' proof checker of the Coq proof assistant. Crucially, we do not need to trust the Iris separation logic framework, nor the separation logic rules we define, as they are linked to the operational semantics of MSWasm by our adequacy theorem (§3.3.4).

3.2 The MSWasmCert Semantics

Michael et al. [69] present MSWasm as an extension of WebAssembly. While their pen-and-paper specification of MSWasm builds on a mostly faithful representation of WebAssembly, it remains an idealised version of the language. This results in a language specification that does not exactly line up with the official language specification of WebAssembly. Meanwhile, unlike most other industrial languages, one of the advantages of WebAssembly is that it has a detailed and comprehensive semantics [41], with a well-defined standard [95]. One of our goals is thus to formalise the MSWasm proposal as an extension of the *official* and *complete* WebAssembly semantics. This is achieved by building our formalisation on top of the WasmCert mechanisation, which covers the full language as per the 1.0 specification.

(numeric type) nt ::= **i32** | **i64** | **f32** | **f64**
(value type) t ::= nt | **handle**
(value) v ::= nt.**const** c | **handle.const** h
(byte tag) tag ::= **Numeric** | **Handle**
(in the original presentation, these are called ◯ and □)

(function type) ft ::= $ts \to ts$
(immediate)
$i, min, max, addr, off, id$ ::= **N**
(tagged byte) $tbyte$ ::= $byte \times tag$

(handles) h ::= {base : $addr$, offset : off, bound : off, valid : **bool**, id : id }

(basic instructions) b ::= nt.**const** c | t.`add` | *other stackops* | `local.{get/set}` i |
`global.{get/set}` i | t.`load` $(tp\_sx)^?$ a o |
t.`store` $tp^?$ a o | `memory.size` | `memory.grow` |
`block` ft bs | `loop` ft bs | `if` ft bs bs | `br` i | `br_if` i |
`br_table` is | `call` i | `call_indirect` i | `return` |
t.**segload** | t.**segstore** | **segalloc** | **segfree** | **handle.add** |
slice
(the flags of the `load` and `store` instructions represent a packed type, an alignment value and an offset. The new `segload` and `segstore` instructions do not have similar flags)

(administrative instructions) e ::= b | **handle.const** h | `trap` | `invoke` i |
$\texttt{label}_i\{es\}$ es `end` | $\texttt{local}_i\{F\}$ es `end` |
`call_host` tf $hidx$ vs

(functions) $func$ ::= `func` i ts bs
(memories) mem ::= `mem` min max
(elem segments) $elem$ ::= `elem` i bs_{off} is

(tables) tab ::= `tab` min max
(globals) $glob$ ::= `glob` mutable t b_{init}
(data segments) $data$ ::= `data` i bs_{off} $bytes$

Figure 3.2: WebAssembly Abstract Syntax in black, with the MSWasm additions in magenta (Part 1)

3.2.1 Plain Wasm Semantics

In this section, we briefly recall WebAssembly, highlighting the features omitted by Michael et al. [69]; a reader familiar with the language can safely skip to §3.2.2. Figures 3.2 and 3.3 show the syntax of WebAssembly, with the additions brought by MSWasm highlighted in magenta.

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

$$
\begin{array}{rl}
\text{(function instances) } \mathit{finst} & ::= \{(\mathit{inst}; ts); es\}^{\mathtt{NativeCl}}_{tf} \mid \{\mathit{hidx}\}^{\mathtt{HostCl}}_{tf} \\
\text{(table instances) } \mathit{tinst} & ::= \{\mathsf{elem} : \mathit{is},\ \ \mathsf{max} : \mathit{max}^?\} \\
\text{(memory instance) } \mathit{minst} & ::= \{\mathsf{data} : \mathit{bytes},\ \mathsf{max} : \mathit{max}^?\} \\
\text{(global instance) } \mathit{ginst} & ::= \{\mathsf{mut} : \mathit{mutable}^?,\ \mathsf{value} : v\} \\
\text{(segment instance) } \mathit{sinst} & ::= \{\mathsf{segdata} : \mathit{tbytes},\ \mathsf{max} : \mathit{max}^?\} \\
\text{(allocator instance) } \mathit{ainst} & ::= \mathit{id} \to (\mathit{addr} \times \mathit{off})^? \\
\text{(store) } S & ::= \left\{\begin{array}{l}\mathsf{funcs} : \mathit{finsts},\ \mathsf{globs} : \mathit{ginsts},\ \mathsf{mems} : \mathit{minsts},\\ \mathsf{tabs} : \mathit{tinsts},\ \mathsf{seg} : \mathit{sinst},\ \mathsf{allocator} : \mathit{ainst}\end{array}\right\} \\
\text{(frame) } F & ::= \{\mathsf{locs} : \mathit{vs},\ \ \mathsf{inst} : \mathit{inst}\} \\
\text{(module instance) } \mathit{inst} & ::= \left\{\begin{array}{l}\mathsf{types} : \mathit{fts},\ \mathsf{funcs} : \mathit{is},\ \mathsf{globs} : \mathit{is},\\ \mathsf{mems} : \mathit{is},\ \mathsf{tabs} : \mathit{is}\end{array}\right\} \\
\text{(modules) } m & ::= \left\{\begin{array}{l}\mathsf{types} : \mathit{fts},\ \mathsf{funcs} : \mathit{funcs},\ \mathsf{globs} : \mathit{globs},\ \mathsf{mems} : \mathit{mems},\\ \mathsf{tabs} : \mathit{tabs},\ \mathsf{data} : \mathit{datas},\ \mathsf{elem} : \mathit{elems},\\ \mathsf{imports} : \mathit{imports},\ \mathsf{exports} : \mathit{exports},\ \mathsf{start} : \mathit{start}\end{array}\right\}
\end{array}
$$

Figure 3.3: WebAssembly Abstract Syntax in black, with the MSWasm additions in magenta (Part 2)

A Stack Language. WebAssembly code is given as a list of instructions, and its operational semantics works as a stack machine that reduces the head instruction. For example, the operational semantics rule for addition is defined as

$$(S, F, [\textbf{i32.const}\ c_1;\ \textbf{i32.const}\ c_2;\ \textbf{i32.add}]) \hookrightarrow (S, F, [\textbf{i32.const}\ (c_1 + c_2)])$$

(we explain S and F below). In order to apply this rule in the context of a larger program, WebAssembly provides structural rules that allow to reduce under a context. For example, if $(S, F, es) \hookrightarrow (S', F', es')$, then $(S, F, vs +\!\!+\ es +\!\!+\ es_2) \hookrightarrow (S', F', vs +\!\!+\ es' +\!\!+\ es_2)$, where we write vs for a list of *values*, and es for a list of *expressions*.

The Store and the Frame. WebAssembly operates over a *store* and a *frame*. The *store* S is a record that bookkeeps all globally available functions, memories, global variables, etc. The *frame* F contains the current function's local variables, as well as its *instance*. The instance symbolises the function's environment, describing which parts of the global store the function has access to. It is defined as a record which contains indices that refer to objects in the store. This means that functions access the store via a level of indirection through the frame.

The key role of the instance in the frame is visible for example in the `global.get` instruction:

$$\frac{F.\mathsf{inst.globs}[i] = k \qquad S.\mathsf{globs}[k].\mathsf{value} = v}{(S, F, [\texttt{global.get}\ i]) \hookrightarrow (S, F, [v])}$$

All WebAssembly variables, as well as functions and all other objects are referred to with indices instead of names. For local variables, this index refers to the place in the list of local variables present in $F.\mathsf{locs}$. However, for all other objects, because they may outlive the current function (and even the current module if they are exported), the value is kept in the store together with that of objects defined in other modules. The index into the store has to be looked up in the instance, $F.\mathsf{inst}$. In the case of a global variable shown above, the instance's globs field is a list of indices into the store, the $i-$th of which corresponds to the location in the store of this module's $i-$th global variable. It is then from that location that we fetch the value of the variable from the store.

This indirection into the store via the frame is the crux of the coarse-grained encapsulation guarantees of WebAssembly. As we discuss in §3.2.2, handles achieve encapsulation very differently: they access the store directly, but are guarded by dynamic checks. The original presentation of MSWasm omits the instance from their description of the frame, and thus only accounts for handles. Meanwhile, our mechanisation captures both the coarse-grained encapsulation guarantees of WebAssembly, and the new fine-grained dynamic guarantees of handles.

Modules and Host Language. Frames and instances are constructed at runtime. Statically, WebAssembly code is shipped in *modules*, each module defining functions, a linear memory, global variables, etc. A module can *import* any of those objects, either from another WebAssembly module that explicitly *export*ed it, or from the *host language* that runs the WebAssembly modules.

Static modules are turned into dynamic module instances via *instantiation*, in which the module's code is typechecked, its imports are satisfied, and its exports are prepared for subsequent imports. This process is not part of WebAssembly itself, and hence WebAssembly code always runs embedded in a *host language*, typically Javascript, that performs module instantiation and can also perform an array of other interactions with WebAssembly code, such as calling WebAssembly functions, accessing or modifying WebAssembly state, etc. The host language can also provide functions or other objects that WebAssembly modules can import. As with frame instances, the original MSWasm presentation omits any description of modules and module instantiation.

Linear Memory. One of the objects that a module can encapsulate is the *linear memory*. In WebAssembly, the *linear memory* of a module (which Michael et al. [69] call *heap memory*) is a growable array of bytes. Linear memory is accessed via **load** and **store** instructions, which take an **i32** argument from the stack and treat it as an address. These instructions take a type as an immediate argument to know how

many bytes to access and which encoding/decoding to use. WebAssembly defines two functions, serialise and deserialise, to encode and decode all four numerical types. The **load** and **store** instructions can also take additional information (such as an offset) as immediate arguments to allow for simple pointer arithmetic. We show here a simple use of the **load** instruction, where the only immediate argument is the type to read:

$$\frac{F.\mathsf{inst.mems}[0] = k \qquad S.\mathsf{mems}[k][c..c+\mathrm{sizeof}(t)] = bs \qquad \mathrm{deserialise}(t, bs) = c'}{(S, F, [\mathbf{i32.const}\ c;\ t.\mathbf{load}]) \hookrightarrow (S, F, [t.\mathbf{const}\ c'])}$$

Just like for the global variables, the index in the store of the current module's linear memory is looked up in the instance $F.\mathsf{inst}$.[1]

Typing. WebAssembly 1.0 defines a simple type system with only four types: **i32**, **i64**, **f32** and **f64** (as we will see in §3.2.2, MSWasm introduces a new **handle** type). Instructions have type $t1s \to t2s$, where $t1s$ is the types of the values expected on the stack by the instruction, and $t2s$ is the types of the values that will be pushed on the stack. For example, $t.\mathsf{add}$ has type $[t,t] \to [t]$ and $t.\mathbf{load}$ has type $[\mathbf{i32}] \to [t]$, since addresses into memory are simple **i32** integers in WebAssembly. The WebAssembly type system guarantees that well-typed programs satisfy progress and preservation.

3.2.2 Segment Memory

In this section, we describe how MSWasm extends WebAssembly with a new kind of memory, *segment memory*, that is accessed not via **i32** integers interpreted as addresses, but via *handles*. More precisely, we present MSWasmCert — our formalisation of MSWasm in Coq — which adapts the prose description of MSWasm to a mechanisation of the full official 1.0 specification, and fixes some minor mistakes and limitations of the original prose definition.

Handles. MSWasm introduces new runtime values, handles, and a corresponding type, **handle**, which is distinct from the numeric types of WebAssembly. A handle is a form of fat pointer, represented as a record with the following fields : a base, an offset, a bound, a valid bit, and an id. The handle points to the bytes beginning at address $(\mathsf{base} + \mathsf{offset})$, its bounds of authority is described by the interval $[\mathsf{base}..\mathsf{base} + \mathsf{bound})$, and its id is used to identify a handle based on its original allocation. Handles are unforgeable, and can only either be derived from other handles, or created when a segment is allocated by the **segalloc** instruction. In particular, this means that **handle**.**const** h cannot occur in the source program, it only appears at runtime. In MSWasmCert, we enforce this syntactically: as shown in Figures 3.2 and 3.3, handle constants are not *basic instructions*, i.e. instructions available to the programmer, but rather *administrative instructions*, i.e. instructions that only appear at runtime.

[1] In WebAssembly 1.0, modules have at most one memory so the list $F.\mathsf{inst.mems}$ is of length at most one, hence the index at which we inspect it is always 0.

Dynamic Checks. A handle does not invariantly require its address $\mathsf{base} + \mathsf{offset}$ to be within its bounds of authority $[\mathsf{base}..\mathsf{base} + \mathsf{bound})$, thus allowing for common code patterns where a forbidden pointer might be created but never used (e.g. right before the end of a loop). Instead, instructions that seek to access the segment memory trigger *dynamic checks*, which guarantee that the accessed addresses are within the bounds of authority of the handle, that the validity bit is **true**, and that the handle's id is still live in the *allocator* (see below). If the conditions are met, the **segload** and **segstore** instructions are permitted to read and write from the *segment memory*. If the conditions are not met, the instructions reduce to `trap`. Just like the **load** and **store** instructions in linear memory, the **segload** and **segstore** instructions take a value type as an immediate argument to know how many bytes to read in the segment memory, and how to interpret these bytes.

Storing Handles in Memory One subtlety arises from reading handles. If no precautions are taken, a user could write a series of integer values into memory and then read them using **handle**.**segload**, effectively forging a handle. To prevent this, the bytes in segment memory are tagged as either **Handle** or **Numeric**. When reading a handle, if any of the involved bytes is tagged as **Numeric**, the read yields a handle with the validity bit set to **false**. MSWasm also mandates that reading and writing handles can only be done at addresses that are aligned with the length of a handle. This prevents forging a handle, which could otherwise be done by writing two handles consecutively in segment memory and reading from a tagged but unaligned address midway through the first. Since the bytes in linear memory are untagged, reading handles from it would be unsafe as this may forge a handle. Hence using the **load** instuction to read a handle from linear memory automatically traps. If handles must be stored and loaded, this can be done safely in segment memory by using the **segstore** and **segload** instructions.

In MSWasmCert, we abstract over what mechanism is used to serialise a handle into a byte representation, and simply assume we are provided two functions serialise_handle and deserialise_handle.

Operations on Handles Two new instructions allow for manipulating handles: **handle**.**add** adds to the offset of a handle, changing its address, while **slice** restricts its bounds of authority. Neither operation increases the *authority* of a handle, and thus both are safe. In both cases, the id stays the same, thus uniquely identifying the handle across changes: all handles that share the same id are all derived from one original handle. Accordingly, freeing one handle (see below) will simultaneously free all handles with the same id.

Modules The original pen-and-paper description of MSWasm [69] implicitly assumes that all programs run in the same module, and thus altogether omits any mention of WebAssembly modules (although their implementation reuses rWasm's support for modules). In MSWasmCert, we formally account for the full module system of

WebAssembly. To do this, we need to decide how the coarse-grained encapsulation properties of the WebAssembly module system ought to interact with the fine-grained encapsulation properties of MSWasm handles. Rather than operating over several coarsely encapsulated segment memories, we choose to limit the store to a single segment memory shared between all modules. This simplifies the design, and makes it seamless to share a handle from one module to another.[2] This also underlines that the encapsulation properties no longer stem from WebAssembly's module system, but from the handles themselves providing fine-grained memory safety.

Allocator Handles can be dynamically allocated and freed. While a handle grants spacial authority over the fragment of segment memory described in its metadata, the handle itself does not express whether that region is still temporally valid, or has already been freed. Instead, MSWasm keeps track of live handles using an *allocator*. Michael et al. [69] state that the allocator should have an 'allocation' and a 'free' function, and describe some of their expected properties. In MSWasmCert, we define the allocator as a map from handle ids to either None, meaning a handle that has been freed, or Some pair of integers representing the handle's original base address and bound. Allocating a handle is modelled by extending the map, and freeing a handle is modelled by updating its mapping to None. The programmer can perform these operations by using the **segalloc** and **segfree** instructions. Allocation is non-deterministic: the handle returned by the **segalloc** instruction could point to any non-live part of segment memory. We impose several (slightly different from Michael et al. [69]) conditions on the handle to be freed: its base and bound fields must be the original address and bound the handle got allocated as (i.e. the handle cannot have been sliced), its offset must be zero, and its validity bit must be **true**.

Operational Semantics The operational semantics rules for MSWasmCert are presented in Figures 3.5 and 3.6. These rules are almost identical to those of Michael et al. [69], with the changes brought by MSWasmCert highlighted in indigo. We describe these changes below. For brevity, we do not include failure rules, which dictate that **segload**, **segstore**, **segfree**, **handle**.**add** and **slice** all reduce to the failure value `trap` if the premises to apply the successful rule are not met. We also provide in Fig. 3.4 our own definitions for the $\langle sinst, ainst\rangle \xrightarrow{\text{salloc}(addr,off,id)} \langle sinst', ainst'\rangle$ and $\langle sinst, ainst\rangle \xrightarrow{\text{sfree}(addr,bound,id)} \langle sinst', ainst'\rangle$ predicates used in the allocation and freeing rules.

The changes in the reduction rules from MSWasm to MSWasmCert are:

- MSWasmCert introduces a second operational semantics rule for **segalloc**, allowing the allocation to non-deterministically fail, to account for realistic machine behaviour.

[2] Related questions arise in the context of capability machines featuring virtual memory [130, §3.11.3].

$$\text{aligned}(a,b) \triangleq a \text{ modulo } b = 0$$
$$\text{compatible}(addr, off, ainst) \triangleq \forall id\ addr'\ off'.\ ainst(id) = \text{Some}(addr', off') \implies (addr + off \le addr' \vee addr \ge addr' + off')$$

$$\frac{addr \le \text{length}(sinst.\mathsf{segdata}) \qquad ainst(id) = \text{None} \qquad sinst' = \{\mathsf{segdata} = sinst.\mathsf{segdata}[addr..addr+off := 0],\ \mathsf{max} = sinst.\mathsf{max}\} \qquad ainst' = ainst[id \mapsto \text{Some}(addr, off)] \qquad \text{compatible}(addr, off, ainst)}{\langle sinst, ainst\rangle \xrightarrow{\text{salloc}(addr,off,id)} \langle sinst', ainst'\rangle}$$

We do not require that $addr + off \le \text{length}(sinst.\mathsf{segdata})$, so in cases like $addr = \text{length}(sinst.\mathsf{segdata})$, we are actually growing the segment memory by appending zeros at the end.

$$\frac{ainst(id) = \text{Some}(addr, bound) \qquad ainst' = ainst[id \mapsto \text{None}]}{\langle sinst, ainst\rangle \xrightarrow{\text{sfree}(addr,bound,id)} \langle sinst, ainst'\rangle}$$

Figure 3.4: Helper functions and helper rules for the allocator

- MSWasmCert adds an extra check on the **handle**.**add** operation to ensure that the new offset is non-negative. This is a design choice that allows us to use unsigned integers to represent offsets, and means that the check for non-negativity of offset in the rules for **segload** and **segstore** are now vacuous in MSWasmCert and can be removed.

- MSWasmCert enforces that freeing a handle must be done with the original handle, not a sliced version of it (the prose definition mandated that the base address must be the original address, but did not enforce that the bound must be the original bound). This is a design choice that we have found convenient when defining our program logic, and it allows for a programmer to easily create a non-freeable handle.

- MSWasmCert fixes two minor typos from the original work [69]: the higher bound check in the **segload** and **segstore** rules should be a $\le$ instead of a $<$ (otherwise, when allocating n spaces of memory, one cannot read a value that has size n), and the bound check for the second component of **slice** should be stricter since the bound is an offset from the base rather than an address: changing the base always needs to be compensated by lowering the bound.

Buffer Example Let us come back to the buffer example from §3.1.1. We assume function $adv has type [**handle**] → [], but nothing more: it could be imported from

$$\frac{t \neq \textbf{handle} \qquad \cancel{0 \leq h.\mathsf{offset}} \qquad h.\mathsf{offset} + \mathrm{sizeof}(t) \leq h.\mathsf{bound} \qquad h.\mathsf{valid} = \textbf{true} \qquad \mathrm{isSome}(S.\mathsf{allocator}(h.\mathsf{id})) \qquad addr = h.\mathsf{base} + h.\mathsf{offset} \qquad S.\mathsf{seg}[addr..addr + \mathrm{sizeof}(t)] = tbs \qquad \mathrm{deserialise}(t, \mathrm{untag}(tbs)) = c}{(S, F, [\textbf{handle.const}\ h;\ t.\textbf{segload}]) \hookrightarrow (S, F, [t.\textbf{const}\ c])}$$

$$\frac{t = \textbf{handle} \qquad \cancel{0 \leq h.\mathsf{offset}} \qquad h.\mathsf{offset} + \mathrm{sizeof}(t) \leq h.\mathsf{bound} \qquad h.\mathsf{valid} = \textbf{true} \qquad \mathrm{isSome}(S.\mathsf{allocator}(h.\mathsf{id})) \qquad addr = h.\mathsf{base} + h.\mathsf{offset} \qquad S.\mathsf{seg}[addr..addr + \mathrm{sizeof}(t)] = tbs \qquad \mathrm{deserialise}(t, \mathrm{untag}(tbs)) = h' \qquad \mathrm{aligned}(addr, \mathrm{handle\_size}) \qquad b = \mathrm{allHandle}(\mathrm{tags}(tbs)) \qquad h_f = \mathrm{updateValid}(h', b \wedge h'.\mathsf{valid})}{(S, F, [\textbf{handle.const}\ h;\ t.\textbf{segload}]) \hookrightarrow (S, F, [t.\textbf{const}\ h_f])}$$

$$\frac{t \neq \textbf{handle} \qquad \cancel{0 \leq h.\mathsf{offset}} \qquad h.\mathsf{offset} + \mathrm{sizeof}(t) \leq h.\mathsf{bound} \qquad h.\mathsf{valid} = \textbf{true} \qquad \mathrm{isSome}(S.\mathsf{allocator}(h.\mathsf{id})) \qquad addr = h.\mathsf{base} + h.\mathsf{offset} \qquad \mathrm{serialise}(t, c) = bs \qquad seg' = S.\mathsf{seg}[addr..addr + \mathrm{sizeof}(t) := \mathrm{addTag}(bs, \textbf{Numeric})] \qquad S' = \{S \text{ with } \mathsf{seg} = sinst'\}}{(S, F, [\textbf{handle.const}\ h; t.\textbf{const}\ c;\ t.\textbf{segstore}]) \hookrightarrow (S', F, [])}$$

$$\frac{t = \textbf{handle} \qquad \cancel{0 \leq h.\mathsf{offset}} \qquad h.\mathsf{offset} + \mathrm{sizeof}(t) \leq h.\mathsf{bound} \qquad h.\mathsf{valid} = \textbf{true} \qquad \mathrm{isSome}(S.\mathsf{allocator}(h.\mathsf{id})) \qquad addr = h.\mathsf{base} + h.\mathsf{offset} \qquad \mathrm{serialise}(t, h') = bs \qquad seg' = S.\mathsf{seg}[addr..addr + \mathrm{sizeof}(t) := \mathrm{addTag}(bs, \textbf{Handle})] \qquad S' = \{S \text{ with } \mathsf{seg} = sinst'\} \qquad \mathrm{aligned}(addr, \mathrm{handle\_size})}{(S, F, [\textbf{handle.const}\ h; \textbf{handle.const}\ h';\ t.\textbf{segstore}]) \hookrightarrow (S', F, [])}$$

Figure 3.5: Operational semantics for the new instructions in MSWasm, phrased in the syntax of MSWasmCert (Part 1).
The non-cosmetic changes brought by MSWasmCert to MSWasm are highlighted in indigo. Clauses made redundant by our mechanisation are crossed out.

$$\frac{\langle S.\mathsf{seg}, S.\mathsf{allocator}\rangle \xrightarrow{\mathrm{salloc}(addr,off,id)} \langle sinst', ainst'\rangle \quad S' = \{S \text{ with } \mathsf{seg} = sinst',\ \mathsf{allocator} = ainst'\} \quad h = \{\mathsf{base} = addr,\ \mathsf{offset} = 0,\ \mathsf{bound} = off,\ \mathsf{valid} = \mathbf{true},\ \mathsf{id} = id\}}{(S, F, [\mathbf{i32.const}\ c;\ \mathbf{segalloc}]) \hookrightarrow (S', F, [\mathbf{handle.const}\ h])}$$

$$\frac{h = \{\mathsf{base} = 0,\ \mathsf{offset} = 0,\ \mathsf{bound} = 0,\ \mathsf{valid} = \mathbf{false},\ \mathsf{id} = 0\}}{(S, F, [\mathbf{i32.const}\ c;\ \mathbf{segalloc}]) \hookrightarrow (S, F, [\mathbf{handle.const}\ h])}$$

$$\frac{\langle S.\mathsf{seg}, S.\mathsf{allocator}\rangle \xrightarrow{\mathrm{sfree}(h.\mathsf{base},h.\mathsf{bound},h.\mathsf{id})} \langle sinst', ainst'\rangle \quad S' = \{S \text{ with } \mathsf{seg} = sinst',\ \mathsf{allocator} = ainst'\} \quad h.\mathsf{offset} = 0 \quad h.\mathsf{valid} = \mathbf{true}}{(S, F, [\mathbf{handle.const}\ h;\ \mathbf{segfree}]) \hookrightarrow (S', F, [])}$$

$$\frac{h.\mathsf{offset} + c \geq 0 \quad h' = \mathrm{updateOffset}(h, h.\mathsf{offset} + c)}{(S, F, [\mathbf{i32.const}\ c;\ \mathbf{handle.const}\ h;\ \mathbf{handle.add}]) \hookrightarrow (S, F, [\mathbf{handle.const}\ h'])}$$

$$\frac{0 \leq c_1 < h.\mathsf{bound} \quad c_1 \leq c_2 \quad h' = \{h \text{ with } \mathsf{base} = h.\mathsf{base} + c_1 \text{ and } \mathsf{bound} = h.\mathsf{bound} - c_2\}}{(S, F, [\mathbf{handle.const}\ h;\ \mathbf{i32.const}\ c_1;\ \mathbf{i32.const}\ c_2;\ \mathbf{slice}]) \hookrightarrow (S, F, [\mathbf{handle.const}\ h'])}$$

Figure 3.6: Operational semantics for the new instructions in MSWasm, phrased in the syntax of MSWasmCert (Part 2).
The non-cosmetic changes brought by MSWasmCert to MSWasm are highlighted in indigo.

another module and we might not know or trust its code. Since we do not share $h with this function, we expect the return value of this program to be 42. In the next section, we present a program logic that lets us prove this.

3.3 Program Logic

In order to reason about programs written in MSWasm, we define a program logic, Iris-MSWasm. Our program logic allows us to specify and verify known programs, and lays the foundations for defining the logical relation in §3.4, which allows to reason about interactions with unknown code.

Iris-MSWasm builds on top of Iris-Wasm [89], a program logic for WebAssembly, and on the Cerise family [32–36, 114, 116–118] of program logics for an idealised capability machine inspired by CHERI. Iris-Wasm captures the coarse-grained encapsulation guarantees of plain WebAssembly, so building on it helps to highlight the differences to the fine-grained encapsulation guarantees we focus on. Building on top

of Iris-Wasm also means that we inherit many properties like higher-orderness and the ability to reason about reentrant host calls. While mostly orthogonal to fine-grained memory safety, they can be desirable in many cases.

In this section, we recall Iris-Wasm, and then explain how we adapted it to MSWasm.

3.3.1 Iris-Wasm

Iris-Wasm [89] is a program logic for WebAssembly, defined in the Iris logical framework [52]. Instantiated with a language's operational semantics, Iris provides a program logic that allows to prove properties of programs, phrased in a higher-order separation logic. Atop the *structural rules* from Iris, we can derive *instruction-specific proof rules* for each instruction of the language. We can then use them to reason about WebAssembly code in a syntax-directed way.

Logical Values We define *logical values*, noted w, to describe expressions that cannot reduce. These can be of several kinds. *Immediate values* `immV` vs represent a list of WebAssembly values. The *trap value* `trapV` represents a program that has safely halted execution. Iris-Wasm also defines other kinds of logical values because of WebAssembly's expressive control flow mechanisms. The original Iris-Wasm paper describes the treatment of these other logical values, which is unchanged in Iris-MSWasm.

Specifications We phrase our proof rules and specifications using either *Hoare triples* or *weakest precondition* statements. The Hoare triple $\{P\}\, es\, \{w, \Phi(w)\}$ means that "if the precondition P holds, the expression es executes safely while maintaining all invariants, and if it terminates on a logical value w, the predicate Φ holds of that value w". A weakest precondition $\mathsf{wp}\, es\, \{w, \Phi(w)\}$ is a separation logic proposition that means "we hold precisely the resources necessary to run es safely and without breaking invariants, and if that run terminates on a logical value w, the predicate Φ holds of that value w".

Resources The Iris-Wasm program logic defines *resources* that describe ownership of the frame or ownership of fragments of the store; and weakest precondition rules corresponding to each instruction of WebAssembly, dictating what resources are needed to run each instruction. For example, the proof rule for t.**load** is given by (the coloured boxed are used to contrast with our `wp_segload` rule we introduce in §3.3.2):

`wp_load`

$$\frac{F.\mathsf{inst.mems}[0] = n \;*\; n \overset{\texttt{wms}}{\longmapsto}_i bs \;*\; \mathsf{deserialise}(t, bs) = v \;*\; \triangleright \Phi(\texttt{immV}\ [v]) \;*\; \overset{\text{FR}}{\hookrightarrow} F}{\mathsf{wp}\ [\textbf{i32.const}\ i; t.\textbf{load}] \left\{ w, \Phi(w) \;*\; \overset{\text{FR}}{\hookrightarrow} F \;*\; n \overset{\texttt{wms}}{\longmapsto}_i bs \right\}}$$

$i \overset{\texttt{wm}}{\mapsto}_{addr} b$	Ownership of a byte in linear memory
$i \overset{\texttt{wms}}{\mapsto}_{addr} bv$	Ownership of a list of bytes in linear memory
$\overset{\texttt{ws}}{\mapsto}_{addr} tb$	Ownership of a tagged byte in segment memory
$\overset{\texttt{wss}}{\mapsto}_{addr} tbs$	Ownership of a list of tagged bytes in segment memory
$id \overset{\texttt{allocated}}{\rightarrow}^{q} (addr, bound)^?$	(Fractional) ownership of a handle id in the allocator
$i \overset{\texttt{wg}}{\mapsto} \{mutability; v\}$	Ownership of a global variable
$\overset{\text{FR}}{\hookrightarrow} F$	Ownership of the WebAssembly frame

Figure 3.7: Points-to assertions corresponding to various components of the state

Taking $\Phi(w) \triangleq w = \texttt{immV}\ [v]$, this means that if we own the frame resource $\overset{\text{FR}}{\hookrightarrow} F$ and the linear memory resource[3] $n \overset{\texttt{wms}}{\mapsto}_i bs$, the load instruction executes safely. The n on the left-hand-side of the linear memory resource corresponds to the index of this module's memory in the store, looked up in the frame. If the instruction returns (which it does in this case), the return value is v, and we are handed back the frame resource $\overset{\text{FR}}{\hookrightarrow} F$ and the memory resources $n \overset{\texttt{wms}}{\mapsto}_i bs$. Fig. 3.7 displays some of the resources of the Iris-Wasm program logic, with the new resources introduced by Iris-MSWasm highlighted in magenta.

In addition to reasoning about individual WebAssembly modules, Iris-Wasm also introduces a simple host language together with a program logic for it, making it possible to reason about multiple WebAssembly modules being sequentially instantiated by the host environment. The most important piece of this host language program logic is the *instantiation lemma*, that roughly states that if a module typechecks and we own resources corresponding all its imports, the module can be instantiated and we get resources corresponding to all objects (e.g. function closures, linear memories, global variables, etc.) created by the module.

Finally, Iris-Wasm is accompanied by a logical relation that allows to reason about unknown code. We describe our extension of this logical relation in detail in §3.4.

3.3.2 Iris-MSWasm

Our program logic, Iris-MSWasm, is defined by adapting Iris-Wasm to the features introduced by MSWasm. This entailed defining the logical ghost state for allocators and segment memories, defining new resources, and proving proof rules for all new instructions.

This constituted a non-trivial programming effort, as many of the new features behave very differently from the existing ones that have been implemented in Iris-

[3] We use superscripts on the arrows (e.g. wms for the linear memory resource) to differentiate the various resources present in the program logic. Some resources like the frame resource additionally use a different arrow shape.

Wasm. For example, in plain WebAssembly, all components of the store, including linear memories, grow monotonically during execution, so a simple heap can be used to represent them. But the segment memory can have parts of it freed, so a ghost map had to be used instead of a heap. Additionally, converting a linear memory to an index-map is as simple as mapping all indices from 0 to the size of the memory to their corresponding byte. For the segment memory, only live addresses should point to a value, increasing the complexity of the definitions.

To accommodate for the new type of memory, we introduce new points-to resources, as described in Fig. 3.7. The segment memory resource $\overset{\texttt{ws}}{\longmapsto}_{addr}\ tb$ represents ownership of a single tagged byte in segment memory. Allocator resources $id \xrightarrow{\texttt{allocated}}{}^{q} \mathsf{None}$ or $id \xrightarrow{\texttt{allocated}}{}^{q} \mathsf{Some}(addr, bound)$ represent fractional ownership of a handle id in the allocator. q is rational in $(0,1]$. The case where $q = 1$ represents full ownership and allows to access or modify the value on the right-hand-side of the arrow; in that case we may omit writing the fraction. If $q < 1$, the resource is only partially owned: the right-hand-side value can be accessed, but not modified. Since freeing a handle corresponds to updating its value in the allocator from $\mathsf{Some}\ (base, bound)$ to None, freeing requires full ownership, and we can use partial resources to symbolise handles that are unfreeable because they have been sliced. We also define a syntactic sugar for ownership of a list of tagged bytes tbs in segment memory: $\overset{\texttt{wss}}{\longmapsto}_{addr} tbs$. Contrary to the resources for linear memory, there is no store index on the left-hand side of the arrow in the segment memory resources. This reflects the fact that all modules share one common segment memory.

Using theses new resources, we define and prove new weakest-precondition rules for all of MSWasm's new instructions. We present these new rules in Figures 3.8 and 3.9. These rules mirror the operational semantics introduced in §3.2.2. For example, the rule for t.**segload** is:

$$\texttt{wp\_segload}$$

$$\frac{t \neq \textbf{handle} * \overset{\texttt{wss}}{\longmapsto}_{addr}\ tbs * h.\mathsf{id} \xrightarrow{\texttt{allocated}} \mathsf{Some}(x) * h.\mathsf{offset} + \mathsf{sizeof}(t) \leq h.\mathsf{bound} * addr = h.\mathsf{base} + h.\mathsf{offset} * h.\mathsf{valid} = \textbf{true} * \mathsf{deserialise}(t, \mathsf{untag}(tbs)) = v * \triangleright \Phi(\texttt{immV}\ [v]) * \xrightarrow{\text{FR}} F}{\mathsf{wp}\ [\textbf{handle.const}\ h; t.\textbf{segload}] \left\{ w, \begin{array}{l} \Phi(w) * \xrightarrow{\text{FR}} F * \overset{\texttt{wss}}{\longmapsto}_{addr}\ tbs * \\ h.\mathsf{id} \xrightarrow{\texttt{allocated}} \mathsf{Some}\ (x) \end{array} \right\}}$$

This rule is quite close to the wp_load rule from §3.3.1. The differences are 1. the segment rule does dynamic checks to ensure the read is admissible, 2. the memory resource is a linear memory resource in the wp_load rule but a segment memory resource in the wp_segload rule (which also means that the premise looking up an index in the frame instance F.inst is unnecessary in the segment rule), and 3. the allocator resource is additionally present in the segment rule.

The premise $t \neq \textbf{handle}$ in wp_segload is required, because in the case of reading a handle from memory, additional checks are necessary and hence requires a separate

wp_segload

$$\frac{t \neq \textbf{handle} \ast \overset{\texttt{wss}}{\longmapsto}_{addr} tbs \ast h.\mathsf{id} \xrightarrow{\texttt{allocated}}{}^{q} \mathsf{Some}(x) \ast h.\mathsf{offset} + \mathsf{sizeof}(t) \leq h.\mathsf{bound} \ast addr = h.\mathsf{base} + h.\mathsf{offset} \ast h.\mathsf{valid} = \textbf{true} \ast \mathsf{deserialise}(t, \mathsf{untag}(tbs)) = v \ast \triangleright \Phi(\texttt{immV}\ [v]) \ast \overset{\text{FR}}{\hookrightarrow} F}{\mathsf{wp}\ [\textbf{handle.const}\ h; t.\textbf{segload}] \left\{ w,\ \begin{array}{l} \Phi(w) \ast \overset{\texttt{wss}}{\longmapsto}_{addr} tbs \ast \\ h.\mathsf{id} \xrightarrow{\texttt{allocated}}{}^{q} \mathsf{Some}(x) \ast \overset{\text{FR}}{\hookrightarrow} F \end{array} \right\}}$$

wp_segload_handle

$$\frac{t = \textbf{handle} \ast \overset{\texttt{wss}}{\longmapsto}_{addr} tbs \ast h.\mathsf{id} \xrightarrow{\texttt{allocated}}{}^{q} \mathsf{Some}(x) \ast h.\mathsf{offset} + \mathsf{sizeof}(t) \leq h.\mathsf{bound} \ast \mathsf{aligned}(addr, \mathsf{handle\_size}) \ast b = \mathsf{allHandle}(\mathsf{tags}(tbs)) \ast h_f = \mathsf{updateValid}(h', b \wedge h'.\mathsf{valid}) \ast addr = h.\mathsf{base} + h.\mathsf{offset} \ast h.\mathsf{valid} = \textbf{true} \ast \mathsf{deserialise}(t, \mathsf{untag}(tbs)) = h' \ast \triangleright \Phi(\texttt{immV}\ [h_f]) \ast \overset{\text{FR}}{\hookrightarrow} F}{\mathsf{wp}\ [\textbf{handle.const}\ h; t.\textbf{segload}] \left\{ w,\ \begin{array}{l} \Phi(w) \ast \overset{\texttt{wss}}{\longmapsto}_{addr} tbs \ast \\ h.\mathsf{id} \xrightarrow{\texttt{allocated}}{}^{q} \mathsf{Some}(x) \ast \overset{\text{FR}}{\hookrightarrow} F \end{array} \right\}}$$

wp_segload_failure1

$$\frac{\left(h.\mathsf{offset} + \mathsf{sizeof}(t) > h.\mathsf{bound} \vee h.\mathsf{valid} = \textbf{false} \vee (t = \textbf{handle} \wedge \neg \mathsf{aligned}(h.\mathsf{base} + h.\mathsf{offset}, \mathsf{handle\_size}))\right) \ast \triangleright \Phi(\texttt{trapV}) \ast \overset{\text{FR}}{\hookrightarrow} F}{\mathsf{wp}\ [\textbf{handle.const}\ h; t.\textbf{segload}] \left\{ w, \Phi(w) \ast \overset{\text{FR}}{\hookrightarrow} F \right\}}$$

wp_segload_failure2

$$\frac{h.\mathsf{id} \xrightarrow{\texttt{allocated}}{}^{q} \mathsf{None} \ast \triangleright \Phi(\texttt{trapV}) \ast \overset{\text{FR}}{\hookrightarrow} F}{\mathsf{wp}\ [\textbf{handle.const}\ h; t.\textbf{segload}] \left\{ w, \Phi(w) \ast h.\mathsf{id} \xrightarrow{\texttt{allocated}} \mathsf{None} \ast \overset{\text{FR}}{\hookrightarrow} F \right\}}$$

(We omit the similar failure rules for **segstore**, **segfree**, **handle.add** and **slice**; these rules are shown in our supplementary material)

Figure 3.8: Iris-MSWasm rules for the **segload** instruction

wp_segload_handle rule, as displayed in Fig. 3.8. A similar $t \neq \textbf{handle}$ premise has to be added to wp_load in Iris-MSWasm, since reading a handle from linear memory is not allowed in the MSWasm semantics. This is the only modification necessary to a rule from Iris-Wasm when defining Iris-MSWasm.

wp_segstore

$$\frac{\begin{array}{c} t \neq \textbf{handle} \ * \ \overset{\texttt{wss}}{\longmapsto}_{addr} tbs \ * \ h.\mathsf{id} \xrightarrow{\texttt{allocated}}{}^{q} \mathsf{Some}(x) \ * \\ h.\mathsf{offset} + \mathsf{sizeof}(t) \leq h.\mathsf{bound} \ * \ addr = h.\mathsf{base} + \mathsf{offset} \ * \ h.\mathsf{valid} = \textbf{true} \ * \\ \textbf{typeof}(v) = t \ * \ |bs| = \mathsf{sizeof}(t) \ * \ \mathsf{serialise}(t, v) = bs \ * \\ \mathsf{addTag}(bs, \textbf{Numeric}) = tbs' \ * \ \triangleright \Phi(\texttt{immV}\ []) \ * \ \hookrightarrow^{\mathrm{FR}} F \end{array}}{\mathsf{wp}\,[\textbf{handle.const}\ h; v; t.\textbf{segstore}] \left\{ w, \begin{array}{l} \Phi(w) \ * \ \overset{\texttt{wss}}{\longmapsto}_{addr} tbs' \ * \\ h.\mathsf{id} \xrightarrow{\texttt{allocated}}{}^{q} \mathsf{Some}(x) \ * \ \hookrightarrow^{\mathrm{FR}} F \end{array} \right\}}$$

wp_segstore_handle

$$\frac{\begin{array}{c} t = \textbf{handle} \ * \ \overset{\texttt{wss}}{\longmapsto}_{addr} tbs \ * \ h.\mathsf{id} \xrightarrow{\texttt{allocated}}{}^{q} \mathsf{Some}(x) \ * \\ h.\mathsf{offset} + \mathsf{sizeof}(t) \leq h.\mathsf{bound} \ * \ addr = h.\mathsf{base} + \mathsf{offset} \ * \ h.\mathsf{valid} = \textbf{true} \ * \\ \mathsf{aligned}(addr, \mathsf{handle\_size}) \ * \ |bs| = \mathsf{sizeof}(t) \ * \\ \mathsf{serialise}(t, h') = bs \ * \ \mathsf{addTag}(bs, \textbf{Handle}) = tbs' \ * \ \triangleright \Phi(\texttt{immV}\ []) \ * \ \hookrightarrow^{\mathrm{FR}} F \end{array}}{\mathsf{wp}\,[\textbf{handle.const}\ h; \textbf{handle.const}\ h'; t.\textbf{segstore}] \left\{ \begin{array}{l} w, \ \Phi(w) \ * \ \overset{\texttt{wss}}{\longmapsto}_{addr} tbs' \ * \\ h.\mathsf{id} \xrightarrow{\texttt{allocated}}{}^{q} \mathsf{Some}(x) \\ * \ \hookrightarrow^{\mathrm{FR}} F \end{array} \right\}}$$

wp_segfree

$$\frac{\begin{array}{c} h.\mathsf{valid} = \textbf{true} \ * \ h.\mathsf{offset} = 0 \ * \ |tbs = b| \ * \ \overset{\texttt{wss}}{\longmapsto}_{h.\mathsf{base}} tbs \ * \\ h.\mathsf{id} \xrightarrow{\texttt{allocated}} \mathsf{Some}(h.\mathsf{base}, b) \ * \ \triangleright \Phi(\texttt{immV}\ []) \ * \ \hookrightarrow^{\mathrm{FR}} F \end{array}}{\mathsf{wp}\,[\textbf{handle.const}\ h; \textbf{segfree}] \left\{ w, \Phi(w) \ * \ \hookrightarrow^{\mathrm{FR}} F \right\}}$$

wp_segalloc

$$\frac{\begin{array}{c} \triangleright \Big(\forall w. \big(\exists h. w = \texttt{immV}\ [\textbf{handle.const}\ h] \ * \ (h.\mathsf{valid} = \textbf{false} \vee \\ (\mathsf{id} \xrightarrow{\texttt{allocated}} \mathsf{Some}(h.\mathsf{base}, n) \ * \ h.\mathsf{bound} = n \ * \ h.\mathsf{offset} = 0 \ * \\ h.\mathsf{valid} = \textbf{true} \ * \ \overset{\texttt{wss}}{\longmapsto}_{h.\mathsf{base}} \mathsf{repeat}(n, 0)))) \mathrel{-\!\!*} \Phi(w) \Big) \ * \ \hookrightarrow^{\mathrm{FR}} F \end{array}}{\mathsf{wp}\,[\textbf{i32.const}\ n; \textbf{segalloc}] \left\{ w, \Phi(w) \ * \ \hookrightarrow^{\mathrm{FR}} F \right\}}$$

wp_handleadd

$$\frac{\begin{array}{c} h' = \{ h \text{ with } \mathsf{offset} = h.\mathsf{offset} + c \} \ * \\ h.\mathsf{offset} + c \geq 0 \ * \ \triangleright \Phi(\texttt{immV}\ [\textbf{handle.const}\ h']) \ * \ \hookrightarrow^{\mathrm{FR}} F \end{array}}{\mathsf{wp}\,[\textbf{i32.const}\ c;\ \textbf{handle.const}\ h;\ \textbf{handle.add}] \left\{ w, \Phi(w) \ * \ \hookrightarrow^{\mathrm{FR}} F \right\}}$$

wp_slice

$$\frac{\begin{array}{c} h' = \{ h \text{ with } \mathsf{base} = h.\mathsf{base} + c_1 \text{ and } \mathsf{bound} = h.\mathsf{bound} - c_2 \} \ * \\ 0 \leq c_1 < h.\mathsf{bound} \ * \ c_1 \leq c_2 \ * \ \triangleright \Phi(\texttt{immV}\ [\textbf{handle.const}\ h']) \ * \ \hookrightarrow^{\mathrm{FR}} F \end{array}}{\mathsf{wp}\,[\textbf{handle.const}\ h;\ \textbf{i32.const}\ c_1;\ \textbf{i32.const}\ c_2;\ \textbf{slice}] \left\{ w, \Phi(w) \ * \ \hookrightarrow^{\mathrm{FR}} F \right\}}$$

Figure 3.9: Other Iris-MSWasm specific rules

3.3.3 Specifying the Known Parts of the Buffer Example

Let us come back to the buffer example from §3.1.1, whose code is in Fig. 3.1. In this section, we show how to reason about the known parts of its code, and we defer the discussion about the adversary call to §3.4.3. This explanation is quite technical, because we detail the entire proof. Its mechanisation can be found in our Coq development in file `buffer_code.v`.

Our goal will be to prove that

$$\left\{\xhookrightarrow{\text{FR}} F\right\}\texttt{buffer\_example}\left\{w, (\exists F'.\xhookrightarrow{\text{FR}} F') * \left(\begin{array}{l} w = \texttt{trapV} \vee \\ w = \texttt{immV}\ [\textbf{i32.const}\ 42] \end{array}\right)\right\}$$

where F is a frame where two local variables $h and $hpub are declared, both of type **handle**, and the instance contains a function $adv of type $[\textbf{handle}] \to []$. Our desired post-condition allows the program to trap: this could correspond either to the allocation failing, or to the function call failing. Crucially, *trapping is safe*, as it ensures that no memory violation has occurred.

Since we focus here on reasoning about known code, we assume until the end of this section that we know a specification for function $adv, say:

$$\forall h, q.\left\{\begin{array}{l} \xmapsto{\texttt{wss}}_{h.\mathsf{base}} - * \\ h.\mathsf{id} \xrightarrow{\texttt{allocated}}{}^{q}\ \mathsf{Some}- \end{array}\right\}$$
$$[\textbf{handle.const}\ h; \texttt{call}\ \$adv]$$
$$\left\{\begin{array}{l} w = \texttt{trapV} \vee \\ w,\ \left(\begin{array}{l} w = \texttt{immV}\ [] * \\ \exists tbs'.\xmapsto{\texttt{wss}}_{h.\mathsf{base}} tbs' * \\ \exists opt.\, h.\mathsf{id} \xrightarrow{\texttt{allocated}}{}^{q}\ opt \end{array}\right) \end{array}\right\}$$

In other words, the function can be called on any handle input, as long as the caller has ownership of the segment memory region pointed by that handle. The function may trap, but if it does not, it yields back ownership of the same segment memory region upon return; the tagged bytes might have changed. We will return in §3.4.3 to the more general case where the function is completely untrusted and we are not given a specification for it.

Let us proceed instruction by instruction, recalling the resources we own at each point of the program. The resources displayed in gold are the ones necessary to fulfil premises of the next rule to be applied; the ones in black are unused and simply carried forward.

$$\left\{\xhookrightarrow{\text{FR}} F\right\}$$

Lines 0–1 At the start, we only own the frame resource $\xhookrightarrow{\text{FR}} F$. We can apply[4] rule `wp_segalloc` from Fig. 3.9 with

$$\begin{array}{rl} \Phi(w) \triangleq & (\exists h.w = \texttt{immV}\ [\textbf{handle.const}\ h] * \\ & (h.\mathsf{valid} = \textbf{false} \vee (\mathsf{id} \xrightarrow{\texttt{allocated}} \mathsf{Some}(h.\mathsf{base}, n) * \\ & h.\mathsf{bound} = n * h.\mathsf{offset} = 0 * h.\mathsf{valid} = \textbf{true} * \xmapsto{\texttt{wss}}_{h.\mathsf{base}} \mathsf{repeat}(n, 0)))) \end{array}$$

[4] We omit the structural rules that allow to bind the first instruction in order to apply the proof rule.

(hence the wand implication in the first premise is a trivial $P \mathrel{-\!\!*} P$). To satisfy the second premise, we yield the resource $\hookleftarrow^{\text{FR}} F$. The post-condition gives us back the resource $\hookleftarrow^{\text{FR}} F$, and tells us that a value **handle**.**const** h has now been placed on the stack, and that either $h.\mathsf{valid} = \textbf{false}$ (representing a failed allocation), or we own the segment and allocator resources. If we define $x \triangleq (h.\mathsf{base}, h.\mathsf{bound})$, we have:

$$\left\{ \hookleftarrow^{\text{FR}} F * \left(h.\mathsf{valid} = \textbf{false} \vee \overset{\texttt{wss}}{\longmapsto}_{h.\mathsf{base}} \mathsf{repeat}(8,0) * h.\mathsf{id} \xrightarrow{\texttt{allocated}} \mathsf{Some}\ x \right) \right\}$$

Lines 2–3 The next two instructions are `local.set` and `local.get`. Both instructions have corresponding proof rules in Iris-Wasm, which we apply sequentially. In both cases, the Iris-Wasm proof rule consumes the frame resource $\hookleftarrow^{\text{FR}} F$ as a premise, and gives it back in the post-condition. `local.set` changes the frame to new one, F', where the value of local variable $h is now h:

$$\left\{ \hookleftarrow^{\text{FR}} F' * \left(h.\mathsf{valid} = \textbf{false} \vee \overset{\texttt{wss}}{\longmapsto}_{h.\mathsf{base}} \mathsf{repeat}(4+4,0) * h.\mathsf{id} \xrightarrow{\texttt{allocated}} \mathsf{Some}\ x \right) \right\}$$

Lines 4–5 The next instruction is **segstore**. At this stage, we perform a case disjunction: if $h.\mathsf{valid} = \textbf{false}$ (i.e. the allocation has failed), then the failure rule `wp_segstore_failure1` (the **segstore** equivalent of rule `wp_segload_failure1` from Fig. 3.8) applies since one of the dynamic checks fails. Hence we trap safely, and in this case we can conclude the whole proof here, as we have filled the first disjunct of the post-condition.

Let us now consider the second case: we own $\overset{\texttt{wss}}{\longmapsto}_{h.\mathsf{base}} \mathsf{repeat}(8,0)$. We can apply rule `wp_segstore` from Fig. 3.9 with $\Phi(w) \triangleq w = \texttt{immV}\ []$. To fulfil the segment resource premise of the rule, we must yield the first half of the resource we hold. Thus we separate $\overset{\texttt{wss}}{\longmapsto}_{h.\mathsf{base}} \mathsf{repeat}(8,0)$ into two resources $\overset{\texttt{wss}}{\longmapsto}_{h.\mathsf{base}} \mathsf{repeat}(4,0)$ and $\overset{\texttt{wss}}{\longmapsto}_{h.\mathsf{base}+4} \mathsf{repeat}(4,0)$. We yield the first of these (as well as the frame resource $\hookleftarrow^{\text{FR}} F'$ and our allocator resource) to satisfy the premises of `wp_segstore`, and the latter is unused for this rule. The other premises are all the necessary dynamic checks, which are satisfied here, and the rules give us our resources back, having updated the tagged bytes in segment memory to now store our private value 42.

$$\left\{ \begin{array}{l} \hookleftarrow^{\text{FR}} F' * \overset{\texttt{wss}}{\longmapsto}_{h.\mathsf{base}} \mathsf{serialise}(\textbf{i32}, 42) * \overset{\texttt{wss}}{\longmapsto}_{h.\mathsf{base}+4} \mathsf{repeat}(4,0) * \\ h.\mathsf{id} \xrightarrow{\texttt{allocated}} \mathsf{Some}\ x \end{array} \right\}$$

Lines 6–11 The next instructions are `local.get`, **slice**, `local.set` and `local.get` again. All of these instructions have associated proof rules: `wp_slice` from Fig. 3.9 for **slice**, and rules from Iris-Wasm for the local variables. The rule for `local.set` has changed the frame again to update the value of variable $hpub to h', the "second half" of h that we obtained via slicing; we call F'' this new frame.

$$\left\{ \begin{array}{l} \hookleftarrow^{\text{FR}} F'' * \overset{\texttt{wss}}{\longmapsto}_{h.\mathsf{base}} \mathsf{serialise}(\textbf{i32}, 42) * \overset{\texttt{wss}}{\longmapsto}_{h.\mathsf{base}+4} \mathsf{repeat}(4,0) * \\ h.\mathsf{id} \xrightarrow{\texttt{allocated}}{}^{1/2+1/2} \mathsf{Some}\ x \end{array} \right\}$$

Line 12 Now, we come to the call to function $adv. In our simplified setting, we have a specification, which we wish to apply. Since by definition $h'.\mathsf{base} = h.\mathsf{base}+4$ and $h'.\mathsf{id} = h.\mathsf{id}$, we have all the resources needed to fill the precondition. If we apply the specification with $q = 1$, we must lose the entire allocator resource to fulfil the precondition of the specification, and we would only get back that there exists *opt* such that $h.\mathsf{id} \xrightarrow{\texttt{allocated}} opt$. This would not allow us to later execute the **segload** instruction. Instead, we can separate our allocator resource into two partial resources $h.\mathsf{id} \xrightarrow{\texttt{allocated}}^{1/2} \mathsf{Some}\ x$. Now we can apply the specification with $q = \frac{1}{2}$ yielding only one of our partial resources, and keeping the second.

The postcondition tells us that either the call has trapped (in which case we can terminate the proof like before), or there exists some tagged bytes tbs' and an option *opt* such that we now own $\mapsto^{\texttt{wss}}_{h'.\mathsf{base}} tbs'$ and $h.\mathsf{id} \xrightarrow{\texttt{allocated}}^{1/2} opt$. Combined with the partial resource we kept, we know that $opt = \mathsf{Some}\ x$, and we can combine our two fragments to get a full allocator resource. Informally, that means that the handle is still allocated.

Importantly, the other handle is not required by the specification, and hence the segment resource $\mapsto^{\texttt{wss}}_{h.\mathsf{base}} \mathsf{serialise}(\mathbf{i32}, 42)$ is framed away.

$$\left\{ \hookleftarrow^{\text{FR}} F'' * \mapsto^{\texttt{wss}}_{h.\mathsf{base}} \mathsf{serialise}(\mathbf{i32}, 42) * \mapsto^{\texttt{wss}}_{h.\mathsf{base}+4} tbs' * h.\mathsf{id} \xrightarrow{\texttt{allocated}} \mathsf{Some}\ x \right\}$$

Lines 13–14 Lastly, we use the Iris-Wasm rule for `local.get` to get the value of variable $h, and rule `wp_segload` from Fig. 3.8 allows us to conclude that the return value is indeed 42 as expected.

In the next section, we show how we can achieve the same result when the function $adv is not specified.

3.3.4 Adequacy

The Iris logical framework provides an *adequacy theorem* [52, §6.4] that relates the weakest precondition statement to the operational semantics. This means that Iris is not in our Trusted Computing Base, as holding a weakest precondition now implies a statement phrased entirely in terms of the operational semantics of MSWasmCert.

Theorem 5 (Adequacy)**.** *If* $wp\, es\, \{w, \Phi(w)\}$ *and* $(S, F, es) \hookrightarrow^* (S', F', vs)$ *for some values vs, then* $\Phi(vs)$ *holds.*

Using the adequacy theorem, we can prove the following result for the buffer example from §3.1.1:

Theorem 6 (Buffer Example)**.** *If the code in Fig. 3.1 terminates, it terminates on either the trap value* `trapV`*, or on value 42*

Proof sketch. We begin by proving

$$\left\{ \hookleftarrow^{\text{FR}} F \right\} \texttt{buffer\_example} \left\{ w, (\exists F'. \hookleftarrow^{\text{FR}} F') * \begin{pmatrix} w = \texttt{trapV} \vee \\ w = \texttt{immV}\ [\mathbf{i32.const}\ 42] \end{pmatrix} \right\}$$

We have shown in §3.3.3 how to reason about the known parts of the code, and we will show in §3.4.3 how to reason about the unknown code; hence we have the wanted Hoare triple. This proof can also be seen in our Coq development in file `buffer_code.v`.

Then, we use the program logic for our host language to reason about the instantiation on the adversary module and the buffer module. The instantiation lemma provides the frame resource $\xhookrightarrow{\text{FR}} F$ from the precondition of the Hoare triple. This yields the weakest precondition statement

$$\texttt{wp}\,\texttt{buffer\_instantiation}\,\{w, w = \texttt{trapV} \lor w = \texttt{immV}\ [\textbf{i32.const}\ 42]\}$$

A proof of this can be seen in our Coq development in file `buffer_instantiation`.

Finally, we apply the adequacy theorem which yields the desired result. This entails carefully providing all of the resource algebras necessary to implement the logical state of Iris and use all the ghost resources that Iris-MSWasm leverages. A mechanised proof can be seen in our Coq development in file `buffer_adequacy.v`. □

3.4 Robust Capability Safety

We have described how to use Iris-MSWasm to reason about known code. What remains to verify a complete example is to explain how to reason about unknown, potentially adversarial code. More precisely, when proving the weakest precondition for the buffer example, we eventually reach the call to the unknown imported function. At that point, one of our proof obligation is to show the weakest precondition for the body of that function. Since the function is arbitrary, we cannot step through its instructions. And since the function is untrusted, we cannot simply assume that we are given a weakest precondition for it. Instead, we want to define a *universal* specification for unknown code, which gives an over-approximation of its behaviour in the form of a weakest precondition.

To that end, we define a logical relation for the MSWasm type system, and prove that it satisfies the fundamental theorem of logical relations. In essence, the logical relation defines what it means for a value to be *safe to share*, and an expression to be *safe to execute*. Our logical relation builds on the logical relation defined in Iris-Wasm [89], and follows the typical design of step-indexed logical relations [2] in Iris [57, 126], and applies the techniques used in the Cerise line of work [33, 34, 36]. We present the intuition behind our logical relation in §3.4.1, and then define it and show that it is sound in §3.4.2, and showcase how it gives us robust safety on our buffer example in §3.4.3.

3.4.1 Informal Intuition

The high-level idea behind our logical relation is to define what it means for a value to be *safe to share*, and an expression to be *safe to execute*. What this means depends on the type of the value or expression: for example, a handle is safe to share if it

grants memory access to its range of authority (i.e. grants access to the relevant points-to predicates), and if that memory recursively contains values that are safe to share. Meanwhile, an expression *es* is safe to execute when there is a weakest precondition for it $\mathtt{wp}\, es\, \{w, w \text{ is safe to share}\}$. In this simplified definition, *es* either loops, or reduces to a value that is safe to share. The formal definition has to account for programs that reduce to `trapV`, as well as programs that either `return` or `break` to the surrounding context. Crucially, as described earlier, a program that reduces to `trapV` (say, because it failed a dynamic check) is safe to execute.[5]

The definitions of *safe to share* and *safe to execute* can be viewed as a *universal contract*, in the sense that it holds for all well-typed MSWasm programs. A key theorem is to prove that this is the case. We call this result the *fundamental theorem of the logical relation*: if a program *es* is a well-typed MSWasm program, then it is *safe to execute*. We state this theorem formally in §3.4.2.

By applying the fundamental theorem, since module instantiation guarantees that its functions are well typed, we can derive weakest precondition specifications for imported functions, even when they are unknown. The caveat is that in order to get this specification, any shared handle must *also* satisfy the universal contract, i.e. satisfy the value interpretation for handles. Thus, a key feature of the logical relation is to capture the fine-grained encapsulation properties of handles, so as not to impose invariants over segment regions that are *not* shared.

3.4.2 Logical Relation

More formally, we define, for each type t, a predicate $\mathscr{V}[\![t]\!]$ describing values that are safe to share, called the *value interpretation* of type t, and a predicate $\mathscr{E}[\![t]\!]$ of expressions that are safe to execute, called the *expression interpretation* of type t.

The difficulty when defining a logical relation for a full industrial language is that one must define a logical interpretation for all objects of the language: not only values and expressions, but also frames, function closures, linear memories, instances, contexts, etc. Iris-Wasm defines a relation for each WebAssembly object. In this work, we extend it to interpret the new types introduced by MSWasm. In particular, we define new interpretations for handle values and allocators. To keep the explanations simple, we will primarily focus on these new logical relations, and refer to the Coq mechanisation for the full definition. That being said, the explanations are somewhat technical, and will assume some familiarity with various Iris concepts.

Value Interpretation The value interpretation is shown in Fig. 3.18. It states that a logical value is safe for types *ts* if it either is the trap value `trapV` (recall that we consider expressions that trap to be safe), or if it is a list of WebAssembly values which satisfy $\mathscr{V}_0$ for each value type in *ts*.

[5] As mentionned in §3.1.2, we only consider integrity properties. If we were to consider confidentiality properties, we would need to consider potential interoperability with IO

$\boxed{\mathscr{V}[\![ts]\!] : \mathit{LogVal} \to \mathit{iProp}}$

$$\text{ValidHandleAddr}(addr, base', bound') \triangleq \begin{array}{l}\text{aligned}(base' + addr, \text{handle\_size}) \wedge \\ 0 \le addr \wedge addr + \text{handle\_size} \le bound'\end{array}$$

$$\begin{array}{l}\mathscr{V}_0[\![\textbf{handle}]\!](v) \quad \triangleq \\ \quad \exists h.\, v = \textbf{handle.const}\ h * \\ \quad \left(\begin{array}{l} h.\mathsf{valid} = \textbf{false} \ \vee \\ \exists \gamma, base', bound', base'', bound'', q. \\ [h.\mathsf{base}..h.\mathsf{base} + h.\mathsf{bound}) \subseteq [base'..base' + bound') * \quad (1) \\ [base'..base' + bound') \subseteq [base''..base'' + bound'') * \quad (2) \\ q \in \left\{\frac{1}{2}, 1\right\} * \left((h.\mathsf{base} = base'' * h.\mathsf{bound} = bound'') \implies q = 1\right) * \quad (3) \\ \boxed{\circ\,(h.\mathsf{id} \hookrightarrow (\gamma, base'', bound'', q))}^{\gamma_{\text{toks}}} * \quad (4) \\ \boxed{\begin{array}{l}\exists tbs. |tbs| = bound' * \overset{\texttt{wss}}{\longmapsto}_{base'} tbs * \\ \quad \forall addr. \text{ValidHandleAddr}(addr, base', bound') \mathrel{-\!\!*} \\ \quad \left(\begin{array}{l}\left(\begin{array}{l}\exists off.\, 0 \le off < \text{handle\_size} \wedge \\ \quad tbs[addr + off] = (-, \textbf{Numeric})\end{array}\right) \vee \\ \mathscr{V}_0[\![\textbf{handle}]\!](\textbf{handle.const} \\ \quad \text{deserialise\_handle}(\\ \quad\quad \text{untag}(tbs[addr..addr + \text{handle\_size}]))\end{array}\right)\end{array}}^{\gamma} (5) \\ \mathsf{CInv}\end{array}\right)\end{array}$$

$$\begin{array}{rcl}\mathscr{V}_0[\![t]\!](v) & \triangleq & \exists c.\, v = t.\texttt{const}\ c \qquad (\text{for } t \neq \textbf{handle}) \\ \mathscr{V}[\![[t_1, \cdots, t_n]]\!](w) & \triangleq & w = \texttt{trapV} \vee \\ & & \exists v_1, \cdots, v_n.\, w = \texttt{immV}\ [v_1, \cdots, v_n] \wedge \\ & & \mathscr{V}_0[\![t_1]\!](v_1) \wedge \cdots \wedge \mathscr{V}_0[\![t_n]\!](v_n)\end{array}$$

Figure 3.10: Our logical relation for values

$\mathscr{V}_0$ defines the interpretation of value types, namely numerical types and handles. For numerical types, Iris-Wasm simply asserts that the numerical value has the appropriate format (32 bit integer for **i32**, etc.). It is interesting to note, that although **i32**s are used to access linear memory, $\mathscr{V}_0$ does not model this usage. Indeed, this is by design: although **i32**s are used as pointers, it is the instance within the frame that provides the *authority* to access linear memory. As such, it is the interpretation of linear memory that determines which points-to predicates can be used by a function. In short, the interpretation of linear memory imposes an invariant over the *entirety* of a module's linear memory, thus expressing how a module may forge pointers to access any byte within it.

Meanwhile, handles specifically do *not* grant authority over the entire segment memory. Instead, our goal is to model the exact authority granted by a handle: namely the authority to access the locations within its bounds of authority, and the authority to free a handle if its bounds match the original bounds of that handle as represented

in the allocator.

Let us take a more detailed look at our definition for the value interpretation for handles in Fig. 3.18. It states that a value is in the interpretation for handles if it is a handle h, and either this handle is invalid, or we own 1. a range $[base'..base' + bound')$, representing the bounds of the handle when it was originally *shared*[6], 2. a range $[base''..base'' + bound'')$, representing the bounds of the handle when it was originally *allocated* (we want to remember so we can determine whether the handle grants the authority to be freed[7]) 3. a fraction q that can be $\frac{1}{2}$ or 1, and has to be 1 if $h.\mathsf{base} = base''$ and $h.\mathsf{bound} = bound''$ (this fraction will be used to model the authority to free a handle) , 4. a ghost resource binding $h.\mathsf{id}$ to an invariant name γ, the range $[base''..base'' + bound'')$ and the fraction q (this resource will be used to remember the original state of a handle, at its allocation), and 5. an invariant that contains the segment memory locations associated to the range $[base'..base' + bound')$, such that all locations in memory that might store a handle (i.e. are in bounds and are aligned that are aligned with the handle size) either have at least one byte tagged as **Numeric**, or hold a value that satisfies the value interpretation for handles.

The ghost resource (4) is a fragment view of a map whose authoritative view is in the interpretation for the allocator. In other words, this ghost resource serves to share information about handle ids between the value interpretation and the allocator interpretation, as we will detail later. The ghost name γ_{toks} is a global value that is also used by the interpretation for the allocator.

Since handles can be freed, we use Iris' *cancellable invariants* [52, §7.1.3]. A cancellable invariant uses a token $[\text{CInv} : \gamma]$ to track whether an invariant is still live. This token is required to open the invariant, and can be consumed to cancel the invariant when the handle is being freed, after which the segment memory resources become unavailable. Previous mechanisations of robust capability safety [36, 120] do not consider temporal safety properties of heap memory. Most existing mechanisations also make the simplifying assumption that memory locations hold full objects rather than individual bytes like in MSWasm. Alignment concerns are responsible for part of the complexity of our definition.

Allocator Intepretation Let us discuss the interpretation for the allocator, shown in Fig. 3.19. It asserts that there exists a mapping f from handle ids to invariant names, such that this mapping agrees with the fragments from the handle value interpretation, and for every binding in this map, there is a corresponding binding in the allocator and a corresponding *partial* allocator resource. If that binding is to a live handle, then we additionally require that we hold the token that will allow us to open the cancellable invariant from the handle value interpretation.

In other words, the handle value interpretation holds the spatial resources inside a cancellable invariant, and the allocator resource holds the token that allows to open

[6]This bound may be greater than the current bounds — recall that handle slicing makes it safe to share any of its sub-bounds.

[7]The handle that was originally allocated might have strictly larger bounds than the handle that was originally shared, as is the case in the running buffer example.

$\boxed{\mathscr{A} : \mathit{Allocator} \to \mathit{iProp}}$

$$\begin{array}{rcl}
\mathrm{cinvOpt}(\mathit{base}, \mathit{bound}, y) & \triangleq & \begin{cases} \mathit{base} = b' * \mathit{bound} = e' & \text{if } y = \mathsf{Some}(b', e') \\ \quad *[\mathrm{CInv} : \gamma] & \\ \top & \text{otherwise} \end{cases} \\
\mathscr{A}(\mathit{allctr}) & \triangleq & \exists f. \boxed{\bullet f}^{\gamma_{\mathrm{tok}}} * \\
 & & \ast_{(\mathit{id} \mapsto (\gamma, \mathit{base}, \mathit{bound}, q)) \in f} \; \exists y. \mathit{allctr}(\mathit{id}) = y * \\
 & & \quad \mathit{id} \xrightarrow{\texttt{allocated}}{}^{q} y * \\
 & & \quad \mathrm{cinvOpt}(\mathit{base}, \mathit{bound}, y)
\end{array}$$

Figure 3.11: Our logical relation for the allocator

said invariant as long as the handle is live, thereby maintaining the temporal authority. The former expresses persistent knowledge over segment memory, while the latter expresses non-duplicable knowledge of the allocator.

The allocator resource is partial with degree q, meaning it can only be modified if $q = 1$, else it can only be inspected but not updated. This means that code looking to free a handle (i.e. update the resource from $\mathsf{Some}(\mathit{base}, \mathit{bound})$ to None) must own $q = 1$.

Allocator and Handle Interpretation Together Let us assume we own the interpretation for a handle value h together with the interpretation for an allocator, and see how we can reason about running **segload**, **segstore** or **segfree** on h.

First, we proceed by cases on $h.\mathsf{valid}$: if it is false, then all three instructions trap safely. Else, we now hold two ghost resources: one from the value interpretation of the handle, and one from the interpretation for the allocator. Combining them yields a binding $h.\mathsf{id} \mapsto \gamma, \mathit{base}'', \mathit{bound}'', q$ for which the allocator interpretation gives us a corresponding binding in the allocator, as well as an allocator resource $h.\mathsf{id} \xrightarrow{\texttt{allocated}}{}^{q} y$. We can then perform a case distinction on y: if it is None then the handle has been freed and all three instructions will safely trap; else, the allocator interpretation gives a token that can be used to open the invariant in the value interpretation for the handle.

Once the invariant is open, we hold both the segment resources for the area of memory pointed by h, and an allocator resource for $h.\mathsf{id}$. This is enough to perform a read or a write on h. In that case, y has remained unchanged, so the invariant can be trivially closed again, giving back the token for the interpretation for the allocator.

In the last case, if the instruction is a **segfree**, we must proceed by case on whether $h.\mathsf{base}$ and $h.\mathsf{bound}$ are equal to the values present in the allocator (which the cinvOpt in the allocator interpretation tells us are equal to base'' and bound''). If they aren't, the freeing operation safely traps. If they are, we can cancel the invariant instead of closing it. The value interpretation mandates that $q = 1$ and hence we can update the allocator resource to $h.\mathsf{id} \xrightarrow{\texttt{allocated}} \mathsf{None}$, which we can use to restore the interpretation for the allocator without needing the cancellable invariant token.

$$\boxed{\mathscr{F}rame[\![ts]\!]_{inst} : Frame \to iProp}$$

$$\mathscr{F}rame[\![ts]\!]_{inst}(F) \triangleq [\text{NaInv} : \top] * \xhookrightarrow{\text{FR}} F * \exists vs.\, F = \{inst; vs\} * \mathscr{V}[\![ts]\!](\texttt{immV}\ vs)$$

$$\boxed{\mathscr{E}[\![ts]\!]^{*}_{*} : Expr \to iProp}$$

$$\mathscr{E}[\![ts]\!]^{\tau_{\texttt{lbs}},\tau_{\texttt{ret}}}_{(\tau l,inst,hfs)}(lh,es) \triangleq \mathsf{wp}\, es \left\{ w,\ \begin{array}{c} \left(\begin{array}{l} \mathscr{V}[\![ts]\!](w) \vee \mathscr{H}[\![ts]\!]^{\tau_{\texttt{lbs}},\tau_{\texttt{ret}}}_{(\tau l,inst,hfs)}(w) \vee \\ \mathscr{B}r[\![\tau_{\texttt{lbs}}]\!]^{\tau_{\texttt{ret}}}_{(\tau l,inst,hfs)}(w,lh) \vee \\ \mathscr{R}et[\![\tau_{\texttt{ret}}]\!]_{(\tau l,inst)}(w) \end{array}\right) * \\ \exists F, allctr.\, \mathscr{F}rame[\![\tau l]\!]_{inst}(F) * \mathscr{A}(allctr) \end{array} \right\}$$

$$\boxed{C \vDash es : ts1 \to ts2}$$

$$\begin{array}{rl} C \vDash es : ts1 \to ts2 \triangleq & \forall inst, lh, hfs.\, (\mathscr{I}[\![C]\!](inst) * \mathscr{C}tx[\![C]\!]_{(inst,hfs)}(lh)) \mathrel{-\!\!*} \\ & \forall F, allctr, vs.\, (\mathscr{V}[\![ts1]\!](vs) * \xhookrightarrow{\text{FR}} F * \\ & \mathscr{F}rame[\![\tau l]\!]_{inst}(F) * \mathscr{A}(allctr)) \mathrel{-\!\!*} \\ & \mathscr{E}[\![ts2]\!]^{\tau_{\text{lbs}},\tau_{\text{ret}}}_{(\tau l,inst,hfs)}(lh, vs +\!\!+ es) \end{array}$$

where $\tau l = C.\mathsf{locals}$, $\tau_{\text{lbs}} = C.\mathsf{labels}$, and $\tau_{\text{ret}} = C.\mathsf{return}$.

Figure 3.12: Excerpts from the definition of our logical relation

As illustrated above, all the necessary resources are obtainable when holding the allocator interpretation and the value interpretation for handles. Hence we do not need an interpretation for the full segment memory, unlike for linear memory. This reflects the fine-grained reasoning that segment memory allows: memory is never considered in its entirety, but only handle by handle.

Expression Interpretation Fig. 3.21 shows excerpts from the definition of the Iris-Wasm logical relation, with the few modifications brought by Iris-MSWasm in magenta. These modifications are the addition of the allocator and corresponding allocator interpretation. We use a weakest precondition to define that an expression is safe to run. During the execution of a WebAssembly expression, the expression might not terminate on a WebAssembly value, but rather on a `br` or `return` instruction, or on a host call; hence the four-way disjunction in the postcondition. We give definitions for $\mathscr{H}$, $\mathscr{B}r$, and $\mathscr{R}et$ in our supplementary material. The post-condition also yields back frame and allocator resources. This expression interpretation is most interesting when considered together with the definition of *semantic typing*, also given in Fig. 3.21. An expression is semantically well typed (written with a double turnstile $\vDash$ instead of the simple turnstile $\vdash$ used for syntactic typing) when, given a context and instance that are safe to use (see our supplementary material for definitions of $\mathscr{I}$ and $\mathscr{C}tx$), as well as arguments that are safe to share, a frame, and an allocator, the resulting expression is in the expression interpretation.

We can now state the fundamental theorem of the logical relation:

Theorem 7 (Fundamental theorem of the logical relation)**.** *If a program bs (a list of* basic instructions*, i.e. only using instructions available to the programmer) typechecks syntactically, then it typechecks semantically:*

$$\forall bs, C, ts1, ts2. C \vdash bs : ts1 \to ts2 \implies C \vDash bs : ts1 \to ts2$$

Proof sketch. The proof proceeds by induction on the syntactic typing judgement. The added challenge with respect to its prior version is to prove the cases for the new segment instructions, each of which depend on the new value relation for handles, and the interpretation of the allocator. A full proof can be found in the Coq development. □

3.4.3 Robust Safety

Buffer Example Let us come back to our buffer example from Fig. 3.1 and show how we can reason about the call to the unknown, untrusted function $adv. All we assume is that this function is well typed in MSWasm's typing system, with type $[\mathbf{handle}] \to []$.

Jumping back into the proof detailed in §3.3.3, right before the call, we own the following resources:

$$\left\{ \begin{array}{l} \xhookrightarrow{\text{FR}} F'' * \overset{\texttt{wss}}{\longmapsto}_{h.\mathsf{base}} \mathrm{serialise}(\mathbf{i32}, 42) * \overset{\texttt{wss}}{\longmapsto}_{h.\mathsf{base}+4} \mathrm{repeat}(4,0) * \\ h.\mathsf{id} \xrightarrow{\texttt{allocated}} \mathsf{Some}\ x \end{array} \right\}$$

Using the second segment memory resource, we can instantiate an invariant and allocate a ghost resource, giving us $h' \in \mathcal{V}[\![[\mathbf{handle}]]\!]$. To get $\mathscr{A}(\{h.\mathsf{id} \mapsto \mathsf{Some}(h.\mathsf{base}, h.\mathsf{bound})\})$, we need to also give an allocator resource. Just like in §3.3.3, we can separate our allocator resource into two fragment resources $h.\mathsf{id} \xrightarrow{\texttt{allocated}}{}^{1/2} \mathsf{Some}\ x$, and only give one of these to get the $\mathscr{A}$ statement; this allows us to keep partial ownership which lets us know that the handle cannot have been freed.

Hence we can apply the fundamental theorem, and know that our call executes safely, and terminates on a value that is safe to share for type $[]$, i.e. the trap value or the unit value. We also get that there exists a new allocator $allctr'$ such that $\mathscr{A}(allctr')$. The segment resource $\overset{\texttt{wss}}{\longmapsto}_{h.\mathsf{base}} \mathrm{serialise}(\mathbf{i32}, 42)$ was unused and hence the caller has held on to them, and we can use these to complete the proof of the specification.

Robust Safety This approach, where we leverage the fundamental theorem of the logical relation to prove specifications in the presence of unknown code, allows us to call library functions from untrusted libraries safely, establishing invariants of the form "No matter what untrusted module calls the functions I export, my internal state will satisfy this invariant". This showcases the strength of MSWasm and the fine-grained safety properties it brings to WebAssembly.

3.5 Stack Example

In this section, we illustrate how our program logic and logical relation scale to a bigger example: a library implementing stacks of **i32** integers. This library builds on a case study from Iris-Wasm [89], but uses handles to enforce stronger guarantees. Since plain WebAssembly does not have handles, the stack library of Rao et al. [89] uses **i32** integers to represent stacks. These values are forgeable, hence the stack library must be encapsulated from untrusted code to prevent the corruption of allocated stacks — technically, by instantiating the adversary without access to the functions of the stack library. In MSWasm, we use handles instead of **i32** integers to represent stacks. With handles, the adversary cannot corrupt the stacks even when it has access to the stack library — technically, when it is instantiated after the stack library, with access to its functions — and we prove this using our logical relation.

Our stack module defines a function $new_stack which uses the **segalloc** instruction to allocate one page (64KiB) of segment memory. Handles to this stack point to the start of this page, and range over all of it. In the first four bytes of the allocated region, we store the *stack pointer* as an offset to the top of the stack, initially the **i32** integer 0. When accessing a stack, we get the offset by loading from the handle, and then combine the handle with the offset by **handle.add** to address the top of the stack.

From here, it is straightforward to define the usual stack operations like $push, $pop, $stack_length, $is_full, and $is_empty. In addition, we define $stack_map, which takes as arguments a stack and a function (more precisely, an index in a table of functions) that it maps on all the elements of the stack. This map function is interesting, because when the function it maps is an adversary function, the execution context has authority over the stack.

In our Coq development, we verify the stack module, and exercise it on key scenarios using different client modules. Here, we focus on a specific client module, $RobustModule in Fig. 3.13, to showcase robust capability safety. This module creates a stack, pushes two values onto it, maps an adversary function (imported from an untrusted *adversary module*) onto the stack, and then asks for the stack's length. We wish to prove that mapping the adversary function does not affect the stack's length. Fig. 3.13 shows the sequence of instantiations performed by the host code.

This example is interesting because both the adversary module and $RobustModule have access to the stack functions and can thus interact with the stack module's memory. Importantly, the adversary function will only be given *values* fetched from the stack module's memory by the $stack_map function, and not handles. If it had access to a handle, the adversary might be able to, say, pop elements from $RobustModule's stack, and then the final length of the stack would change. This form of attack is made impossible by the fact that the handles that represent stacks are unforgeable. Because $RobustModule never shares its handle with the adversary module, the adversary module cannot push, pop, or perform any operations on *that* stack; it may only use the stack module to create its own stacks and perform operations on those. This showcases the strength MSWasm adds to plain WebAssembly.

We prove the following theorem:

Theorem 8 (Robust stack example)**.** *If the host code h_{code} from Fig. 3.13 terminates, it terminates on either the trap value* `trapV`*, or on the* **i32** *value 2.*

In particular, this means the adversary cannot push or pop on the client module's stack. A full proof can be found in our Coq development, and we give a succinct overview here.

Proof sketch. In essence, the proof is similar to the one for the buffer example from §3.3.4. However, since there are multiple modules involved and since some resources must be placed in invariants to apply the fundamental theorem of the logical relation, the order in which the steps on the proof is carried out is crucial.

We begin by proving specifications for all the functions of the stack module. Since this is known code, the proofs are similar to that of §3.3.3.

We then apply the instantiation lemma three times in a row. First we instantiate the stack module, which does not make any imports, and hence we do not need any resources; the lemma gives us resources corresponding to each individual function closure. Then we instantiate the adversary module. The instantiation lemma requires function closure resources for the stack functions, which we have, and gives these resources back as well as an extra function closure resource corresponding to the adversary function $advf. Finally, we instantiate $RobustModule. Again, the instantiation lemma requires function closure resources for the stack functions as well as the $advf function, and gives these resources back together with an extra function closure resource corresponding to the main function of $RobustModule.

All that remains to do is run the code of $RobustModule. Like in the buffer example, we need to apply the fundamental theorem in order to reason about the unknown function $advf. One additional subtlety we face here is that because the adversary module imports the stack functions, we must first show that these functions are safe to share. These functions are defined in the stack module, hence we must prove all components of that module to be safe. This actually includes the adversary function $advf itself, since that function was placed in the stack module's function table when instantiating $RobustModule. Since these functions can call each other, there is a circularity, which we address (as is standard) by Löb induction. To do this, we need to allocate invariants corresponding to all objects that will need to be proved safe to share. The induction then gives us that the adversary function $advf is safe to share, and in particular we have a weakest precondition that we can use to reason about calls to it.

Using this, we can specify the code of $RobustModule like we did for the buffer example in §3.3.4 and obtain a weakest precondition. Finally, we apply the adequacy theorem from §3.3.4 to get the desired result.

□

$RobustModule:

```
Create a stack s
Push 42 and 10 onto s
Map $advf onto s
Set x := length(s)
Return x
```

Host code h_{code}:

Instantiate $stackmodule
    No imports
    Export *stack functions*
Instantiate $advmodule
    Import *stack functions*
    Export $advf
Instantiate $RobustModule
    Import *stack functions*
    Import $advf
    No exports

Figure 3.13: Pseudo-code for the robust stack client, and host code for the instantiation sequence

3.6 Discussion and Related Works

We discuss prior work that we build on (§3.6.1), and then return to the question of sharing state (§3.6.2)

3.6.1 Prior Work

Iris-Wasm MSWasmCert is a conservative extension of WasmCert, and Iris-MSWasm is accordingly an extension of Iris-Wasm. In particular, we inherit all the separation logic proof rules for the constructs of WebAssembly, and add new proof rules for the new constructs of MSWasm. Our logical relation is correspondingly an extension of the logical relation of Iris-Wasm: in the absence of handles and segment memory, it collapses to the logical relation of Iris-Wasm.

Cerise Iris-Wasm uses the same ideas as Cerise, but in the setting of WebAssembly rather than that of capability machines. The main differences are that: the MSWasm allocator enforces temporal safety; MSWasm only feature a single type of permission, which corresponds to an 'RW' write-and-read permission (considering other types of permissions would be interesting); and MSWasm functions and their local arguments are handled by the language, not implemented using capabilities, so MSWasm does not feature function-flavoured capabilities (executable permission, stack-local capabilities, etc.). However, the most significant difference is not one of feature, but of scale: as illustrated in §3.4.2, our logical relation needs to cover all the language constructs of MSWasm, including frames, the module system, etc., and these pose a significant challenge.

MSWasm's Memory Safety When introducing MSWasm, Michael et al. [69] propose a new formal, colour-based definition of memory safety, that captures spatial and temporal memory safety, and pointer integrity. Their definition hinges on a

monitor that inspects the execution trace of the program, and checks that the memory accesses agree with the colouring. Concretely, the monitor maintains a *shadow memory* that associates every address with its allocation state (either Allocated or Free), its colour (an arbitrary identifier), and its shade (for intra-object safety). Using the shadow memory as reference, the monitor then checks every event in the trace, making sure that 1. any read and write events are performed on allocated addresses, using the right colour and shade, 2. allocation events only allocate free addresses, and 3. freeing events only free allocated addresses, for which a corresponding allocation event exists in the trace, and no free event since. A trace is said to be *memory safe* if the monitor does not get stuck. While this approach allows them to capture a notion of memory safety, it does not directly make it possible to reason about the combination of known and unknown, potentially adversarial code. First, the monitor does not distinguish events emitted by trusted modules, and events emitted by untrusted ones. In other words, the monitor does not have any notion of private and public state. In the example of §3.1.1, the monitor does not know whether the $\mathtt{read}(h)$ event comes from the known code, or if it comes from the adversary: the monitor accepts the trace in both cases, as the event is legal according to the shadow memory. Second, the monitor definition does not keep track of the values read and written by the memory events. This is especially limiting for reasoning about functional properties, and for keeping track of how values are preserved throughout adversary calls. In the example of §3.1.1, the colour-based monitor does not check whether the value stored and read by the handle h is 42.

In this paper, we are interested in fine-grained interactions between trusted, known code and untrusted, unknown, arbitrary code. Those properties usually require keeping track of the values preserved throughout adversary calls, by carefully over-approximating the behaviour of the adversary. Adapting the monitor-based definition to tractably bound the set of traces that the adversary can generate would require addressing the frame problem [67].

Our notion of *capability safety* is built on top of a separation logic, and takes account of ownership of resources. It offers an explicit distinction between private and public state: values shared with unknown code need to be owned by the logical relation. As explained in section §3.4.2, the logical relation recursively computes the addresses reachable from a given value. Giving ownership of an address over to the logical relation gives away knowledge of its contents, as its value is now existentially quantified. Crucially, the frame rule of separation logic keeps track of the private state during an adversary call: the private value are simply framed away.

Michael et al. [69] use their monitor-based definition of memory safety in the context of secure compilation, which we do not explore in this paper.

3.6.2 Sharing State in WebAssembly

The rigid nature of WebAssembly 1.0 means that C cannot be compiled in the ‘naive’ way to WebAssembly. For example, C local variables are too expressive to be compiled to WebAssembly locals, and therefore most production C-to-WebAssembly

compilers compile the C stack to a data structure in linear memory. Lehmann et al. [61] illustrate how this and other limitations mean that many isolation mechanisms provided by usual OS infrastructure for process hardening are not available when compiling to WebAssembly.

As described in the introduction, capabilities are one approach to address this. However, they raise some challenges: common compiler optimisations violate capability safety [143], and so writing optimising capability-safety-preserving compilers is an open problem; and capability compression on hardware causes a mismatch between source and target languages. This is particularly problematic for MSWasm, as it is meant as an intermediate language, both compiled to, and compiled from. Nonetheless, by making MSWasm precise and proving that it satisfies robust capability safety, we ensure that projects exploring its use as an intermediate language can rely on its design being validated, and can know exactly what MSWasm guarantees.

RichWasm [30] extends WebAssembly with a static notion of capability, at the type level instead of at runtime, and use them to statically enforce safe fine-grained sharing.

As a closely related approach, WebAssembly is being enriched with aggregate types [100] that WebAssembly 2.0 references can point to. WebAssembly references are opaque and unforgeable, and as such act as a simple form of capabilities. Developing a logical relation that captures both handles and references would make it possible to make a more formal comparison.

A very different approach, taken by the WebAssembly Component Model [122, 123] and adopted by several vendors of WebAssembly for cloud computing is to eschew sharing in favour of copying. One of the aims of the WebAssembly Component Model is to allow language interoperability, which typically requires marshalling, and so already incurs the cost of copying.

Our work lays the foundation to evaluate the language-level guarantees of these different approaches, and we hope that it informs future developments and deployments.

3.6.3 Semantic Language Integrity

Maintaining datatype and notation consistency in a large language specification is challenging, especially if one wants to be able to automatically extract human-readable rules, an interpreter for testing, and theorem prover definitions [74, 81, 108]. WebAssembly is now getting a DSL [9] for that specific purpose.

However, maintaining the semantic integrity of the language is as important as maintaining its syntactic integrity. The key properties that form the *universal contract* of the language make it possible for specifiers, implementers, and users to work together instead of against each other. In this paper, we demonstrated how to capture key aspect of such a universal contract for as complex an extension of WebAssembly as MSWasm.

Acknowledgements

This work was supported in part by a Villum Investigator grant (no. 25804), Center for Basic Research in Program Verification (CPV), from the VILLUM Foundation, for Birkedal, and by an AUFF Starter Grant for Pichon-Pharabod. This work was co-funded by the European Union (ERC, CHORDS, 101096090). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

Data Availability Statement

The artifact [60] of this paper, containing the full Coq development, is available on Zenodo. Detailed instructions of usage are provided within the artifact itself. The appendix, which contains the full version of figures that had to be shortened in the paper, can be found together with the artifact.

The code is also available on github at https://github.com/logsem/MSWasm.

Appendices

In these appendices, we present figures that complete our paper.

3.A Proof rules

$$\texttt{wp\_handleadd}\quad \frac{h.\mathsf{offset}+c\geq 0 * h' = \{h \text{ with } \mathsf{offset} = h.\mathsf{offset}+c\} * \triangleright\Phi(\texttt{immV}\ [\textbf{handle.const}\ h']) * \xhookrightarrow{\text{FR}} F}{\mathsf{wp}\ [\textbf{i32.const}\ c;\ \textbf{handle.const}\ h;\ \textbf{handle.add}]\left\{w, \Phi(w) * \xhookrightarrow{\text{FR}} F\right\}}$$

$$\texttt{wp\_handleadd\_failure}\quad \frac{h.\mathsf{offset}+c<0 * \triangleright\Phi(\texttt{trapV}) * \xhookrightarrow{\text{FR}} F}{\mathsf{wp}\ [\textbf{i32.const}\ c;\ \textbf{handle.const}\ h;\ \textbf{handle.add}]\left\{w, \Phi(w) * \xhookrightarrow{\text{FR}} F\right\}}$$

$$\texttt{wp\_slice}\quad \frac{0\leq c_1 < h.\mathsf{bound} * c_1\leq c_2 * \quad h' = \{h \text{ with } \mathsf{base} = h.\mathsf{base}+c_1 \text{ and } \mathsf{bound} = h.\mathsf{bound}-c_2\} * \triangleright\Phi(\texttt{immV}\ [\textbf{handle.const}\ h']) * \xhookrightarrow{\text{FR}} F}{\mathsf{wp}\ [\textbf{handle.const}\ h;\ \textbf{i32.const}\ c_1;\ \textbf{i32.const}\ c_2;\ \textbf{slice}]\left\{w, \Phi(w) * \xhookrightarrow{\text{FR}} F\right\}}$$

$$\texttt{wp\_slice\_failure}\quad \frac{\left(c_1<0 \vee c_1\geq h.\mathsf{bound} \vee c_1 > c_2\right) * \triangleright\Phi(\texttt{trapV}) * \xhookrightarrow{\text{FR}} F}{\mathsf{wp}\ [\textbf{handle.const}\ h;\ \textbf{i32.const}\ c_1;\ \textbf{i32.const}\ c_2;\ \textbf{slice}]\left\{w, \Phi(w) * \xhookrightarrow{\text{FR}} F\right\}}$$

Figure 3.14: Our proof rules for handle instructions

wp_segload

$$\frac{t \neq \textbf{handle} \ * \ h.\mathsf{id} \xrightarrow{\texttt{allocated}}{}^{q} \mathsf{Some}(x) \ * \ h.\mathsf{offset} + \mathrm{sizeof}(t) \leq h.\mathsf{bound} \ * \ addr = h.\mathsf{base} + h.\mathsf{offset} \ * \ h.\mathsf{valid} = \textbf{true} \ * \ \mathrm{deserialise}(t, \mathrm{untag}(tbs)) = v \ * \ \triangleright \Phi(\texttt{immV}\ [v]) \ * \ \xhookrightarrow{\text{FR}} F}{\mathsf{wp}\,[\textbf{handle.const}\ h; t.\textbf{segload}] \left\{ w, \begin{array}{l} \Phi(w) \ * \ \overset{\texttt{wss}}{\longmapsto}_{addr} tbs \ * \\ h.\mathsf{id} \xrightarrow{\texttt{allocated}}{}^{q} \mathsf{Some}(x) \ * \ \xhookrightarrow{\text{FR}} F \end{array} \right\}}$$

wp_segload_handle

$$\frac{t = \textbf{handle} \ * \ \overset{\texttt{wss}}{\longmapsto}_{addr} tbs \ * \ h.\mathsf{id} \xrightarrow{\texttt{allocated}}{}^{q} \mathsf{Some}(x) \ * \ h.\mathsf{offset} + \mathrm{sizeof}(t) \leq h.\mathsf{bound} \ * \ \mathrm{aligned}(addr, \mathrm{handle\_size}) \ * \ b = \mathrm{allHandle}(\mathrm{tags}(tbs)) \ * \ h_f = \mathrm{updateValid}(h', b \wedge h'.\mathsf{valid}) \ * \ addr = h.\mathsf{base} + h.\mathsf{offset} \ * \ h.\mathsf{valid} = \textbf{true} \ * \ \mathrm{deserialise}(t, \mathrm{untag}(tbs)) = h' \ * \ \triangleright \Phi(\texttt{immV}\ [h_f]) \ * \ \xhookrightarrow{\text{FR}} F}{\mathsf{wp}\,[\textbf{handle.const}\ h; t.\textbf{segload}] \left\{ w, \begin{array}{l} \Phi(w) \ * \ \overset{\texttt{wss}}{\longmapsto}_{addr} tbs \ * \\ h.\mathsf{id} \xrightarrow{\texttt{allocated}}{}^{q} \mathsf{Some}(x) \ * \ \xhookrightarrow{\text{FR}} F \end{array} \right\}}$$

wp_segload_failure1

$$\frac{\Big(h.\mathsf{offset} + \mathrm{sizeof}(t) > h.\mathsf{bound} \ \vee \ h.\mathsf{valid} = \textbf{false} \ \vee \ (t = \textbf{handle} \wedge \neg \mathrm{aligned}(h.\mathsf{base} + h.\mathsf{offset}, \mathrm{handle\_size})) \Big) \ * \ \triangleright \Phi(\texttt{trapV}) \ * \ \xhookrightarrow{\text{FR}} F}{\mathsf{wp}\,[\textbf{handle.const}\ h; t.\textbf{segload}] \left\{ w, \Phi(w) \ * \ \xhookrightarrow{\text{FR}} F \right\}}$$

wp_segload_failure2

$$\frac{h.\mathsf{id} \xrightarrow{\texttt{allocated}}{}^{q} \mathsf{None} \ * \ \triangleright \Phi(\texttt{trapV}) \ * \ \xhookrightarrow{\text{FR}} F}{\mathsf{wp}\,[\textbf{handle.const}\ h; t.\textbf{segload}] \left\{ w, \Phi(w) \ * \ h.\mathsf{id} \xrightarrow{\texttt{allocated}}{}^{q} \mathsf{None} \ * \ \xhookrightarrow{\text{FR}} F \right\}}$$

Figure 3.15: Our proof rules for the **segload** instruction

wp_segstore

$$\frac{t \neq \textbf{handle} \ * \ \overset{\texttt{wss}}{\longmapsto}_{addr} tbs \ * \ h.\mathsf{id} \xrightarrow{\texttt{allocated}}{}^{q} \mathsf{Some}(x) \ * \\ h.\mathsf{offset} + \mathrm{sizeof}(t) \leq h.\mathsf{bound} \ * \ addr = h.\mathsf{base} + \mathsf{offset} \ * \ h.\mathsf{valid} = \textbf{true} \ * \\ \textbf{typeof}(v) = t \ * \ |bs| = \mathrm{sizeof}(t) \ * \ \mathsf{serialise}(t, v) = bs \ * \\ \mathrm{addTag}(bs, \textbf{Numeric}) = tbs' \ * \ \triangleright \Phi(\texttt{immV}\ []) \ * \ \hookrightarrow^{\mathrm{FR}} F}{\mathsf{wp}\,[\textbf{handle.const}\ h; v; t.\textbf{segstore}] \left\{ w, \begin{array}{l} \Phi(w) \ * \ \overset{\texttt{wss}}{\longmapsto}_{addr} tbs' \ * \\ h.\mathsf{id} \xrightarrow{\texttt{allocated}}{}^{q} \mathsf{Some}(x) \ * \ \hookrightarrow^{\mathrm{FR}} F \end{array} \right\}}$$

wp_segstore_handle

$$\frac{t = \textbf{handle} \ * \ \overset{\texttt{wss}}{\longmapsto}_{addr} tbs \ * \ h.\mathsf{id} \xrightarrow{\texttt{allocated}}{}^{q} \mathsf{Some}(x) \ * \\ h.\mathsf{offset} + \mathrm{sizeof}(t) \leq h.\mathsf{bound} \ * \ addr = h.\mathsf{base} + \mathsf{offset} \ * \ h.\mathsf{valid} = \textbf{true} \ * \\ \mathrm{aligned}(addr, \mathrm{handle\_size}) \ * \ |bs| = \mathrm{sizeof}(t) \ * \ \mathsf{serialise}(t, h') = bs \ * \\ \mathrm{addTag}(bs, \textbf{Handle}) = tbs' \ * \ \triangleright \Phi(\texttt{immV}\ []) \ * \ \hookrightarrow^{\mathrm{FR}} F}{\mathsf{wp} \left[\begin{array}{l} \textbf{handle.const}\ h; \\ \textbf{handle.const}\ h'; \\ t.\textbf{segstore} \end{array} \right] \left\{ w, \begin{array}{l} \Phi(w) \ * \ \overset{\texttt{wss}}{\longmapsto}_{addr} tbs' \ * \\ h.\mathsf{id} \xrightarrow{\texttt{allocated}}{}^{q} \mathsf{Some}(x) \ * \ \hookrightarrow^{\mathrm{FR}} F \end{array} \right\}}$$

wp_segstore_failure1

$$\frac{\Big(h.\mathsf{offset} + \mathrm{sizeof}(t) > h.\mathsf{bound} \ \vee \ h.\mathsf{valid} = \textbf{false} \ \vee \\ (t = \textbf{handle} \wedge \neg \mathrm{aligned}(h.\mathsf{base} + h.\mathsf{offset}, \mathrm{handle\_size})) \Big) \\ * \ \triangleright \Phi(\texttt{trapV}) \ * \ \hookrightarrow^{\mathrm{FR}} F}{\mathsf{wp}\,[\textbf{handle.const}\ h; v; t.\textbf{segstore}] \left\{ w, \Phi(w) \ * \ \hookrightarrow^{\mathrm{FR}} F \right\}}$$

wp_segstore_failure2

$$\frac{h.\mathsf{id} \xrightarrow{\texttt{allocated}}{}^{q} \mathsf{None} \ * \ \triangleright \Phi(\texttt{trapV}) \ * \ \hookrightarrow^{\mathrm{FR}} F}{\mathsf{wp}\,[\textbf{handle.const}\ h; v; t.\textbf{segstore}] \left\{ w, \Phi(w) \ * \ h.\mathsf{id} \xrightarrow{\texttt{allocated}}{}^{q} \mathsf{None} \ * \ \hookrightarrow^{\mathrm{FR}} F \right\}}$$

Figure 3.16: Our proof rules for the **segstore** instruction

wp_segfree

$$\frac{h.\mathsf{valid} = \mathbf{true} \ * \ h.\mathsf{offset} = 0 \ * \ |tbs = b| \ * \ \overset{\texttt{wss}}{\longmapsto}_{h.\mathsf{base}} tbs \ * \ h.\mathsf{id} \ \overset{\texttt{allocated}}{\rightarrow} \mathsf{Some}(h.\mathsf{base}, b) \ * \ \triangleright \Phi(\texttt{immV}\ []) \ * \ \overset{\mathrm{FR}}{\hookrightarrow} F}{\mathsf{wp}\,[\mathbf{handle.const}\ h; \mathbf{segfree}] \left\{ w, \Phi(w) \ * \ \overset{\mathrm{FR}}{\hookrightarrow} F \right\}}$$

wp_segfree_failure1

$$\frac{\left(h.\mathsf{valid} = \mathbf{false} \ \vee \ h.\mathsf{offset} \neq 0\right) \ * \ \triangleright \Phi(\texttt{trapV}) \ * \ \overset{\mathrm{FR}}{\hookrightarrow} F}{\mathsf{wp}\,[\mathbf{handle.const}\ h; \mathbf{segfree}] \left\{ w, \Phi(w) \ * \ \overset{\mathrm{FR}}{\hookrightarrow} F \right\}}$$

wp_segfree_failure2

$$\frac{h.\mathsf{id} \ \overset{\texttt{allocated}}{\rightarrow}^{q} \mathsf{None} \ * \ \triangleright \Phi(\texttt{trapV}) \ * \ \overset{\mathrm{FR}}{\hookrightarrow} F}{\mathsf{wp}\,[\mathbf{handle.const}\ h; t.\mathbf{segfree}] \left\{ w, \Phi(w) \ * \ h.\mathsf{id} \ \overset{\texttt{allocated}}{\rightarrow}^{q} \mathsf{None} \ * \ \overset{\mathrm{FR}}{\hookrightarrow} F \right\}}$$

wp_segfree_failure3

$$\frac{h.\mathsf{id} \ \overset{\texttt{allocated}}{\rightarrow}^{q} \mathsf{Some}(base, bound) \ * \ (base \neq h.\mathsf{base} \ \vee \ bound \neq h.\mathsf{bound}) \ * \ \triangleright \Phi(\texttt{trapV}) \ * \ \overset{\mathrm{FR}}{\hookrightarrow} F}{\mathsf{wp}\,[\mathbf{handle.const}\ h; t.\mathbf{segfree}] \left\{ w, \begin{array}{l} \Phi(w) \ * \ \overset{\mathrm{FR}}{\hookrightarrow} F \ * \\ h.\mathsf{id} \ \overset{\texttt{allocated}}{\rightarrow}^{q} \mathsf{Some}(base, bound) \end{array} \right\}}$$

wp_segalloc

$$\frac{\begin{array}{l} \triangleright \Big(\forall w, \big(\exists h, w = \texttt{immV}\ [\mathbf{handle.const}\ h] \ * \ (h.\mathsf{valid} = \mathbf{false} \ \vee \\ (\mathsf{id} \ \overset{\texttt{allocated}}{\rightarrow} \mathsf{Some}(h.\mathsf{base}, n) \ * \ h.\mathsf{bound} = n \ * \ h.\mathsf{offset} = 0 \ * \\ h.\mathsf{valid} = \mathbf{true} \ * \ \overset{\texttt{wss}}{\longmapsto}_{h.\mathsf{base}} \mathrm{repeat}(0)))) \mathrel{-\!\!*} \Phi(w) \Big) \ * \ \overset{\mathrm{FR}}{\hookrightarrow} F \end{array}}{\mathsf{wp}\,[\mathbf{i32.const}\ n; \mathbf{segalloc}] \left\{ w, \Phi(w) \ * \ \overset{\mathrm{FR}}{\hookrightarrow} F \right\}}$$

Figure 3.17: Our proof rules for the **segalloc** and **segfree** instructions

3.B Syntactic Typing for MSWasm

MSWasm completes the syntactic type system of plain WebAssembly. The typing statement is of the form $C \vdash \mathit{instruction} : \mathit{ts} \to \mathit{ts}$. The context C is never useful in MSWasm-specific instructions.

The syntactic typing rules for MSWasm are given by:

$$
\begin{array}{llcrcl}
C \vdash & t.\mathbf{segload} & : & [\mathbf{handle}] & \to & [t] \\
C \vdash & t.\mathbf{segstore} & : & [\mathbf{handle}; t] & \to & [] \\
C \vdash & t.\mathbf{segalloc} & : & [\mathbf{i32}] & \to & [\mathbf{handle}] \\
C \vdash & t.\mathbf{segfree} & : & [\mathbf{handle}] & \to & [] \\
C \vdash & t.\mathbf{handle.add} & : & [\mathbf{i32}; \mathbf{handle}] & \to & [\mathbf{handle}] \\
C \vdash & t.\mathbf{slice} & : & [\mathbf{handle}; \mathbf{i32}; \mathbf{i32}] & \to & [\mathbf{handle}]
\end{array}
$$

For other instructions, the rules are identical to the ones in WebAssembly, except for **load** for which we add a $t \neq \mathbf{handle}$ constraint:

$$
\frac{t \neq \mathbf{handle}}{C \vdash t.\mathbf{load} : [\mathbf{i32}] \to [t]}
$$

3.C Logical Relation

$\boxed{\mathscr{V}[\![ts]\!] : \mathit{LogVal} \to \mathit{iProp}}$

$$\mathrm{ValidHandleAddr}(addr, base', bound') \triangleq \begin{array}{l}\mathrm{aligned}(base' + addr, \mathrm{handle\_size}) \wedge \\ 0 \le addr \wedge addr + \mathrm{handle\_size} \le bound'\end{array}$$

$$\begin{array}{l}\mathscr{V}_0[\![\mathbf{handle}]\!](v) \quad \triangleq \\ \quad \exists h.\, v = \mathbf{handle.const}\ h * \\ \quad \left(\begin{array}{l} h.\mathsf{valid} = \mathbf{false} \ \vee \\ \exists \gamma, base', bound', base'', bound'', q. \\ [h.\mathsf{base}..h.\mathsf{base} + h.\mathsf{bound}) \subseteq [base'..base' + bound') * \quad (1) \\ [base'..base' + bound') \subseteq [base''..base'' + bound'') * \quad (2) \\ q \in \left\{\frac{1}{2}, 1\right\} * \left((h.\mathsf{base} = base'' * h.\mathsf{bound} = bound'') \implies q = 1\right) * \quad (3) \\ \boxed{\circ (h.\mathsf{id} \hookrightarrow (\gamma, base'', bound'', q))}^{\gamma_{\mathrm{oks}}} * \quad (4) \\ \boxed{\begin{array}{l}\exists tbs. |tbs| = bound' * \overset{\mathtt{wss}}{\longmapsto}_{base'} tbs * \\ \quad \forall addr. \mathrm{ValidHandleAddr}(addr, base', bound') \mathrel{-\!\!*} \\ \quad \left(\begin{array}{l}\left(\begin{array}{l}\exists off.\, 0 \le off < \mathrm{handle\_size} \wedge \\ \quad tbs[addr + off] = (-, \mathbf{Numeric})\end{array}\right) \vee \\ \mathscr{V}_0[\![\mathbf{handle}]\!](\mathbf{handle.const} \\ \quad \mathrm{deserialise\_handle}(\\ \quad\quad \mathrm{untag}(tbs[addr..addr + \mathrm{handle\_size}])))\end{array}\right)\end{array}}^{\gamma} \quad (5) \\ \left.\begin{array}{l}\end{array}\right)\end{array}\right)^{\mathsf{CInv}}\end{array}$$

$$\begin{array}{rcl}\mathscr{V}_0[\![t]\!](v) & \triangleq & \exists c.\, v = t.\mathtt{const}\ c \qquad (\text{for } t \neq \mathbf{handle}) \\ \mathscr{V}[\![[t_1, \cdots, t_n]]\!](w) & \triangleq & w = \mathtt{trapV} \vee \\ & & \exists v_1, \cdots, v_n.\, w = \mathtt{immV}\ [v_1, \cdots, v_n] \wedge \\ & & \mathscr{V}_0[\![t_1]\!](v_1) \wedge \cdots \wedge \mathscr{V}_0[\![t_n]\!](v_n)\end{array}$$

Figure 3.18: Our logical relation for values

$\boxed{\mathscr{A} : Allocator \to iProp}$

$$\begin{array}{rcl}
\mathrm{cinvOpt}(base, bound, y) & \triangleq & \begin{cases} base = b' * bound = e' & \text{if } y = \mathsf{Some}(b', e') \\ \quad * [\mathrm{CInv} : \gamma] & \\ \top & \text{otherwise} \end{cases} \\
\mathscr{A}(allctr) & \triangleq & \exists f. \boxed{\bullet f}^{\gamma_{\mathrm{tok}}} * \\
 & & \ast_{(id \mapsto (\gamma, base, bound, q)) \in f} \begin{array}{l} \exists y.\, allctr(id) = y * \\ id \xrightarrow{\texttt{allocated}}{}^{q} y * \\ \mathrm{cinvOpt}(base, bound, y) \end{array}
\end{array}$$

Figure 3.19: Our logical relation for the allocator

$\boxed{\mathscr{F}rame[\![ts]\!]_{inst} : Frame \to iProp}$

$$\begin{array}{rcl}
\mathscr{F}rame[\![ts]\!]_{inst}(F) & \triangleq & [\mathrm{NaInv} : \top] * \xhookrightarrow{\mathrm{FR}} F * \\
 & & \exists vs.\, F = \{inst; vs\} * \mathscr{V}[\![ts]\!](\texttt{immV}\ vs)
\end{array}$$

Figure 3.20: Our logical relation for the frame

$\boxed{\mathscr{E}_0[\![ts]\!]_* : Expr \to iProp}$ $\boxed{\mathscr{E}[\![ts]\!]^*_* : Lholed \times Expr \to iProp}$

$$\mathscr{E}_0[\![ts]\!]_{(F,hfs)}(es) \triangleq \mathsf{wp}\, es \left\{ w, \begin{array}{l} (\mathscr{V}[\![ts]\!](w) \vee \mathscr{H}[\![ts]\!]_{hfs}(w)) * [\mathrm{NaInv} : \top] \\ * \xhookrightarrow{\mathrm{FR}} F * \exists allctr.\, \mathscr{A}(allctr) \end{array} \right\}$$

$$\mathscr{E}[\![ts]\!]^{\tau_{\mathtt{lbs}}, \tau_{\mathtt{ret}}}_{(\tau l, inst, hfs)}(lh, es) \triangleq \mathsf{wp}\, es \left\{ w, \begin{array}{l} \left(\begin{array}{l} \mathscr{V}[\![ts]\!](w) \vee \\ \mathscr{H}[\![ts]\!]^{\tau_{\mathtt{lbs}}, \tau_{\mathtt{ret}}}_{(\tau l, inst, hfs)}(w) \vee \\ \mathscr{B}r[\![\tau_{\mathtt{lbs}}]\!]^{\tau_{\mathtt{ret}}}_{(\tau l, inst, hfs)}(w, lh) \vee \\ \mathscr{R}et[\![\tau_{\mathtt{ret}}]\!]_{(\tau l, inst)}(w) \end{array} \right) * \\ \exists F, allctr.\, \mathscr{F}rame[\![\tau l]\!]_{inst}(F) * \mathscr{A}(allctr) \end{array} \right\}$$

Figure 3.21: Our logical relation for expressions

$$\boxed{\mathscr{R}et[\![ts]\!]_{(\tau l,inst)} : Val \to iProp}$$

$$\begin{aligned}\mathscr{R}et[\![ts]\!]_{(\tau l,inst)}(w) \triangleq\ & \exists lh, v.\, w = lh[\mathbf{return}] * lh(0) = v * \\ & \exists \tau s'.\, \mathscr{V}[\![\tau s']\!](v) * \\ & \forall F, F', allctr.\, \xhookrightarrow{\text{FR}} F' * \mathscr{A}(allctr) \mathrel{-\!\!*} \\ & \mathsf{wp}\,[\mathbf{local}_n\{F\}\ w\ \mathbf{end}] \left\{ w,\ \begin{array}{l} \mathscr{V}[\![ts]\!](w) * \xhookrightarrow{\text{FR}} F' * \\ \exists allctr'.\, \mathscr{A}(allctr') \end{array} \right\}\end{aligned}$$

$$\boxed{\mathscr{K}[\![\tau s]\!]^{\tau_{\mathtt{lbs}},\tau_{\mathtt{ret}}}_{(\tau l,inst,hfs)} : Lholed \times \mathbf{N} \to iProp}$$

$$\begin{aligned}\mathscr{K}[\![\tau s]\!]^{\tau_{\mathtt{lbs}},\tau_{\mathtt{ret}}}_{(\tau l,inst,hfs)}(lh_n, j) \triangleq\ & \exists vs, k, es, lh'_m, es', lh''_{n-S(j)}. \\ & lh_n(n-S(j)) = (vs, k, es, lh', es') * \\ & \mathtt{outer}_{n-S(j)}(lh_n) = lh_{n-S(j)} * \\ & \square \forall v, F, allctr.\, \mathscr{V}[\![\tau s]\!](v) \mathrel{-\!\!*} \\ & \quad \mathscr{F}rame[\![\tau l]\!]_{inst}(F) \mathrel{-\!\!*} \mathscr{A}(allctr) \mathrel{-\!\!*} \\ & \quad \exists \tau s_2.\, \mathscr{E}[\![\tau s_2]\!]^{\mathtt{drop}\ S(j)\ \tau_{\mathtt{lbs}},\tau_{\mathtt{ret}}}_{\tau l,inst,hfs)} \\ & \quad \left(lh''_{n-S(j)}, vs +\!\!+ v +\!\!+ es +\!\!+ es'\right)\end{aligned}$$

Figure 3.22: Our logical relation for control flow (Part 1)

$$\boxed{\mathscr{H}[\![ts]\!]^{\tau_{\mathtt{lbs}},\tau_{\mathtt{ret}}}_{(\tau l,inst,hfs)} : Val \times Lholed \to iProp}$$

$$\begin{array}{l}
\mathscr{H}[\![ts]\!]^{\tau_{\mathtt{lbs}},\tau_{\mathtt{ret}}}_{(\tau l,inst,hfs)}(w,lh) \triangleq \\
\quad \exists llh, vs, ft, hidx, ts_1, ts_2. \\
\qquad w = \mathtt{call_hostV}\ ft\ hidx\ vs\ llh * \\
\qquad ft = ts_1 \to ts_2 * \\
\qquad (hidx, ft) \in hfs * \\
\qquad \mathtt{allBasicInstr}(llh) * \\
\qquad \mathscr{V}[\![ts_1]\!](vs) * \\
\qquad \square\ \forall vs', F, allctr.\, \mathscr{V}[\![ts_2]\!](vs') * \mathscr{F}rame[\![\tau l]\!]_{inst}(F) * \mathscr{A}(allctr) \mathrel{-\!\!*} \\
\qquad \mathsf{wp}\, llh[vs'] \left\{ w,\ \begin{array}{l} \left(\begin{array}{l} \mathscr{V}[\![\tau_{\mathtt{lbs}}]\!](w) \vee \\ \triangleright \mathscr{H}[\![ts]\!]^{\tau_{\mathtt{lbs}},\tau_{\mathtt{ret}}}_{(\tau l,inst,hfs)}(w,lh) \vee \\ \triangleright \mathscr{B}r[\![ts]\!]^{\tau_{ret}}_{(\tau l,inst,hfs)}(w,lh) \vee \\ \mathscr{R}et[\![\tau_{ret}]\!]_{(\tau l,inst)}(w) \end{array} \right) * \\ \exists F, allctr.\, \mathscr{F}rame[\![\tau l]\!]_{inst}(F) * \mathscr{A}(allctr) \end{array} \right\}
\end{array}$$

$$\boxed{\mathscr{B}r[\![[ts_1;\cdots;ts_l]]\!]^{\tau_{ret}}_{(\tau l,inst,hfs)} : Val \times Lholed \to iProp}$$

$$\begin{array}{l}
\mathscr{B}r[\![[ts_1;\cdots;ts_l]]\!]^{\tau_{ret}}_{(\tau l,inst,hfs)}(w,lh_n) \triangleq \\
\quad \exists lh'''_p, j.\, w = lh'''_p[\mathbf{br}\ \mathtt{j}] * \\
\quad \exists vs', vs, k, es, lh'_m, es', lh''_{n-S(j-p)}, ts'. \\
\quad lh_n(n - S(j-p)) = (vs, k, es, lh'_m, es') * \\
\quad \mathtt{outer}_{n-S(j-p)}(lh_n) = lh''_{n-S(j-p)} * \\
\quad lh'''_p(0) = (vs', -) * \\
\quad \mathscr{V}[\![ts' +\!\!+ ts_{j-p}]\!](vs') * \\
\quad \forall F, allctr.\, \mathscr{F}rame[\![\tau l]\!]_{inst}(F) * \mathscr{A}(allctr) \mathrel{-\!\!*} \\
\quad \mathsf{wp}\,(vs +\!\!+ \mathtt{label}_k\{es\}\ lh'_m\ \mathtt{end} +\!\!+ es')[\mathtt{drop}\ (\mathtt{len}(ts'))\ vs' +\!\!+ [\mathtt{br}\ (j-p)]] \\
\quad \left\{ w,\ \begin{array}{l} \left(\begin{array}{l} (\exists \tau s.\, \mathscr{V}[\![\tau s]\!](w)) \vee \\ \triangleright \left(\exists \tau s.\, \mathscr{H}[\![\tau s]\!]^{[ts_{S(S(j-p))},\cdots,ts_l],\tau_{ret}}_{(\tau l,inst,hfs)}(w) \right) \vee \\ \triangleright \mathscr{B}r[\![[ts_{S(S(j-p))},\cdots,ts_l]]\!]^{\tau_{ret}}_{(\tau l,inst,hfs)} \left(w, lh''_{n-S(j-p)} \right) \vee \\ \mathscr{R}et[\![\tau_{ret}]\!]_{(\tau l,inst)}(w) \end{array} \right) * \\ \exists F, allctr.\, \mathscr{F}rame[\![\tau l]\!]_{inst}(F) * \mathscr{A}(allctr) \end{array} \right\}
\end{array}$$

Figure 3.23: Our logical relation for control flow (Part 2)

$\boxed{\mathscr{C}los[\![ts \to ts']\!]_{hfs} : Closure \to iProp}$

$$\mathscr{C}los[\![ts \to ts']\!]_{hfs}\left(\{(inst, tlocs); e\}^{\texttt{NativeCl}}_{ts \to ts'}\right) \triangleq$$
$$\square \forall vs, f, allctr.\ \mathscr{V}[\![ts]\!](\texttt{immV}\ vs) * [\text{NaInv} : \top] * \xhookrightarrow{\text{FR}} F * \mathscr{A}(allctr) \mathrel{-\!\!*}$$
$$\mathscr{E}[\![ts']\!]_{(F,hfs)}(\textbf{local}_{\texttt{len}(ts')}\{inst; vs +\!\!+ \texttt{zeros}(tlocs)\}$$
$$\textbf{label}_{\texttt{len}(ts')}\{\varepsilon\}\ e\ \textbf{end end})$$

$$\mathscr{C}los[\![ts \to ts']\!]_{hfs}\left(\{h\}^{\texttt{HostCl}}_{ts \to ts'}\right) \triangleq (h, ts \to ts') \in hfs$$

$\boxed{\mathscr{F}unc[\![ts \to ts']\!]_{hfs} : \mathbb{N} \to iProp}$

$$\mathscr{F}unc[\![ts \to ts']\!]_{hfs}(n) \triangleq \exists cl.\ \boxed{n \overset{\texttt{wf}}{\longmapsto} cl}^{\mathscr{N}_{wf}.n}_{\text{NaInv}} * \triangleright\, \mathscr{C}los[\![ts \to ts']\!]_{hfs}(cl)$$

$\boxed{\mathscr{T}able[\![\{min; max\}]\!] : \mathbb{N} \to iProp}$

$$\mathscr{T}able[\![\{min; max\}]\!](t) \triangleq \exists n, m.\ t \xhookrightarrow{\texttt{tlim}} m * \ m \leq max * \ t \xhookrightarrow{\texttt{tlen}} n *$$
$$*_{i \in [0..n]}\ \exists \tau s, \eta s, fe.\ \boxed{t \overset{\texttt{wt}}{\longmapsto}_i fe}^{\mathscr{N}_{wt}.(t,i)}_{\text{NaInv}} *$$
$$\mathscr{F}unc[\![\tau s \to \eta s]\!](fe)$$

$\boxed{\mathscr{M}em[\![\{min; max\}]\!] : \mathbb{N} \to iProp}$

$$\mathscr{M}em[\![\{min; max\}]\!](n) \triangleq \exists mem.\ \boxed{\begin{array}{l} n \overset{\texttt{mlen}}{\longmapsto} \texttt{MemLen}(mem) \\ *_{i \mapsto b \in \texttt{MemData}(mem)}\ n \overset{\texttt{wm}}{\longmapsto}_i b \end{array}}^{\mathscr{N}_{wm}.n}_{\text{NaInv}} *$$
$$min \leq \texttt{MemLen}(mem) \wedge max = \texttt{MemMax}(mem)$$

$\boxed{\mathscr{G}lob[\![\{\mu; \tau\}]\!] : \mathbb{N} \to iProp}$

$$\mathscr{G}lob[\![\{\textbf{mut}; \tau\}]\!](n) \triangleq \boxed{\exists w.\ n \overset{\texttt{wg}}{\longmapsto} \{\textbf{mut}; w\} * \mathscr{V}_0[\![\tau]\!](w)}^{\mathscr{N}_{wg}.n}_{\text{NaInv}}$$
$$\mathscr{G}lob[\![\{\textbf{immut}; \tau\}]\!](n) \triangleq \exists (P : \texttt{valuetype} \to \texttt{value} \to iProp).$$
$$(\square \forall w, P(\tau)(w) \mathrel{-\!\!*} \mathscr{V}_0[\![\tau]\!](w)) *$$
$$\boxed{\exists w.\ n \overset{\texttt{wg}}{\longmapsto} \{\textbf{immut}; w\} * P[\![\tau]\!](w)}^{\mathscr{N}_{wg}.n}_{\text{NaInv}}$$

Figure 3.24: Our logical relation for instance elements

$\boxed{\mathscr{I}[\![C]\!]_{hfs} : Instance \to iProp}$

$$\mathscr{I}\left[\!\!\left[\left\{\begin{array}{l}\mathsf{types} = ts,\ \ \mathsf{func} = fts,\ \ \mathsf{global} = gts\\ \mathsf{table} = [tt,\cdots],\ \ \mathsf{memory} = [mt,\cdots]\\ \mathsf{locals},\mathsf{labels},\mathsf{return} = \cdots\end{array}\right\}\right]\!\!\right]_{hfs}\left(\left\{\begin{array}{l}\mathsf{types} = ts',\ \ \mathsf{funcs} = fs\\ \mathsf{globs} = gs,\ \ \mathsf{tabs} = [t,\cdots]\\ \mathsf{mems} = [m,\cdots]\end{array}\right\}\right) \triangleq$$

$$ts = ts' * \mathop{*}_{f\in fs; ft\in tfs} \mathscr{F}unc[\![ft]\!]_{hfs}(f) * \mathop{*}_{g\in gs; gt\in gts} \mathscr{G}lob[\![gt]\!](g) *$$
$$\mathscr{T}able[\![tt]\!](t) * \mathscr{M}em[\![mt]\!](m)$$

$\boxed{\mathscr{C}tx[\![C]\!]_{(inst,hfs)} : Lholed \to iProp}$

$$\mathscr{C}tx[\![\{\cdots;\tau l;\tau_{\mathtt{lbs}};\tau_{\mathtt{ret}}\}]\!]_{(inst,hfs)}(lh) \triangleq \mathtt{StructuralCond}(\tau_{\mathtt{lbs}}, lh) * \mathop{*}_{j\mapsto ts\in\tau_{\mathtt{lbs}}} \mathscr{K}[\![ts]\!]^{\tau_{\mathtt{lbs}},\tau_{\mathtt{ret}}}_{(\tau l,inst,hfs)}(lh,j)$$

Figure 3.25: Our logical relation for instances

$\boxed{C \vDash es\colon ts1 \to ts2}$

$$C \vDash es\colon ts1 \to ts2 \triangleq \forall inst, lh, hfs.\, (\mathscr{I}[\![C]\!](inst) * \mathscr{C}tx[\![C]\!]_{(inst,hfs)}(lh)) \mathrel{-\!\!*} \forall F, allctr, vs.\, \Big(\mathscr{V}[\![ts1]\!](vs) * \xhookrightarrow{\text{FR}} F * \mathscr{F}rame[\![\tau l]\!]_{inst}(F) * \mathscr{A}(allctr)\Big) \mathrel{-\!\!*} \mathscr{E}[\![ts2]\!]^{\tau_{\mathrm{lbs}},\tau_{\mathrm{ret}}}_{(\tau l,inst,hfs)}(lh, vs +\!\!+ es)$$

where $\tau l = C.\mathsf{locals}$, $\tau_{\mathrm{lbs}} = C.\mathsf{labels}$ and $\tau_{\mathrm{ret}} = C.\mathsf{return}$

Figure 3.26: Our definition of semantic typing

3.D Fundamental Theorem

As stated in the paper, if a program *bs* (a list of *basic instructions*, i.e. only using instructions available to the programmer) typechecks syntactically, then it typechecks semantically:

$$\forall bs, C, ts1, ts2. C \vdash bs : ts1 \to ts2 \implies C \vDash bs : ts1 \to ts2$$

Chapter 4

Iris-WasmFX: Modular Reasoning for Wasm Stack Switching

This chapter consists of a manuscript that we hope to submit for PLDI 2026:

Maxime Legoupil, Jean Pichon-Pharabod, Sam Lindley, Lars Birkedal
Iris-WasmFX: Modular Reasoning for Wasm Stack Switching
To be submitted to PLDI 2026

Before submitting this manuscript to PLDI, we plan on making the following additions:

- The progam logic will encompass the entirety of stack switching, not just the core new instructions as is currently the case
- New examples will be added
- The paper will be reformatted to fit the ACM standard layout. During this process, some parts of the paper may be moved to an appendix

Abstract

The stack switching proposal extends WebAssembly with primitives for explicit manipulations of the execution stack as continuations. By exposing an interface in the style of effect handlers, stack switching makes it easy to efficiently compile non-local control flow primitives in a modular way: one handcrafts a library that efficiently implements the desired source language primitives, and compilation then merely calls this effect-based library. However, code involving non-local control flow is infamously 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 formalisation of the stack switching proposal that we extend to include runtime terms, 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 demonstrate it on a key use case of effect handlers: a coroutine library. Together, these validate the design of stack switching, and make it possible to verify future effect-based libraries.

4.1 Introduction

WebAssembly (abbreviated Wasm) [41, 94, 98] 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, such as promises, async/await, generators, and others. For now, such a compilation needs to address the fact that WebAssembly 1.0 only features local control flow. This restriction means that, with the state of the art, such a compilation is a whole-program transformation. This is currently what is used in production, for example in Binaryen's Asyncify [142]. However, this transformation suffers from three problems. First, it incurs a significant overhead, in particular in code size. Second, it involves a complex, whole-program transformation. And third, it is not compositional: transformed code can only be composed with transformed code, not with off-the-shelf WebAssembly code.

To enable more efficient, simpler, and compositional compilation to WebAssembly, the stack switching proposal [66] extends WebAssembly with non-local control flow primitives in the form of effect handlers [86, 87]. The name of this proposal refers to the fact that it allows a WebAssembly program to manage multiple execution stacks as continuations, and to switch between them. Stack switching follows ideas from WasmFX [66, 83], typed continuations [45, 96, 102], and several other projects. Stack switching already has implementations and industrial uses, but two important aspects remain to be investigated. First, although the static aspects of WebAssembly are described in the proposal, the operational semantics of stack switching, in particular to do with the details of the runtime state of programs, has not yet been fully formally validated. Second, the expected use case of stack switching is to combine compiled code with hand-optimised libraries efficiently implementing the non-local control flow primitives of the source languages, raising the question of how to ascertain correctness of these libraries.

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

First, to capture the semantic details of stack switching, we introduce WasmFX-Cert, a mechanisation of stack switching in the Rocq proof assistant, build atop the existing WasmCert formalisation of plain WebAssembly. WasmFXCert defines the full operational semantics and syntactic type system of stack switching, including formally defined typing of runtime 'administrative' instructions, which existing specifications did not cover beyond an informal description [83, §3.4]. WasmFXCert also, on top of that definition, provides the very first result on the type safety of stack

switching, in the form of a mechanised proof by progress and preservation — which needs to refer to those runtime 'administrative' instructions — thus validating the basic consistency of the stack switching proposal.

Second, because effect handlers introduce non-local control flow (as intended), they famously hinder modular reasoning at the level of the operational semantics [23, 63]. In particular, the capturing and storing of continuations make it challenging [125] to reason about a program module-by-module, as it 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, a program logic for stack switching defined in the Iris separation logic framework by extending the existing Iris-Wasm program logic [89] for plain WebAssembly. Using Iris-WasmFX, we show how to reason modularly about the behaviour of stack switching programs instantiated by a simple host language that captures the main elements of the Javascript WebAssembly interface.

To reason about effect handlers in particular, we take inspiration from Hazel [18–20], an Iris-based program logic designed to reason about effect handlers in a toy ML-like language, which we adapt to WebAssembly and stack switching.

In this paper, we outline some of the more technical 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 to gain a fine understanding of the behaviour of stack switching programs and have confidence in desired properties of given programs. Libraries whose behaviour is critical for safety can be verified manually, and users of stack switching can have strong confidence in the soundness of its design.

Together, WasmFXCert and Iris-WasmFX validate the design of stack switching, and illustrate how to achieve modular reasoning for the kind of key libraries that one would implement using stack switching.

Plan We first present our running example of a typical use case of stack switching, which we use throughout the paper to demonstrate how to program with stack switching, and how to specify and verify in Iris-WasmFX (§4.2). We then give an introduction to stack switching and describe WasmFXCert, our formalisation of it, by illustrating it on this example (§4.3), and show how we handle typing of running programs to establish type safety. With this foundation in place, we illustrate how to verify stack switching programs using Iris-WasmFX (§4.4). We then return to our motivating example, and show how to use Iris-WasmFX to verify it (§4.5). We finish with related work (§4.6) and limitations (§4.7), and conclude (§4.8).

4.2 A Canonical Use Case of Stack Switching: a Coroutine Library

In the remainder of this paper, we illustrate the design of stack switching, our mechanisation WasmFXCert, and our program logic Iris-WasmFX, on a canonical use case

of stack switching: a small cooperative coroutine library module, together with a client. 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 stack switching makes possible. Without stack switching, this kind of functionality would require a somewhat complex compilation of the client code, incurring some code size inflation. Instead, with stack switching, a concise definition can be given to $par and $yield, as we show below. Thanks to WebAssembly's modularity, from the point of view of a client module, the functions $par and $yield can be called like normal functions, and the code of the client can thus be written without usage of any of the new instructions in the stack switching semantics.

A Simple Specification of Coroutines A natural specification for this coroutine module in the style of Concurrent Separation Logic (CSL) [11] is centered around the notion of an invariant: if both client functions $\$f_i$ (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. (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[1] [52, §2.2]. We present the specification in CSL style for clarity.) Assuming this, if each function $\$f_i$ starting from precondition P_i, ensures postcondition Q_i at termination, with access to invariant I at the start and reliquinshing 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 slightly more precisely intuition of the shape of the specification by giving Hoare triples for the $yield and $par functions. The Hoare triple for $par uses nested Hoare triples [105]: it requires in its precondition corresponding specifications for $\$f_1$ and $\$f_2$ in the form of Hoare triples:

$$\begin{array}{l}\forall P_1, P_2, Q_1, Q_2, I.\\ \{I\}[\texttt{invoke}\ \mathit{addryield}]\{I\} *\\ \left\{\begin{array}{l} P_1 * P_2 * I * \\ \{P_1 * I\}[\texttt{invoke}\ \mathit{addrf}_1]\{Q_1 * I\} * \\ \{P_2 * I\}[\texttt{invoke}\ \mathit{addrf}_2]\{Q_2 * I\} \end{array}\right\} \left[\begin{array}{l}\texttt{func.ref}\ \$f_1;\\ \texttt{func.ref}\ \$f_2;\\ \texttt{invoke}\ \mathit{addrpar}\end{array}\right] \{Q_1 * Q_2 * I\}\end{array}$$

where *addryield*, *addrpar*, addrf_1 and addrf_2 are the addresses of the closures for $yield, $par, $\$f_1$ and $\$f_2$. We refine this specification in Section 4.5 (see Fig. 4.11) to account for potential surrounding uses of stack switching.

[1] Throughout this paper, what we call the *invariant* I is an invariant in the usual sense, *not* an Iris invariant

Modularity Importantly, this specification can be proved for the coroutine module by itself, without access to the code of the client functions $\$f_i$, 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 stack switching: $yield internally `suspend`s the current program, saving the current continuation, and $par internally `resume`s the continuations of $\$f_1$ and $\$f_2$, round-robin, using the corresponding stack switching instructions. We describe such a simple implementation of this coroutine module in Section 4.3, and give its code in Fig. 4.4.

Library Client A simple example use of this toy coroutine library is a client implementing a classic 'message passing' concurrency example [109, 8–1][64, §3]. We implement this in WebAssembly in Fig. 4.1, in the form of a client module that uses the coroutine module to concurrently run a 'writer' function $\$f_1$ 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, and a 'reader' function $\$f_2$ that reads the 'flag' $y and then reads the 'data' in $x, with both functions $yield`ing` in between their two accesses. 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 remainder of this paper, we build up to making the above specification precise, and we prove it in Section 4.5.

4.3 WasmFXCert: Formal Operational Semantics for Stack Switching

In this section, we introduce stack switching and our mechanisation of it in the Rocq proof assistant, WasmFXCert. Phipps-Costin et al. [83, §2–3] give a more detailed introduction of WasmFX.

4.3.1 Stack Switching, Informally

Design of Stack Switching Stack switching 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 is halted, and control is handed over to the innermost *handler* for that effect, thereby defining a delimited continuation [27]. If it was intended for programmers to routinely write code directly in WebAssembly, it would make most sense to define a specific, user-friendly mechanism. But since WebAssembly 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 WebAssembly's effect handlers in stack switching is somewhat unusual, and we illustrate it on our motivating example. Phipps-Costin et al. [83, §4] show how to implement other types of non-local control flow using the instructions of WasmFX.

```
;; 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 value of $y
  (global $b (mut i32) (i32.const 0))        ;; reader's copy of 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
```

Figure 4.1: 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.

```
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 loop, circumventing code for yield below
7   ...                      ;; handle yield, changing function to run
8   br $l)                   ;; resume the loop
```

Figure 4.2: High-level structure of the implementation of $par (full code in Figure 4.4)

Stack Switching Instructions The core of stack switching revolves around two new instructions: `suspend`, which stops the current execution context, and reifies it as a *continuation*, a new kind of WebAssembly value with a corresponding new WebAssembly type; and `resume`, which starts executing such a continuation — thereby switching the stack, hence the name of the WebAssembly 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 one function, then the other, in a loop. We show the general structure of this loop in Fig. 4.2. We now focus on how to understand the `resume` instruction on line 4.

Block-Based Effect Handling The main unusual feature of stack switching is that `suspend` suspends to a code `block` (despite the word 'assembly' its name, WebAssembly only has *structured* control flow and no arbitrary `goto` instruction) specified by the `resume`, as opposed to some more user-friendly mechanism. As such, stack switching continuations are *delimited* continuations [27]. 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 WebAssembly, `loop` only declares a `block` to loop to: actual looping only takes place given an explicit `br` $l (line 6) to the label $l of the `loop`— otherwise, execution just falls through out of the `loop`; the handling of $yield also needs to explicitly loop with a `br` (line 8).

This fine-grained setup dissociates the catching of the effect from its handling, while they are usually conflated in higher-level languages (`try` ... `with` ..., `handle` ... `with` ..., etc.), making it possible to implement different mechanisms for non-local control flow.

Tags To compose different effects modularly, `suspend` is annotated with a `tag`, which is usually called an *effect* in the effect handler literature. Each `tag` specifies (via its tag type) 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: `resume` (`on` $t $b).

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.

4.3.2 Operational Semantics

Pedigree The stack switching proposal builds upon WebAssembly 2.0 extended with the exception handling proposal[2] [4]. However, it only makes use of a few features not present in WebAssembly 1.0. Our mechanisation, WasmFXCert, takes advantage of this, and is built on top of WasmCert 1.0 [133], to which we have manually added the features of WebAssembly 2.0 and of the exception handling proposal that stack switching uses. This is because Iris-Wasm [89], which we used as a departure point to define our program logic (see §4.4), is defined for WebAssembly 1.0; defining WasmFXCert atop WebAssembly 2.0 (using WasmCert 2.0) would have necessitated defining proof rules in the program logic for the numerous features of WebAssembly 2.0 that are orthogonal to stack switching.

In the rest of this section, we give a succinct introduction to the semantics of WebAssembly. A reader familiar with WebAssembly can safely skip to §4.3.3, where we describe the new reduction rules for stack switching. A reader familiar with stack switching can skip to §4.3.4, where we describe the type system for stack switching, including our additions and our theorem for type safety.

Syntax We give the syntax[3] for stack switching in Figures 4.3, 4.5 and 4.6, and illustrate how a WebAssembly module is built and how WebAssembly code runs using the implementation of our coroutine module in Fig. 4.4. The coroutine module in Fig. 4.4 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 WebAssembly program is separated into two entities: the *frame* and the *store*. The store S contains all objects (e.g. global variables, functions, memories, as well as, in stack switching, 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 WebAssembly's local state encapsulation property, but increases the complexity of defining a mechanisation and program logic. As we show

[2]The exception handling constructs are only necessary to define the `resume_throw` instruction, which is mere syntactic sugar not crucial for understanding stack switching; thus we elide details about exception handling outside of our figures.

[3]To avoid confusion with the symbol $*$ for the separating conjunction, we depart from the WebAssembly standard and use an '*s*' suffix to indicate a list; e.g. *bs* indicates a list of *b* elements.

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

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

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

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

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

$$
\begin{array}{rcl}
\text{(import descriptions) } importdesc & ::= & \texttt{func}_{\texttt{i}}\ i \mid \texttt{tab}_{\texttt{i}}\ min\ max \mid \texttt{mem}_{\texttt{i}}\ min\ max \mid \\
 & & \texttt{glob}_{\texttt{i}}\ \texttt{mutable}^{?}\ t \mid \texttt{tag}_{\texttt{i}}\ i \\
\text{(imports) } import & ::= & \texttt{import } string\ string\ importdesc \\
\text{(export descriptions) } exportdesc & ::= & \texttt{func}_{\texttt{e}}\ i \mid \texttt{tab}_{\texttt{e}}\ i \mid \texttt{mem}_{\texttt{e}}\ i \mid \texttt{glob}_{\texttt{e}}\ i \mid \texttt{tag}_{\texttt{e}}\ i \\
\text{(exports) } export & ::= & \texttt{export } string\ exportdesc \\
\text{(start) } start & ::= & \texttt{Some } i \mid \texttt{None}
\end{array}
$$

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

Figure 4.3: The AST of the WasmFXCert language. Black parts are in WasmCert 1.0 (plain WebAssembly 1.0). Parts in indigo are the instructions (those mentioning references) from Wasm 2.0 that are needed for stack switching. Parts in orange are the additions of the exception handling variant of WebAssembly, that are also needed for stack switching. Parts in magenta belong to stack switching proper. Places where WasmFXCert departs from stack switching are shown 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 (effect) $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
```

Figure 4.4: Implementation of our simple coroutine module using stack switching

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

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

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

(translated exception clause) dec ::= $\texttt{catch}\ addr\ i \mid \texttt{catch\_ref}\ addr\ i \mid \texttt{catch\_all}\ i \mid \texttt{catch\_all\_ref}\ i$
(translated continuation clause) dcc ::= $\texttt{on}\ addr\ i \mid \texttt{on}\ addr\ \texttt{switch}$

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

Figure 4.5: The AST for structures necessary to evaluate code in WasmFXCert (part 1). Black parts are borrowed directly from WasmCert (plain WebAssembly 1.0). Parts in indigo pertain to the references from WebAssembly 2.0, parts in orange pertain to exception handling, and parts in magenta are new in stack switching. Places where WasmFXCert departs from stack switching are shown boxed.

in §4.4, Iris-WasmFX treats the frame differently to what is done in Iris-Wasm [89], which simplifies the logic a little.

Reductions A reduction in the WebAssembly operational semantics is thus of the form $(S, F, es) \hookrightarrow (S', F', es')$, where S and F are the aforementioned store and frame, and es is a hybrid stack of values and instructions. Specifically, es is a list of *administrative instructions*, a superset of basic instructions that includes instructions used to represent a program mid-execution; we describe these in the rest of this subsection.

Control Flow To express the semantics of `loop` and `block`, WebAssembly 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

$$
\begin{array}{rcl}
\text{(evaluation context) } L & ::= & \boxed{vs \mathbin{+\!\!+} [\_] \mathbin{+\!\!+} es} \mid \\
& & vs \mathbin{+\!\!+} \texttt{label}_n\{es'\}\ L\ \texttt{end} \mathbin{+\!\!+} es \mid \\
& & vs \mathbin{+\!\!+} \texttt{handler}\{decs\}\ L\ \texttt{end} \mathbin{+\!\!+} es \mid \\
& & vs \mathbin{+\!\!+} \texttt{prompt}_{ts}\{dccs\}\ L\ \texttt{end} \mathbin{+\!\!+} es \\
\text{(handler context) } H & ::= & vs \mathbin{+\!\!+} [\_] \mathbin{+\!\!+} es \mid vs \mathbin{+\!\!+} \texttt{label}_n\{es'\}\ H\ \texttt{end} \mathbin{+\!\!+} es \mid \\
& & vs \mathbin{+\!\!+} \texttt{frame}_n\{F\}\ H\ \texttt{end} \mathbin{+\!\!+} es \mid \\
& & vs \mathbin{+\!\!+} \texttt{handler}\{decs\}\ H\ \texttt{end} \mathbin{+\!\!+} es \mid \\
& & vs \mathbin{+\!\!+} \texttt{prompt}_{ts}\{dccs\}\ H\ \texttt{end} \mathbin{+\!\!+} es
\end{array}
$$

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

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

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

Figure 4.6: The AST for structures necessary to evaluate code in WasmFXCert (part 2). Black parts are borrowed directly from WasmCert (plain WebAssembly 1.0). Parts in orange pertain to exception handling, and parts in magenta are new in stack switching. Places where WasmFXCert departs from stack switching are shown boxed.

nothing in case of a `block`), and n is the number of values that should be on the stack in case of a break.

WebAssembly defines evaluation by the composition of head reduction rules with the following evaluation-under-context rule:

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

where evaluation contexts[4] L are as in Fig. 4.6. 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

[4]These are also sometimes noted lh_i in other works (e.g. in Iris-Wasm [89]) and is noted B^i in the standard [98].

can run. If that code terminates on a value, WebAssembly defines a reduction rule that says 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. Aside from an explicit `br` instruction, a few other instructions reduce to `br`, like `br_if` (see lines 26 and 34) and `br_table`. In stack switching, as we will detail in §4.3.3, the `resume` instruction also reduces to a `br` $b when an effect is performed, meaning the code after the instruction corresponds to the code that will be run if no effect is performed, and the code es_{cont} of the `label` instruction labelled $b will run if an effect is performed.

Function Calls WebAssembly 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 closure for $f, and `call` $f reduces to `invoke` *addr*. A module can also define a function table and use the `call_indirect` instruction to call functions from it. In WebAssembly 2.0, the `ref.function` $f instruction creates a reference to an existing function, which can be called using the same `call_indirect` instruction in the official specification. This requires modifying the exact semantics of the `call_indirect` instruction. In order to avoid making that modification (which would later require adapting the program logic rule for this instruction), WasmFXCert defines a separate instruction `ref.call` for calling references. In WasmFXCert, the `call_indirect` instruction is thus reserved for the function table just like in WebAssembly 1.0.

Closures The closures in the store can either be defined by the embedding host language (see §4.3.5, 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[5]; or it can be defined in WebAssembly, in which case `invoke` *addr* reduces to $(\texttt{frame}_{|ts_2|}\{F'\}\ \texttt{block}\ ([] \to ts_2)\ bs\ \texttt{end})$ 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[6] represents the execution of

[5]This departure from the official WebAssembly standard was introduced by Rao et al. [89, §2.2] when defining Iris-Wasm, to ease the mechanisation of the integration of WebAssembly into a host language.

[6]This used to be called `local` in older versions of the WebAssembly standard, and correspondingly in the Iris-Wasm program logic that we build on. The name was changed in the standard to avoid confusion with other meanings of the word 'local', for example for local variables, but the new name 'frame' also clashes with WebAssembly function frames and separation logic frames (as well as *frame stacks* [84, §3], a term which we do not use anywhere in this paper but is sometimes used when describing WebAssembly evaluation contexts)

a function call, and is responsible for ensuring encapsulation. The rule for running the code inside of a `frame` instruction is similar to REDUCE-LABEL, but updates the frame in its premise:

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

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.

4.3.3 New Instructions in Stack Switching

We can now introduce formally the new stack switching instructions sketched in §4.3.1. Stack switching defines a new type, `ref.cont` *ft*, for continuations of type *ft*. A new continuation can be created from a function reference using the `cont.new` instruction. The new continuation is placed in the store at an address *addr*, and an object `ref.cont` *addr* is returned:

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

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 stack switching, 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`, stack switching defines an administrative instruction `prompt` to represent a continuation being run. Just like the `resume` instruction, the `prompt` instruction 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, and the clauses in the `prompt` instruction has the explicit store addresses of these tags.

To account for the fact that in stack switching, continuations are one-shot [83, §2.4], upon resuming, the store is updated with a `dagger` token to indicate the continuation has already been resumed, and that therefore calling it again will cause a `trap`:

REDUCE-RESUME

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

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 use notation $F \vdash ccs \to dccs$ to describe this translation.

Stack switching augments the definition of the execution context 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, stack switching 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 the reduction rule 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 documentation for the stack switching proposal attempted to do this translation implicitly but ended up using the wrong function instance to do so. This bug[7] in the proposal was exposed by the mechanisation and has now been fixed.

Our proposed solution in WasmFXCert is to make this translation from tag names to tag addresses explicit: we define 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 stack switching 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 stack switching 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

[7]The official reference interpreter for the stack switching proposal implements a smaller-step semantics for the `suspend` instruction and avoids the issue, hence the bug is only present in the *documentation* and not in the reference implementation

maintain consistency with other places in the standard and limit the proliferation of administrative instuctions. 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 Fig. 4.10): 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 §4.4.2 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:

REDUCE-SUSPEND-TRANSLATE

$$\frac{F.\mathsf{inst.tags}[i] = addr \qquad S.\mathsf{tags}[addr] = ts_1 \to ts_2 \qquad |vs| = |ts_1|}{(S, F, vs +\!\!+ [\texttt{suspend}\ i]) \hookrightarrow (S, F, [\texttt{suspend.addr}\ vs\ addr])}$$

The Rule for Suspension The `suspend.addr` instruction itself is stuck and needs to be placed under an appropriate context to reduce. When reducing $H[\texttt{suspend.addr}\ vs\ addr]$, 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 $H_1[\texttt{prompt}_{ts}\{dccs\}\ H_2[\texttt{suspend.addr}\ vs\ addr]\ \texttt{end}]$ 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 $H^{\texttt{sus}}_{addr}[es]$ to denote capture-avoiding plugging (i.e. no prompts with a suspend clause handling $addr$ in H) of es into context H.

REDUCE-SUSPEND

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

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 use notation $H^{\texttt{sw}}_{addr}[es]$ 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

$$\frac{\begin{array}{c} F.\mathsf{inst.types}[i] = ft \\ ft = ts_1 \to ts_2 \qquad F.\mathsf{inst.tags}[i'] = addr' \qquad |vs| + 1 = |ts_1| \end{array}}{(S, F, vs +\!\!+ [\texttt{ref.cont}\ addr;\ \texttt{switch}\ i\ i']) \hookrightarrow (S, F, [\texttt{switch.addr}\ vs\ addr\ ft\ addr'])}$$

REDUCE-SWITCH

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

Behaviour of `cont.bind` and `resume_throw` Finally, stack switching 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 stack switching, so we omit further mention of them. They are modelled in our mechanisation WasmFXCert, but not yet in our program logic Iris-WasmFX. We plan on adding support for these instructions in our program logic before this manuscript is submitted for publication.

Departures from the Stack Switching Proposal in WasmFXCert WasmFXCert makes a few minor departures from the official stack switching semantics. These are shown boxed in Figures 4.3, 4.5 and 4.6:

- The exact shape of the grammar for reference types and evaluation contexts varies from version to version of the WebAssembly standard and its various extensions. All of these presentations are equivalent and ours is designed to be as readable as possible given what features are present in the rest of the grammar.
- As discussed, the distinction between `call_indirect` and `ref.call` is implicit in the standard but made explicit in WasmFXCert.
- As discussed, we place the payload of an effect in the translated `suspend.addr` and `switch.addr`.
- We add a type annotation on the `prompt` instruction and on the continuations in the store. Those type annotations are only on runtime terms, not source terms, 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.

$\boxed{C \vdash cc \texttt{ ok with type } ts}$

$$\frac{C.\mathsf{tags}[i] = ts_1 \to ts_2 \qquad C.\mathsf{label}[i'] = ts_1 +\!\!+ \ (ts_2 \to ts).\texttt{cont}}{C \vdash \texttt{on } i\ i' \texttt{ ok with type } ts}$$

$$\frac{C.\mathsf{tags}[i] = [] \to ts}{C \vdash \texttt{on } i \texttt{ switch} : ts}$$

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

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

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

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

Figure 4.7: The typing rules for stack switching

4.3.4 Syntactic Typing and Type Safety

In this section, we introduce stack switching's syntactic type system. When defining WasmFXCert, we give new typing rules for runtime terms, and we state and prove a type safety theorem.

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

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

where C is a typing context (see Fig. 4.6) 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 typesof the values deposited back on the stack after execution. For example, for all contexts C and all numerical types nt, $C \vdash nt.\textbf{add} : [nt; nt] \to [nt]$.

We show the typing rules for the new instructions of stack switching in Fig. 4.7.

When defining WasmFXCert, we have included the first result on type safety of stack switching, 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 WebAssembly store. While Phipps-Costin et al. [83] 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 Fig. 4.8.

Two main challenges arise when defining these typing rules. The first is that, unlike in plain WebAssembly 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 values keep on being well-typed. 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'.

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 $[\texttt{frame}_n\{F\}\ H\ \texttt{end}]$, and the body of a `frame` instruction is typed with a specifically built typing context in the WebAssembly 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. To avoid modifying the operational semantics, our solution is to mandate that the body of a `prompt` should typecheck in the empty typing context, as illustrated in Fig. 4.8. This necessitated replacing all tag and type annotations with their explicit values (as is done implicitly in stack switching during the typing phase), and proving lemmas about typing expressions in the empty typing context, such as the following:

Lemma 3. *For all stores S, typing contexts C, administrative instructions es, and function types ft,*

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

Type Safety Theorem As is standard in WebAssembly'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 4 (Type Preservation)**.** *If* $\vdash S\ \texttt{ok}$ *and* $S,C(S,F) \vdash es : [] \to ts$ *and* $(S,F,es) \hookrightarrow (S',F',es')$*, then* $\vdash S'\ \texttt{ok}$ *and* $S',C(S',F') \vdash es' : [] \to ts$

Lemma 5 (Type Progress)**.** *If* $\vdash S\ \texttt{ok}$ *and* $S,C(S,F) \vdash es : [] \to ts$*, 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.*

$\boxed{S,C \vdash dcc \texttt{ ok with type } ts}$

$$\frac{S.\mathsf{tags}[addr] = ts_1 \to ts_2 \qquad C.\mathsf{label}[i] = ts_1 +\!\!+ \ (ts_2 \to ts).\texttt{cont}}{S,C \vdash \texttt{on } addr\ i\ \texttt{ok with type } ts}$$

$$\frac{S.\mathsf{tags}[addr] = [] \to ts}{S,C \vdash \texttt{on } addr\ \texttt{switch ok with type } ts}$$

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

$$\frac{S.\mathsf{conts}[addr] = (\texttt{cont } ft\ \_ \vee \texttt{dagger } ft)}{S,C \vdash \texttt{ref.cont } addr : [] \to ft.\texttt{cont}}$$

$$\frac{S.\mathsf{tags}[addr] = ts_1 \to ts_2 \qquad S,C \vdash vs : [] \to ts_1}{S,C \vdash \texttt{suspend.addr } vs\ addr : [] \to ts_2}$$

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

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

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

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

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

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

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

$$\text{Premises for } \begin{array}{c} \text{functions} \\ \text{tables} \\ \text{memories} \\ \text{globals} \end{array} \text{ as in WasmCert}$$

$$\frac{\forall c \in S.\mathsf{conts},\ S \vdash c\ \texttt{ok} \qquad \forall e \in S.\mathsf{exns},\ S \vdash e\ \texttt{ok}}{\vdash S\ \texttt{ok}}$$

Figure 4.8: Our new typing rules for the administrative instructions and store elements of stack switching

And together, these give the type safety result:

Theorem 9 (Type Safety). *If* $\vdash S$ ok *and* $S, C(S,F) \vdash es : [] \to ts$ *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.*

4.3.5 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 WebAssembly itself but instead is done by an embedding *host language*. The Iris-Wasm program logic [89] comes with a custom-designed host language that can instantiate WebAssembly modules and do a few other WebAssembly operations like calling WebAssembly functions or reading WebAssembly global variables. For example, to run our example from §4.2, the host program (shown in Theorem 12) consists of three steps: instantiating the coroutine module, instantiating the client module, and calling the $main function of the client module (see line 2 in Fig. 4.1 for how to run these three steps in the reference interpreter of stack switching). This host language is mechanised in Rocq and has its own simple program logic, which we reuse in our program logic for stack switching.

The host language can also be a source of effectfulness (see the discussion on the Javascript promise interface in §4.6). In the rest of this paper, we consider a simpler host language that does not introduce its own effects.

4.4 Iris-WasmFX: Modular Reasoning for Stack Switching Programs

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

4.4.1 Iris and Iris-Wasm

Our program logic is defined in Iris [52], a framework for higher-order separation logic which has already been used to reason about WebAssembly with Iris-Wasm [89] and Memory-Safe WebAssembly (MSWasm) with Iris-MSWasm [59], 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 most important construct in our logic, used to give the specifications for functions, is our custom-defined *extended weakest precondition*

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

We define it by adding WebAssembly-specific components to Hazel's extended weakest precondition [18–20] (see §4.4.2), which is itself built on the (plain) *weakest precondition* statement from the Iris logic:

$$\mathsf{wp}\, es\, \{w, \Phi(w)\}$$

This (plain) 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, verification results are sometimes displayed under the form of a *Hoare triple*: $\{P\}es\{w, \Phi(w)\}$ 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. The Hoare triple can be derived from a weakest precondition:

$$\{P\}es\{w, \Phi(w)\} = \Box(P \mathrel{-\!\!*} \mathsf{wp}\, es\, \{w, \Phi(w)\})$$

where $-\!\!*$ is a separating implication and the persistent modality $\Box$ means that the implication inside the Hoare triple can be used as many times as necessary; as a counterpart, it 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 (a conjunction[8] of) *resources*, each resource representing *ownership* of a piece of the state. In Iris-Wasm [89], these represent individual pieces of the WebAssembly store, like function closures ($addr \overset{\texttt{wf}}{\mapsto} cl$), global variables ($addr \overset{\texttt{wg}}{\mapsto} gval$), etc. The annotation on the arrow differentiates the various kinds of resources.

To accommodate for stack switching, our program logic Iris-WasmFX defines two new resources: one for tags, and one for continuations.

Since tags cannot be redefined and knowledge of a tag is useless on its own (one needs a continuation to perform any action), we define a persistent resource $addr \overset{\texttt{wtag}}{\mapsto} \Box\, ft$ for tags, representing *knowledge* rather than exclusive ownership. Technically, this is done by discarding the ownership fraction; a reader unfamiliar with Iris need only note that this means the resource $addr \overset{\texttt{wtag}}{\mapsto} \Box\, ft$ can be duplicated as many times as needed. We add a square $\Box$ in the notation to symbolise this.

The second resource we introduce represents exclusive ownership of a *safe* continuation and its type: $addr \overset{\texttt{wcont}}{\mapsto} (ft, scont)$. We define a safe continuation as being either a dead continuation (recall stack switching continuations are one-shot), or a *safe context*, which is either the context $[\_] +\!\!+ [\texttt{ref.func \$f; ref.call}\, ft]$ for some $\texttt{\$f}$ and ft, or a context of the form $[\texttt{frame}_n\{F\}\ H\ \texttt{end}]$ (with H a handler context as defined in Fig. 4.6). 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 agnostically 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

[8] In separation logic, the conjunction of P and Q is noted $P * Q$

`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 the `resume` instruction below.

Proof Rules The Iris-Wasm program logic [89] for plain WebAssembly defines proof rules for all WebAssembly instructions using the plain weakest precondition statement. Before we introduce our new extended weakest precondition, let us describe some of these proof rules; we inherit them almost as-is in Iris-WasmFX. For example, the Iris-Wasm rule for updating a global variable is:

$$\text{WP-GLOBAL-SET}\quad \frac{\hookrightarrow^{\text{FR}} F * F.\mathsf{inst.globs}[\$g] = addr * addr \mapsto^{\texttt{wg}} v_1}{\mathsf{wp}\,[v_2; \texttt{global.set}\ \$g]\left\{w, w = \texttt{immV}\ [] * \hookrightarrow^{\text{FR}} F * addr \mapsto^{\texttt{wg}} v_2\right\}}$$

In the following paragraphs, we explain the meaning of `immV` and $\hookrightarrow^{\text{FR}}$.

Logical Values In order to reason more modularly, Iris-Wasm defines several kinds of *logical values*, all representing expressions that do not reduce:

- Immediate values `immV` *vs* represent lists of WebAssembly values.
- Trap values `trapV` represent a `trap` failure value.
- Breaking values `brV` *i vh* represent `br` *i* expressions in a context *vh* syntactically enforced[9] to be too shallow to break out from.
- Return values `retV` *sh* represent `return` expression in a context *sh* that does not contain a `frame` instruction.
- Host call values `callhostV` *vs k tf llh* represent calls to host closures.

The reason that Iris-Wasm defines logical values more broadly than just WebAssembly values is that it makes reasoning modular by making it possible to have a standard *bind rule* [52, 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 reasoning about the result w of *es* in context L. In Iris-Wasm, this rule looks like this:

$$\text{WP-BIND-LABEL}\quad \frac{\mathsf{wp}\, es\,\{w, \mathsf{wp}\, L[w]\,\{v, \Phi(v)\}\}}{\mathsf{wp}\, L[es]\,\{v, \Phi(v)\}}$$

This rule mirrors reduction rule REDUCE-LABEL, and in particular makes it possible to focus reasoning on a specific instruction so that one can, say, apply the

[9] This is done by giving a dependent type to *vh*, which is orthogonal to Iris-WasmFX so we omit the exact definition

proof rule specific to that instruction. Thanks to `brV` i vh being a value, one can even focus on code that will get stuck: once it is reduced to the value $w = \texttt{brV}\ i\ vh$, the postcondition must hold of w, which means there remains to prove a weakest precondition on $L[vh[\texttt{br}\ i]]$. 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 $\texttt{brV}\ i\ (L \circ vh)$, and one must now prove that Φ holds of that value.

The above rule remains true in Iris-WasmFX *provided one limits* the evaluation context L to only contain `label` instructions and no `prompt` instructions, since non-local control flow famously breaks the bind rule [125]. Instead, we show later a different, slightly more intricate rule that allows to soundly bind into a `prompt` instruction.

Frame Another novelty we bring in Iris-WasmFX with respect to the existing Iris-Wasm program logic is the way we deal with the WebAssembly 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,[10] 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 $\texttt{wp}\, es; F\, \{w\ F, \Phi(w, F)\}$ 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:

$$\frac{\text{WP-BIND-FRAME}\quad \texttt{wp}\, es; F\, \{w\ F', \texttt{wp}\, [\texttt{frame}_n\{F'\}\ w\ \texttt{end}]; F_0\, \{v\ F_1, \Phi(v, F_1)\}\}}{\texttt{wp}\, [\texttt{frame}_n\{F\}\ es\ \texttt{end}]; F_0\, \{v\ F_1, \Phi(v, F_1)\}}$$

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

[10] 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.

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 Fig. 4.4. Recall from §4.2 we want this specification to have the form $\{I\}\,[\texttt{invoke}\ \$addryield]\,\{I\}$. More precisely, given an invariant I and a frame F, we wish to prove that

$$I \mathrel{-\!\!*} addryield \overset{\texttt{wf}}{\longmapsto} cl_{\text{yield}} \mathrel{-\!\!*} addrt \overset{\texttt{wtag}}{\longmapsto} \square\,([] \to []) \mathrel{-\!\!*} \mathsf{wp}\,[\,\texttt{invoke}\ addryield\,];F \left\{ w\ F',\ \begin{array}{l} w = \texttt{immV}\ []\ast F = F'\ast \\ I \ast addryield \overset{\texttt{wf}}{\longmapsto} cl_{\text{yield}} \end{array} \right\}$$

where *addryield* is $F.\mathsf{inst.funcs}[\$yield]$ (the address of function $yield in the store), *addrt* is $F.\mathsf{inst.tags}[\$t]$ (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 $addryield \overset{\texttt{wf}}{\longmapsto} cl_{\text{yield}}$ is necessary to reason about the act of invoking, and is given back in the post-condition; the tag resource $addrt \overset{\texttt{wtag}}{\longmapsto} \square\,([] \to [])$ is necessary to run the actual code of the function (recall it contains a `suspend` instruction; we will showcase in §4.4.3 the part where this resource is used) and is persistent, hence there is no need to explicitly repeat it 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 §4.3.2) 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:

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

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

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

with

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

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 WebAssembly 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 $[\texttt{label}_0\{[]\}\ \mathit{code}_{\text{yield}}\ \texttt{end}]$ instead of the `block` instruction; and finally, applying rule WP-BIND-LABEL from above, our goal becomes

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

i.e. we are now entirely focused on the code of $yield. As displayed in Fig. 4.4, that code is just one line: $[\texttt{suspend}\ \$t]$. In the following sections, we build up to the Iris-WasmFX proof rules for the new stack switching instructions, which we need to finish the current proof in §4.4.3.

4.4.2 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. This means that it cannot be given a weakest precondition by itself. It could be given a weakest precondition within a wider context that allows it to continue executing, but this would be inconveniently not modular: it would be impossible to reason about `suspend` in isolation, or even specify it, and one would have to reason about the whole program up to the enclosing handler.

In the context of 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 any modularity.

One solution could be to make suspend instructions a logical value, perhaps noted `suspendV`, just like Rao et al. [89] did for the `br` instruction in Iris-Wasm. However, while this approach works reasonably for `br`, which can only break within a function which is likely verified at the same time as the `br` it contains, it is not suitable for `suspend`, which can escape far out of its immediate context, including into code that is not at hand.

Hazel's Extended Weakest Precondition Instead, the approach of Hazel [18–20] is to extend the weakest precondition with a new component that locally accounts for non-local effects. Hazel's *extended weakest precondition*

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

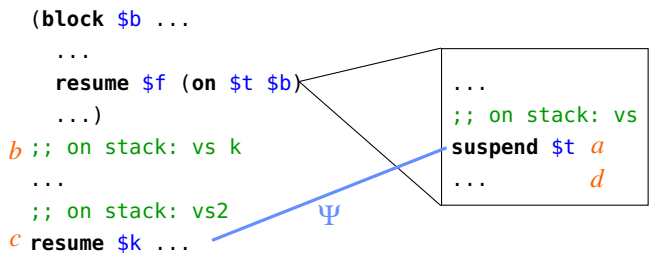


Figure 4.9: Ψ relates the payload of the suspend (raised at *a*, and control flow going to *b*) to the conditions under which the continuation can be resumed (resumption at *c*, control flow going to *d*).

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 Ψ intuitively relates the effect's payload (some values *vs*) with a resumption condition (a predicate on values) describing the desired behaviour of the program when the suspended continuation is later resumed, as illustrated in Fig. 4.9. 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. $\Psi\ v\ \Phi$ is thus a proposition that should hold whenever an effect is performed (site *a* in Fig. 4.9) with payload *v*, and $\Phi(w)$ is expected to hold when the continuation is later resumed (site *c* in Fig. 4.9) with argument *w*. This proposition must be established when suspending (site *a* in Fig. 4.9) and can be relied upon when handling (site *b* in Fig. 4.9). 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*. We define an effect to be of the form `suspendE`, `switchE` or `throwE`. For instance, effect `suspendE` *vs addr H* represents expression $H[\texttt{suspend.addr}\ vs\ addr]$ where *H* is syntactically enforced to not contain a `prompt` that captures the tag with address *addr* (if such a prompt exists, the expression is not stuck and thus not a logical effect). 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, our extended weakest precondition uses a *family* of protocols, indexed by tag addresses, rather than a single protocol. This is so that every effect (identified by its tag's address) can have a different specified behaviour. Hence, our custom extended weakest-precondition is of the form

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

where the WebAssembly frame F comes alongside the expression es, the family Ψ of protocols 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. We now say more about the meaning and usage of the protocols.

Practical Protocols While in principle a protocol can be any predicate at all, de Vilhena and Pottier [19] define a specific constructor, inspired by session types [46], which can be used to create protocols that they found were sufficient for most practical examples: $!x(v_1)\langle P\rangle\ ?y(v_2)\langle Q\rangle$. Informally, this can be read as "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 from the handler". x and y are quantified variables that make the formula more general.

More formally, this protocol applied to values vs and predicate Φ is the predicate

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

where the fact that the quantifiers quantify all the way to the right means that x can be mentioned in v_1, P, v_2 and Q; and y can be mentioned in v_2 and Q.

Another notable protocol is the bottom protocol $\bot$, defined as being false for any arguments. This symbolises code that is not allowed to perform any effects. In practice, we often use a family of bottom protocols at top-level, and only start introducing non-bottom protocols when reasoning about continuations run by way of a `resume` instruction. 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.

Example For our running 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, so the protocol family is the bottom protocol $\bot$ for all tags, except the tag address corresponding to local index $t, for which the protocol is

$$!()\{I\}\ ?()\{I\}$$

Let us use notation Ψ_0 to denote that protocol family.

Definition We define the extended weakest precondition for Iris-WasmFX formally in Fig. 4.10. We give an informal reading before explaining the more technical elements. The definition reads as follows: we inspect expression es.

- In the case where es is a value (the *Some* case in the first match in Fig. 4.10), we check (as usual) that the postcondition holds.
- In the case where es is an effect[11] (the *Some* case in the second match in Fig. 4.10), that is, a `suspend.addr` with some values vs and a tag address $addr$,

[11] This is the point at which it is crucial that inspecting the expression without knowledge of the store is enough to identify that it is an effect and what its payload is, cf. the discussion in §4.3.3.

$$
\begin{array}{l}
\mathsf{ewp}\, es ; F\, \langle \Psi \rangle\, \{ w\ F, \Phi(w,F) \} = \\
\quad \text{match to\_val}(es) \text{ with} \\
\quad\quad \mid \mathsf{Some}\ w \implies \Rrightarrow \Phi(w,F) \\
\quad\quad \mid \mathsf{None} \implies \\
\quad\quad\quad \text{match to\_eff}(es) \text{ with} \\
\quad\quad\quad\quad \mid \mathsf{Some}\ (\texttt{suspendE}\ vs\ addr\ H) \implies \\
\quad\quad\quad\quad\quad (\uparrow (\Psi\ addr))(\texttt{immV}\ vs)\ (\lambda w, \triangleright \mathsf{ewp}\, H[w] ; F\, \langle \Psi \rangle\, \{ w\ F, \Phi(w,F) \}) \\
\quad\quad\quad\quad \mid \mathsf{None} \implies \\
\quad\quad\quad\quad\quad \forall S, \text{state\_interp}(S) \Rrightarrow \\
\quad\quad\quad\quad\quad\quad (\exists S'\ F'\ es', (S,F,es) \hookrightarrow (S',F',es')) * \\
\quad\quad\quad\quad\quad\quad \forall S'\ F'\ es', (S,F,es) \hookrightarrow (S',F',es') \Rrightarrow \\
\quad\quad\quad\quad\quad\quad\quad \triangleright (\text{state\_interp}(S') * \mathsf{ewp}\, es' ; F'\, \langle \Psi \rangle\, \{ w\ F, \Phi(w,F) \})
\end{array}
$$

Figure 4.10: The extended weakest precondition of Iris-WasmFX

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 $\texttt{immV}\ vs$ is equal to v and we have the resources P; and we 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 $H[w]$.

- In the case where es is neither a value nor an effect (the *None* case in the second match in Fig. 4.10), we require (as usual) that for any *valid*[12] 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 statement holds recursively of the pair (F',es').

We now give a short explanation of some of the technical symbols in Fig. 4.10 (like $\Rrightarrow$, $\triangleright$ and $\uparrow$).

First, the notations $\Rrightarrow$ and $\triangleright$ are Iris modalities. The update modality $\Rrightarrow$ means Iris ghost state can be updated. A reader unfamiliar with Iris can think of it as a regular implication for simplicity. 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 largely be ignored by readers unfamiliar with Iris.

Second, the functions to_val and state_interp are language-specific functions present in the standard Iris definition of a weakest precondition. The first attempts to transform an expression into a value and returns None if the expression is not a value, and the second describes what a *valid* store is. Iris-Wasm defines these two functions for WebAssembly, and we use mostly unchanged definitions in Iris-WasmFX.

[12]Technically, this means we own all relevant ghost resources [52].

Third, the function to_eff is a language-specific function present 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 stack switching.

Finally, the upwards closure of the protocol $\uparrow \Psi$ is present in Hazel's definition [18, 19] of the extended weakest precondition in order to allow choosing a stronger protocol than Ψ. This is useful in practice but can be ignored for simplicity.

4.4.3 Proof Rules

In this section, we show the reasoning rules for the new instructions of stack switching. At time of writing, our program logic only encompasses the most important instructions of the extension: `suspend`, `resume` and `cont.new`. These are enough to implement meaningful examples like the coroutine module from §4.2. Support for all other instructions will be available shortly.

Building Continuations The proof rule for `cont.new` is straightforward and mirrors its reduction rule:

EWP-CONTNEW

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

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.

Suspend The rule for `suspend` simply translates the tag annotation of the instruction and then relies on the protocol:

EWP-SUSPEND

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

The last premise indicates that the payload vs must be valid with regards to the protocol, and that if the continuation is ever resumed, postcondition Φ must 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`.

Example Let us resume our proof of the specification for the $yield function from our example in Figure 4.4, which we started in §4.4.1. Recall that we wish to prove that, for all F and I,

$$I \mathrel{-\!\!*} \mathit{addryield} \overset{\texttt{wf}}{\mapsto} \mathit{cl}_{\text{yield}} \mathrel{-\!\!*} \mathit{addrt} \overset{\texttt{wtag}}{\mapsto} \square\,([] \to []) \mathrel{-\!\!*} \mathsf{ewp}\,[\texttt{invoke}\ \mathit{addryield}];F\,\langle \Psi_0 \rangle \left\{ w\ F',\ \begin{array}{l} w = \texttt{immV}\ []\ast F = F'\ast \\ I \ast \mathit{addryield} \overset{\texttt{wf}}{\mapsto} \mathit{cl}_{\text{yield}} \end{array} \right\}$$

where $\mathit{addryield} = F.\mathsf{inst.funcs}[\$\mathsf{yield}]$, $\mathit{addrt} = F.\mathsf{inst.tags}[\$\mathsf{t}]$ and $\mathit{cl}_{\text{yield}}$ is the closure of the $yield function.

Notice we now phrase this specification with our extended weakest precondition and have added a protocol family, the family Ψ_0 defined above. The beginning of the proof can, however, still be done exactly as described in §4.4.1.

We are then left with proving an extended weakest precondition statement for the code of the function itself. Since this code is only one line, we can finish the proof by invoking rule EWP-SUSPEND: we need the tag resource, which we have, and we need to show that protocol $\Psi_0(\$\mathsf{t})$ is followed. Recall that this protocol is $!()\{I\}\ ?()\{I\}$, meaning we must prove that the payload is the empty list (which it is), we must have I (which we do), and prove that I implies the post-condition, which is trivial.

Resume Finally, let us discuss the rule for `resume`. 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 hinted at earlier, the simple EWP-LABEL rule would be unsound due to the presence of effects. Instead, we prove the following rule:

EWP-PROMPT

$$\begin{array}{ll} (1) & (\forall \mathit{addr}, \mathit{addr} \notin \mathit{dccs} \implies \Psi(\mathit{addr}) = \Psi'(\mathit{addr}))\ \ast \\ (2) & \mathrm{CExp}(\mathit{es})\ \ast\ (\forall F, \neg\Phi(\texttt{trapV}, F))\ \ast\ \neg\Phi'(\texttt{trapV})\ \ast \\ (3) & \mathsf{ewp}\,\mathit{es};\varnothing\,\langle\Psi\rangle\,\{w\ F, \Phi(w,F)\}\ \ast \\ (4) & (\forall w, \Phi(w,\varnothing) \mathrel{-\!\!*} \mathsf{ewp}\,[\texttt{prompt}_{ts}\{\mathit{dccs}\}\ w\ \texttt{end}];\varnothing\,\langle\Psi'\rangle\,\{w\ \_, \Phi'(w)\})\ \ast \\ (5) & \forall \mathit{dcc} \in \mathit{dccs}, \langle\Psi\rangle\,\{\Phi\}\,\mathit{dcc};\ \mathit{ts}_2\,\langle\Psi'\rangle\,\{\lambda v\ \_, \Phi'(v)\} \\ \hline & \mathsf{ewp}\,[\texttt{prompt}_{ts}\{\mathit{dccs}\}\ \mathit{es}\ \texttt{end}];F\,\langle\Psi'\rangle\,\{w\ F', \Phi'(w)\ast F = F'\} \end{array}$$

Looking at lines (3) and (4) only, this rule resembles a normal bind rule: on line (3), we focus reasoning on the body es of the `prompt` instruction, and on line (4), we inject into the `prompt` the value w that es reduces to, and reason about the resulting expression.

Notice that the protocol family used to reason about the body es need not be the same as the one used to reason about the `prompt` instruction itself. This makes sense, as 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` instruction's 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:

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

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` instructon (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').

Let us now explain both the meaning of line (5) from rule EWP-PROMPT, and why frame-agnosticity is crucial. The idea of line (5) is to specify the behaviour of the program when an effect handled by one of the clauses of the `prompt` is performed using the `suspend` instruction. Recall that in that case, the 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 and 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 $vs +\!\!+ [\texttt{ref.cont}\ kaddr; \texttt{br}\ ilab]$ expression, where *kaddr* is the address of the newly created continuation. Little is known about the exact form of that continuation, other than that the `suspend` instruction will have followed a protocol from family Ψ. More precisely, we define the *clause triple* from line (5) as follows:

$$\begin{array}{l}\langle\Psi\rangle\,\{\Phi\}\,\texttt{on}\;taddr\;ilab;\;ts\,\langle\Psi'\rangle\,\{\Phi'\} = \\ \quad \exists ts_1, ts_2, taddr \xmapsto{\texttt{wtag}} \square\,(ts_1 \to ts_2) * \\ \quad \forall vs\;kaddr\;H, \\ \quad\quad kaddr \xmapsto{\texttt{wcont}} (ts_2 \to ts, H) \mathrel{-\!\!*} \\ \quad\quad\quad (\uparrow(\Psi\;taddr))\;(\texttt{immV}\;vs)\;(\lambda w, \triangleright \mathsf{ewp}\,H[w]\,;\varnothing\,\langle\Psi\rangle\,\{w\;F, \Phi(w,F)\}) \mathrel{-\!\!*} \\ \quad\quad\quad \mathsf{ewp}\,vs +\!\!+\,[\texttt{ref.cont}\;kaddr; \texttt{br}\;ilab]\,;\varnothing\,\langle\Psi'\rangle\,\{w\;F, \Phi'(w,F)\}\end{array}$$

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 $\Psi\;taddr$. 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. To illustrate this, let us consider the case where $\Psi\;taddr$ is of the form $!x(v)\{P\}\;?y(w)\{Q\}$. In that case,

$$\begin{array}{l}\langle\Psi\rangle\,\{\Phi\}\,\texttt{on}\;taddr\;ilab;\;ts\,\langle\Psi'\rangle\,\{\Phi'\} = \\ \quad \exists ts_1, ts_2, taddr \xmapsto{\texttt{wtag}} \square\,(ts_1 \to ts_2) * \\ \quad \forall kaddr\;H\;x, \\ \quad\quad kaddr \xmapsto{\texttt{wcont}} (ts_2 \to ts, H) \mathrel{-\!\!*} \\ \quad\quad\quad P \mathrel{-\!\!*} (\forall y, Q \mathrel{-\!\!*} \triangleright \mathsf{ewp}\,H[w]\,;\varnothing\,\langle\Psi\rangle\,\{w\;F, \Phi(w,F)\}) \mathrel{-\!\!*} \\ \quad\quad\quad \mathsf{ewp}\,v +\!\!+\,[\texttt{ref.cont}\;kaddr; \texttt{br}\;ilab]\,;\varnothing\,\langle\Psi'\rangle\,\{w\;F, \Phi'(w,F)\}\end{array}$$

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

Notice the presence of the empty frame in the last two lines of definition of the clause triple. In theory, 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. Ultimately this choice is 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.

The only remaining difficulty is bridging the frame F in the conclusion of EWP-PROMPT with the empty frame in premise (3). To this end, we prove the following lemma, which says that if an expression es verifying $\mathrm{CExp}(es)$ is safe to run under the empty frame, then the same expression can execute safely with any frame, resulting in an unchanged frame and with the same post-condition on the resulting value:

$$\text{EWP-EMPTY-FRAME}\quad \frac{\mathrm{CExp}(es) \;*\; \mathsf{ewp}\,es\,;\varnothing\,\langle\Psi\rangle\,\{w\;\_, \Phi(w)\}}{\mathsf{ewp}\,es\,;F\,\langle\Psi\rangle\,\{w\;F', \Phi(w) \;*\; F = F'\}}$$

Resuming Rule We can now state the rule for the `resume` instruction itself:

EWP-RESUME

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

Most of the premises are the same as for EWP-PROMPT. The additional premises are the continuation resource on line (2), and the premises on lines (1) and (2) which simply 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 (5) and (6).

The rest of the premises are the ones from EWP-PROMPT, allowing to bind into the actual code of the continuation being resumed. The only missing premise is $\mathrm{CExp}(H[vs])$, which we get for free since our continuation resource enforces that the context H is safe.

4.4.4 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 Fig. 4.10. Our Rocq development also includes proof rules for every instruction of stack switching, with at time of writing the exception of `switch`, `cont.bind`, `resume_throw`, and the exception handling suite.[13]

Theorem 10 (Soundness)**.** *All proof rules displayed in this paper have been proven sound with regards to the model in the Rocq proof assistant.*

The Iris separation logic framework also comes with an *adequacy* result [52, §6.4]. What this means is that we can prove in the Rocq proof assistant that a weakest precondition statement implies a simpler statement phrased in terms of the operational semantics of the language and making no mention of Iris. For example, when reasoning about a concrete execution of our coroutine example from §4.2, the final result is a theorem (Theorem 12) on execution traces in WasmFXCert that makes no mention of Iris constructs. In other words, Iris is not in our trusted computing base, only the Rocq proof assistant itself is.

Our extended weakest precondition statement also satisfies adequacy:

[13] We expect to complete our program logic with support for these instructions by the time this manuscript is submitted for publication.

Theorem 11 (Adequacy)**.** *If*

$$\text{state\_interp}(S) \ * \ \mathit{ewpes}; F \ \langle \bot \rangle \ \{w \ F, \Phi(w, F)\}$$

for some pure *predicate* Φ *(meaning* Φ *does not contain any resourceful Iris propositions), then for any reduction trace* $(S, F, es) \hookrightarrow^\star (S', F', vs)$ *(where vs are values), it holds that* $\Phi(vs, F')$.

The proof is very similar to the adequacy theorem in plain Iris.

In the next section, we show how we use the rules described in this section on our running example.

4.5 Case Study: Coroutine Library

We now revisit our running coroutine example from §4.2, focusing on proving the implementation's specification (§4.5.1) and the simple client we introduced in Fig. 4.1 (§4.5.2).

4.5.1 Library implementation

The code for the coroutine library module is shown in Fig. 4.4. We presented an informal version of its specification in §4.2. Now that we have introduced the full machinery of Iris-WasmFX, we show the full specification in Fig. 4.11.

Compared to the snapshot given in §4.2, 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.

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 protocol family Ψ 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 protocol family Ψ 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.

$$
\begin{array}{l}
\exists \mathit{addryield}\ \mathit{addrpar}\ \mathit{cl}_{\text{yield}}\ \mathit{cl}_{\text{par}},\\
J_{\text{yield}} * J_{\text{par}} * \quad \text{(we own the closures for \$par and \$yield, ...)}\\
\forall P_1\ Q_1\ P_2\ Q_2\ I, \exists \Psi,\\
\text{(... and we have specifications for both functions)}\\
\text{(for function \$yield:)}\\
\square \left(\forall F, I \mathrel{-\!\!*} J_{\text{yield}} \mathrel{-\!\!*} \right.\\
\qquad \left. \mathsf{ewp}\,[\texttt{invoke}\ \mathit{addryield}]; F\,\langle \Psi \rangle \left\{ w\ F',\ \begin{array}{l} w = \texttt{immV}\ [] * F = F' * \\ I * J_{\text{yield}} \end{array} \right\} \right) *\\
\text{(and for function \$par:)}\\
\forall \mathit{addrf}_1\ \mathit{addrf}_2\ \mathit{cl}_1\ \mathit{cl}_2,\\
\left(\begin{array}{l}
\forall F, P_1 \mathrel{-\!\!*} P_2 \mathrel{-\!\!*} I \mathrel{-\!\!*} J_1 \mathrel{-\!\!*} J_2 \mathrel{-\!\!*} J_{\text{yield}} \mathrel{-\!\!*} J_{\text{par}} \mathrel{-\!\!*} \\
\text{(precondition of specification for \$par contains a specification for \$f}_1\text{:)}\\
\square \left(\forall F',\ P_1 \mathrel{-\!\!*} I \mathrel{-\!\!*} J_1 \mathrel{-\!\!*} J_{\text{yield}} \mathrel{-\!\!*} \right.\\
\qquad \left. \mathsf{ewp} \left[\begin{array}{l} \texttt{ref.func}\ \mathit{addrf}_1; \\ \texttt{ref.call} \end{array} \right]; F'\,\langle \Psi \rangle \left\{ w\ F'',\ \begin{array}{l} w = \texttt{immV}\ [] * \\ F' = F'' * \\ Q_1 * I * \\ J_1 * J_{\text{yield}} \end{array} \right\} \right) \mathrel{-\!\!*} \\
\text{(precondition of specification for \$par contains a specification for \$f}_2\text{:)}\\
\square \left(\forall F',\ P_2 \mathrel{-\!\!*} I \mathrel{-\!\!*} J_2 \mathrel{-\!\!*} J_{\text{yield}} \mathrel{-\!\!*} \right.\\
\qquad \left. \mathsf{ewp} \left[\begin{array}{l} \texttt{ref.func}\ \mathit{addrf}_2; \\ \texttt{ref.call} \end{array} \right]; F'\,\langle \Psi \rangle \left\{ w\ F'',\ \begin{array}{l} w = \texttt{immV}\ [] * \\ F' = F'' * \\ Q_2 * I * \\ J_2 * J_{\text{yield}} \end{array} \right\} \right) \mathrel{-\!\!*} \\
\mathsf{ewp} \left[\begin{array}{l} \texttt{ref.func}\ \mathit{addrf}_1; \\ \texttt{ref.func}\ \mathit{addrf}_2; \\ \texttt{invoke}\ \mathit{addrpar} \end{array} \right]; F\,\langle \bot \rangle \left\{ w\ F',\ \begin{array}{l} w = \texttt{immV}\ [] * F = F' * \\ Q_1 * Q_2 * I * \\ J_1 * J_2 * J_{\text{yield}} * J_{\text{par}} \end{array} \right\}
\end{array} \right)
\end{array}
$$

Where J_{yield} is a shorthand for $\mathit{addryield} \overset{\texttt{wf}}{\longmapsto} \mathit{cl}_{\text{yield}}$,
J_{par} is a shorthand for $\mathit{addrpar} \overset{\texttt{wf}}{\longmapsto} \mathit{cl}_{\text{par}}$,
J_1 is a shorthand for $\mathit{addrf}_1 \overset{\texttt{wf}}{\longmapsto} \mathit{cl}_1$, and J_2 is a shorthand for $\mathit{addrf}_2 \overset{\texttt{wf}}{\longmapsto} \mathit{cl}_2$

Figure 4.11: The specification for the coroutines module

$$
\begin{array}{rcl}
LI & = & \exists H_a, \\
& & \left(\begin{array}{l}
I * J * \$\mathsf{current} \overset{\texttt{wcont}}{\longmapsto} H_a * \\
\left(\begin{array}{l}
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 * \$\mathsf{f}_1 \overset{\texttt{wf}}{\longmapsto} cl_1 * J \end{array} \right\}
\end{array}\right) * \\
\left(\begin{array}{l}
\$\mathsf{next\_is\_done} = 0 * \exists H_b, \$\mathsf{next} \overset{\texttt{wcont}}{\longmapsto} H_b * \\
\left(\begin{array}{l}
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 * \$\mathsf{f}_2 \overset{\texttt{wf}}{\longmapsto} cl_2 * J \end{array} \right\}
\end{array}\right) \\
\vee \quad (\$\mathsf{next\_is\_done} = 1 * Q_2 * \$\mathsf{f}_2 \overset{\texttt{wf}}{\longmapsto} cl_2)
\end{array}\right)
\end{array}\right) \\
& \vee & \left(\begin{array}{l}
I * J * \$\mathsf{current} \overset{\texttt{wcont}}{\longmapsto} H_a * \\
\left(\begin{array}{l}
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 * \$\mathsf{f}_2 \overset{\texttt{wf}}{\longmapsto} cl_2 * J \end{array} \right\}
\end{array}\right) * \\
\left(\begin{array}{l}
\$\mathsf{next\_is\_done} = 0 * \exists H_b, \$\mathsf{next} \overset{\texttt{wcont}}{\longmapsto} H_b * \\
\left(\begin{array}{l}
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 * \$\mathsf{f}_1 \overset{\texttt{wf}}{\longmapsto} cl_1 * J \end{array} \right\}
\end{array}\right) \\
\vee \quad (\$\mathsf{next\_is\_done} = 1 * Q_1 * \$\mathsf{f}_1 \overset{\texttt{wf}}{\longmapsto} cl_1)
\end{array}\right)
\end{array}\right)
\end{array}
$$

where J is $addryield \overset{\texttt{wf}}{\longmapsto} cl_{\text{yield}}$.

Figure 4.12: Loop invariant for proving the specification of the $par function

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 Ψ as the protocol family Ψ_0 described in §4.4.2. The function resources for $yield and $par are provided by the host language's program logic (see §4.3.5) during the instantiation process.

Then it remains to prove the two specifications; we gave an outline of the proof for $yield in §4.4.1 and §4.4.3. 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 Fig. 4.12.

The invariant is a disjunction where the left case corresponds to the case where local variable $current is the continuation obtained originally from function $f1 and $next is the continuation obtained originally from function $f2; and the right disjunct is vice-versa, $current is $f2 and $next is $f1. 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.

Lines 13 to 20 create the continuations; using rule EWP-CONTNEW, we get the continuation resources corresponding to calling functions $\$f_1$ and $\$f_2$. Then we enter the loop on line 21: for this we must establish that the loop invariant LI holds, which is does (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. $Q_i * I * \$f_i \overset{\texttt{wf}}{\longmapsto} cl_i$ 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 $Q_j * \$f_j \overset{\texttt{wf}}{\longmapsto} cl_j$ also holds and we can conclude the proof. If $next_is_done is 0, then we update its value (lines 27-28), place $next is $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 $\overset{\texttt{wcont}}{\longmapsto}$ 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 `corountines_implementation_code.v`.

4.5.2 Client module

We illustrate usage of our coroutine module on a simple client whose code can be seen in Fig. 4.1. For this client, we show the following specification:

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

meaning that as long as we hold the corresponding function closure resource, we can run the $main function safely, and if it returns, it will return two integer values 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 Fig. 4.13. 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 use *ghost resources* that assist in reasoning but do not directly correspond to a piece of the WebAssembly state. We use the *one-shot algebra* [50], 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 Fig. 4.13, 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 $f1 and $f2. For $f1, the proof follows this scheme:

(1) $\{I\}$ — Start with I (and vacous P_1)
i32.const 42
`global.set` $x
(2) $\left\{I * \boxed{\text{Shot}}^{\gamma_x}\right\}$ — Reestablish I but keep a copy of $\boxed{\text{Shot}}^{\gamma_x}$
`call` $yield
(3) $\left\{I * \boxed{\text{Shot}}^{\gamma_x}\right\}$ — Using the specification for $yield
i32.const 1
`global.set` $y
(4) $\left\{I * \boxed{\text{Shot}}^{\gamma_x}\right\}$ — 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) $\left\{ I * addrb \overset{wg}{\mapsto} - * \left((addra \overset{wg}{\mapsto} 1 * \boxed{\text{Shot}}^{\gamma_y}) \vee addra \overset{wg}{\mapsto} 0 \right) \right\}$

`call $yield` — Using the specification for $yield

(3) $\left\{ I * addrb \overset{wg}{\mapsto} - * \left((addra \overset{wg}{\mapsto} 1 * \boxed{\text{Shot}}^{\gamma_y}) \vee addra \overset{wg}{\mapsto} 0 \right) \right\}$

`global.get $x`

`global.set $b`

(4) $\left\{ I * \left((addra \overset{wg}{\mapsto} 1 * addrb \overset{wg}{\mapsto} 42) \vee (addra \overset{wg}{\mapsto} 0 * addrb \overset{wg}{\mapsto} -) \right) \right\}$

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 $addra \overset{wg}{\mapsto} 1$, 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.

Final Theorem Using the Adequacy Theorem from §4.4.4 on the specification above, we can prove the following theorem, which makes no mention of Iris:

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

Figure 4.13: Resources for the specification of the client of our coroutine library.

Theorem 12 (Coroutines Module)**.** *If Cor is the module whose code is shown in Fig. 4.4, and Cl is the module whose code is shown in Fig. 4.1, and if the host program*

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

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

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

4.6 Related Work

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

Formalising stack switching There is an ongoing project [62] to model stack switching in SpecTec [139]. SpecTec is a domain-specific language designed specifically to write the semantics of WebAssembly, with tooling to generate readable LaTeX definitions, prose explanations of the opsem, and theorem prover definitions. SpecTec has been officially adopted for authoring future editions of the Wasm spec [101], and could eventually replace the hand-written definitions of WasmFXCert and thereby avoid duplicating work. However, this would require more mature theorem prover output, and would in any case not replace the proof aspects of §4.3.4. 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 §4.3.3.

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, WebAssembly. Osiris [17, 106, 107] 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 WebAssembly, 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 WebAssembly 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 stack switching, in which effect handlers are exposed for code generation. 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 WebAssembly standard. The combined existence of Osiris and Iris-WasmFX opens up the possibility of formally relating programs in OCaml and WebAssembly, be it by verified compilation, translation validation, or other means (see below).

Integration with JavaScript/ECMAScript The extensive use of non-local control flow in JavaScript/ECMAScript is one of the main motivations for extending WebAssembly with stack switching. Khayam et al. [55] formalise parts of ECMAScript and describe how, even though they do not model ECMAScript 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) [65] is the 'reverse' of stack switching: it allows plain WebAssembly, 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 WebAssembly 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 WebAssembly is that WebAssembly is only meant to suspend or resume the execution of a JavaScript via promises. Verifying this is outside of the scope of this paper, but we lay the foundations to do so.

Compilation to stack switching WebAssembly is designed as a compilation target, and so the introduction stack switching raises the question of the correctness of compilation of these infamously challenging non-local control flow primitives. RichWasm [30] is a richly typed language based on WebAssembly which was designed as a target for typed compilation enforcing strong memory safety guarantees. Extending RichWasm to cover stack switching as modelled by WasmFXCert would make it possible to consider source languages featuring non-local control flow.

Frame reasoning Adjoint separation logic [128] elegantly deals with the frame problem discussed in §4.4.1 by using modalities that make a given frame available or lock it away.

4.7 Limitations

All the results in this paper are about the specification of stack switching, not any actual implementation, and thus we still need to trust that the implementation is faithful to the specification.

At the time of writing, our program logic only covers key instructions of stack switching, and there are no proof rules yet for `switch`, `cont.bind`, `resume_throw`, nor any of the instructions of the exception handling suite. The WasmFXCert mechanisation and the proof of type soundness, however, cover the entire language.

While stack switching officially builds on WebAssembly 2.0, our mechanisation WasmFXCert and our program logic Iris-WasmFX only support 1.0 together with a few select instructions from WebAssembly 2.0.

4.8 Conclusion

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

Bibliography

[1] Inteface types proposal for webassembly. Technical report, 2022. URL https://github.com/WebAssembly/interface-types/blob/main/proposals/interface-types/Explainer.md. 17

[2] Amal Ahmed. *Semantics of Types for Mutable State*. PhD thesis, Princeton University, 2004. 95

[3] Amal Ahmed, Matthew Fluet, and Greg Morrisett. L^3: A linear language with locations. *Fundam. Informaticae*, 77(4):397–449, 2007. URL http://content.iospress.com/articles/fundamenta-informaticae/fi77-4-06. 19

[4] Heejin Ahn and Ben Titzer. Exception handling proposal for webassembly. Technical report, 2022. URL https://webassembly.github.io/exception-handling/. 4, 126

[5] Thomas Bauereiss, Brian Campbell, Thomas Sewell, Alasdair Armstrong, Lawrence Esswood, Ian Stark, Graeme Barnes, Robert N. M. Watson, and Peter Sewell. Verified security for the morello capability-enhanced prototype arm architecture. In Ilya Sergey, editor, *Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings*, volume 13240 of *Lecture Notes in Computer Science*, pages 174–203. Springer, 2022. doi: 10.1007/978-3-030-99336-8_7. URL https://doi.org/10.1007/978-3-030-99336-8_7. 74

[6] Lars Birkedal and Aleš Bizjak. Lecture notes on iris: Higher-order concurrent separation logic. Technical report, Aarhus University, 2017. 42

[7] Tyler K. Bletsch, Xuxian Jiang, Vincent W. Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In Bruce S. N. Cheung, Lucas Chi Kwong Hui, Ravi S. Sandhu, and Duncan S. Wong, editors, *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ASIACCS 2011, Hong Kong, China, March 22-24, 2011*, pages 30–40. ACM, 2011. doi: 10.1145/1966913.1966919. URL https://doi.org/10.1145/1966913.1966919. 13

[8] M. Bodin, P. Gardner, J. Pichon, and X. Rao. Wasmcert-coq, version 2.0, 2025. URL https://github.com/WasmCert/WasmCert-Coq?tab=readme-ov-file. 7

[9] Joachim Breitner, Philippa Gardner, Jaehyun Lee, Sam Lindley, Matija Pretnar, Xiaojia Rao, Andreas Rossberg, Sukyoung Ryu, Wonho Shin, Conrad Watt, and Dongjun Youn. Wasm spectec: Engineering a formal language standard. *CoRR*, abs/2311.07223, 2023. doi: 10.48550/ARXIV.2311.07223. URL https://doi.org/10.48550/arXiv.2311.07223. 106

[10] Stephen Brookes. A semantics for concurrent separation logic. *Theor. Comput. Sci.*, 375(1-3):227–270, 2007. doi: 10.1016/J.TCS.2006.12.034. URL https://doi.org/10.1016/j.tcs.2006.12.034. 8

[11] Stephen Brookes and Peter W. O'Hearn. Concurrent separation logic. *ACM SIGLOG News*, 3(3):47–65, 2016. doi: 10.1145/2984450.2984457. 122

[12] Matt Butcher. How to think about webassembly (amid the hype), February 2022. URL https://www.fermyon.com/blog/how-to-think-about-wasm. 74

[13] Nicholas Carlini, Antonio Barresi, Mathias Payer, David A. Wagner, and Thomas R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In Jaeyeon Jung and Thorsten Holz, editors, *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, pages 161–176. USENIX Association, 2015. URL https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/carlini. 13

[14] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. Hardware support for fast capability-based addressing. In Forest Baskett and Douglas W. Clark, editors, *ASPLOS-VI Proceedings - Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, USA, October 4-7, 1994*, pages 319–327. ACM Press, 1994. doi: 10.1145/195473.195579. URL https://doi.org/10.1145/195473.195579. 14

[15] Chromium. Memory safety, 2020. URL https://www.chromium.org/Home/chromium-security/memory-safety. 12

[16] Lin Clark. Announcing the Bytecode Alliance: Building a secure by default, composable future for WebAssembly, November 2019. URL https://hacks.mozilla.org/2019/11/announcing-the-bytecode-alliance/. 74

[17] Arnaud Daby-Seesaram, François Pottier, and Armaël Guéneau. Osiris: an iris-based program logic for ocaml. OCaML'23: Caml Users and Developers Workshop 2023, September 2023. https://icfp23.sigplan.org/details/ocaml-2023-papers/7/Osiris-an-Iris-based-program-logic-for-OCaml. 17, 159

[18] Paulo de Vilhena. *Proof of Programs with Effect Handlers. (Preuve de Programmes avec Effect Handlers)*. PhD thesis, Paris Cité University, France, 2022. URL https://tel.archives-ouvertes.fr/tel-03891381. 16, 121, 140, 144, 148

[19] Paulo Emílio de Vilhena and François Pottier. A separation logic for effect handlers. *Proc. ACM Program. Lang.*, 5(POPL):1–28, 2021. doi: 10.1145/3434314. URL https://doi.org/10.1145/3434314. 146, 148

[20] Paulo Emílio de Vilhena and François Pottier. 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. URL https://doi.org/10.46298/lmcs-19(4:5)2023. 16, 121, 140, 144

[21] Jack B. Dennis and Earl C. Van Horn. Programming Semantics for Multiprogrammed Computations. *Commun. ACM*, 9(3):143–155, March 1966. ISSN 0001-0782. doi: 10.1145/365230.365252. URL https://doi.org/10.1145/365230.365252. 74

[22] Dominique Devriese, Lars Birkedal, and Frank Piessens. Reasoning about object capabilities with logical relations and effect parametricity. In *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*, pages 147–162. IEEE, 2016. doi: 10.1109/EuroSP.2016.22. URL https://doi.org/10.1109/EuroSP.2016.22. 75

[23] Edsger W. Dijkstra. Letters to the editor: go to statement considered harmful. *Commun. ACM*, 11(3):147–148, 1968. doi: 10.1145/362929.362947. URL https://doi.org/10.1145/362929.362947. 121

[24] Craig Disselkoen, John Renner, Conrad Watt, Tal Garfinkel, Amit Levy, and Deian Stefan. Position paper: Progressive memory safety for webassembly. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450372268. doi: 10.1145/3337167.3337171. URL https://doi.org/10.1145/3337167.3337171. 4, 12, 17, 52, 74

[25] Daniel Ehrenberg. WebAssembly JavaScript interface W3C recommendation. Technical report, W3C, December 2019. URL https://www.w3.org/TR/wasm-js-api-1/. 26, 27

[26] Fastly documentation. Compute@Edge, May 2022. URL https://docs.fastly.com/products/compute-at-edge. 74

[27] Matthias Felleisen. The theory and practice of first-class prompts. In Jeanne Ferrante and Peter Mager, editors, *Conference Record of the Fifteenth Annual*

ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988, pages 180–190. ACM Press, 1988. doi: 10.1145/73560.73576. 123, 125

[28] Xinyu Feng and Zhong Shao. Modular verification of concurrent assembly code with dynamic thread creation and termination. In Olivier Danvy and Benjamin C. Pierce, editors, *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26-28, 2005*, pages 254–267. ACM, 2005. doi: 10.1145/1086365.1086399. URL https://doi.org/10.1145/1086365.1086399. 6, 52

[29] Michael Fitzgibbons. CapableWasm: Bringing better interop down to WebAssembly, January 2022. URL https://www.youtube.com/watch?v=E44lTaa2qHk. POPL'22 student research competition presentation. 52

[30] Michael Fitzgibbons, Zoe Paraskevopoulou, Noble Mushtak, Michelle Thalakottur, Jose Sulaiman Manzur, and Amal Ahmed. Richwasm: Bringing safe, fine-grained, shared-memory interoperability down to webassembly. *Proc. ACM Program. Lang.*, 8(PLDI):1656–1679, 2024. doi: 10.1145/3656444. 4, 14, 17, 19, 106, 160

[31] Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. On the expressive power of user-defined effects: Effect handlers, monadic reflection, delimited control. *J. Funct. Program.*, 29:e15, 2019. doi: 10.1017/S0956796819000121. URL https://doi.org/10.1017/S0956796819000121. 15

[32] Aïna Linn Georges. *Designing and Proving Robust Safety of Efficient Capability Machine Programs*. PhD thesis, Aarhus University, July 2023. 13, 14, 75, 77, 86

[33] Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Sander Huyghebaert, Dominique Devriese, and Lars Birkedal. Efficient and provable local capability revocation using uninitialized capabilities. *Proc. ACM Program. Lang.*, 5(POPL):1–30, 2021. URL https://doi.org/10.1145/3434287. 14, 47, 49, 52, 95

[34] Aïna Linn Georges, Alix Trieu, and Lars Birkedal. Le temps des cerises: efficient temporal stack safety on capability machines using directed capabilities. *Proc. ACM Program. Lang.*, 6(OOPSLA1):1–30, 2022. doi: 10.1145/3527318. URL https://doi.org/10.1145/3527318. 95

[35] Aïna Linn Georges, Armaël Guéneau, Thomas Van-Strydonck, Amin Timany, Dominique Trieu, Alix Devriese, and Lars Birkedal. Cap' ou pas cap' ?: Preuve de programmes pour une machine à capacités en présence de code inconnu. In *Journées Francophones des Langages Applicatifs 2021*, April 2021. URL https://cris.vub.be/ws/portalfiles/portal/55081793/paper.pdf. 52

[36] Aïna Linn Georges, Armaël Guéneau, Thomas van Strydonck, Amin Timany, Alix Trieu, Dominique Devriese, and Lars Birkedal. Cerise: Program verification on a capability machine in the presence of untrusted code. Technical report, Aarhus University, 2022. URL https://cs.au.dk/~birke/papers/cerise.pdf. 14, 75, 86, 95, 98

[37] Aïna Linn Georges, Alix Trieu, and Lars Birkedal. Le temps des cerises: Efficient temporal stack safety on capability machines using directed capabilities (technical report). Technical report, Aarhus University, 2022. URL https://cs.au.dk/~ageorges/publications_pdfs/monotone-technical.pdf. 52

[38] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep specifications and certified abstraction layers. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 595–608. ACM, 2015. doi: 10.1145/2676726.2676975. URL https://doi.org/10.1145/2676726.2676975. 6, 53

[39] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An extensible architecture for building certified concurrent os kernels. In Kimberly Keeton and Timothy Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 653–669. USENIX Association, 2016. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu. 6, 52, 53

[40] Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. Certified concurrent abstraction layers. In Jeffrey S. Foster and Dan Grossman, editors, *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 646–661. ACM, 2018. doi: 10.1145/3192366.3192381. URL https://doi.org/10.1145/3192366.3192381. 6, 53

[41] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and J. F. Bastien. Bringing the web up to speed with webassembly. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 185–200. ACM, 2017. doi: 10.1145/3062341.3062363. URL https://doi.org/10.1145/3062341.3062363. 3, 74, 77, 120

[42] Adam Hall and Umakishore Ramachandran. An execution model for serverless functions at the edge. In Olaf Landsiedel and Klara Nahrstedt, ed-

itors, *Proceedings of the International Conference on Internet of Things Design and Implementation, IoTDI 2019, Montreal, QC, Canada, April 15-18, 2019*, pages 225–236. ACM, 2019. doi: 10.1145/3302505.3310084. URL https://doi.org/10.1145/3302505.3310084. 4

[43] Pat Hickey. How Fastly and the developer community are investing in the WebAssembly ecosystem, May 2020. URL https://www.fastly.com/blog/how-fastly-and-developer-community-invest-in-webassembly-ecosystem. 27, 74

[44] Daniel Hillerström. *Foundations for Programming and Implementing Effect Handlers*. PhD thesis, The University of Edinburgh, 2021. 15

[45] Daniel Hillerström. Typed continuations: A basis for stack-switching in wasm, September 2021. URL https://raw.githubusercontent.com/WebAssembly/meetings/main/stack/2021/presentations/2021-9-20-TypedContinuations.pdf. 16, 120

[46] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *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*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998. doi: 10.1007/BFB0053567. URL https://doi.org/10.1007/BFb0053567. 17, 146

[47] Sander Huyghebaert, Steven Keuchel, Coen De Roover, and Dominique Devriese. Formalizing, verifying and applying ISA security guarantees as universal contracts. In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, pages 2083–2097. ACM, 2023. doi: 10.1145/3576915.3616602. URL https://doi.org/10.1145/3576915.3616602. 9

[48] Koen Jacobs, Dominique Devriese, and Amin Timany. Purity of an ST monad: full abstraction by semantically typed back-translation. *Proc. ACM Program. Lang.*, 6(OOPSLA1):1–27, 2022. doi: 10.1145/3527326. URL https://doi.org/10.1145/3527326. 27

[49] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 637–650, 2015. doi: 10.1145/2676726.2676980. URL https://doi.org/10.1145/2676726.2676980. 7, 26

[50] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. Higher-order ghost state. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 256–269, 2016. doi: 10.1145/2951913.2951943. URL https://doi.org/10.1145/2951913.2951943. 157

[51] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: securing the foundations of the Rust programming language. *Proc. ACM Program. Lang.*, 2(POPL):66:1–66:34, 2018. doi: 10.1145/3158154. URL https://doi.org/10.1145/3158154. 13, 49

[52] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.*, 28:e20, 2018. doi: 10.1017/S0956796818000151. 7, 16, 26, 32, 46, 47, 87, 94, 98, 122, 139, 141, 147, 152

[53] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Safe systems programming in rust. *Commun. ACM*, 64(4):144–152, 2021. doi: 10.1145/3418295. URL https://doi.org/10.1145/3418295. 13

[54] Andrew Kennedy, Nick Benton, Jonas Braband Jensen, and Pierre-Évariste Dagand. Coq: the world's best macro assembler? In Ricardo Peña and Tom Schrijvers, editors, *15th International Symposium on Principles and Practice of Declarative Programming, PPDP '13, Madrid, Spain, September 16-18, 2013*, pages 13–24. ACM, 2013. doi: 10.1145/2505879.2505897. URL https://doi.org/10.1145/2505879.2505897. 6, 52

[55] Adam Khayam, Louis Noizet, and Alan Schmitt. 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*, pages 8:1–8:14. ACM, 2022. doi: 10.1145/3551357.3551381. URL https://doi.org/10.1145/3551357.3551381. 160

[56] Matthew Kolosick, Shravan Narayan, Evan Johnson, Conrad Watt, Michael Lemay, Deepak Garg, Ranjit Jhala, and Deian Stefan. Isolation without taxation: Near-zero-cost transitions for WebAssembly and SFI. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. ACM, January 2022. doi: 10.1145/3498688. URL https://doi.org/10.1145/3498688. 52

[57] Robbert Krebbers, Amin Timany, and Lars Birkedal. Interactive proofs in higher-order concurrent separation logic. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 205–217. ACM, 2017. doi: 10.1145/3009837.3009855. URL https://doi.org/10.1145/3009837.3009855. 49, 95

[58] Maxime Legoupil, Jean Pichon-Pharabod, Sam Lindley, and Lars Birkedal. Iris-wasmfx: Modular reasoning for wasm stack switching. Technical report. 20

[59] Maxime Legoupil, June Rousseau, Aïna Linn Georges, Jean Pichon-Pharabod, and Lars Birkedal. Iris-mswasm: Elucidating and mechanising the security invariants of memory-safe webassembly. *Proc. ACM Program. Lang.*, 8 (OOPSLA2):304–332, 2024. doi: 10.1145/3689722. URL https://doi.org/10.1145/3689722. 19, 139

[60] Maxime Legoupil, June Rousseau, Aïna Linn Georges, Jean Pichon-Pharabod, and Lars Birkedal. Artifact and Appendix of 'Iris-MSWasm: elucidating and mechanising the security invariants of Memory- Safe WebAssembly', August 2024. URL https://doi.org/10.5281/zenodo.13383121. 107

[61] Daniel Lehmann, Johannes Kinder, and Michael Pradel. Everything old is new again: Binary security of webassembly. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 217–234. USENIX Association, 2020. URL https://www.usenix.org/conference/usenixsecurity20/presentation/lehmann. 5, 13, 52, 74, 106

[62] Yalun Liang, Sam Lindley, and Andreas Rossberg. Experience report: Stack switching in wasm spectec. presented at WAW: the WebAssembly Workshop https://effect-handlers.org/static/papers/sss-waw-2025.pdf, 2025. 7, 159

[63] David Madore. The unlambda programming language. http://www.madore.org/~david/programs/unlambda/, 2001. 121

[64] Luc Maranget, Susmit Sarkar, and Peter Sewell. A tutorial introduction to the ARM and POWER relaxed memory models. http://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf, 2012. 123

[65] Francis McCabe, Thibaud Michaud, Ilya Rezvov, and Brendan Dahl. Introducing the webassembly javascript promise integration api, July 2024. URL https://v8.dev/blog/jspi. 160

[66] Francis McCabe, Sam Lindley, and WebAssembly Community Group. Stack switching webassembly proposal. https://github.com/WebAssembly/stack-switching/blob/main/proposals/stack-switching/Explainer.md, 2025. 4, 7, 16, 120

[67] J. McCarthy and P.J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In Bonnie Lynn Webber and Nils J. Nilsson, editors, *Readings in Artificial Intelligence*, pages 431–450. Morgan Kaufmann, 1981. ISBN 978-0-934613-03-3. doi: https://doi.org/10.

1016/B978-0-934613-03-3.50033-7. URL https://www.sciencedirect.com/science/article/pii/B9780934613033500337. 105

[68] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. Into the depths of C: elaborating the de facto standards. In Chandra Krintz and Emery D. Berger, editors, *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 1–15. ACM, 2016. doi: 10.1145/2908080.2908081. URL https://doi.org/10.1145/2908080.2908081. 74

[69] Alexandra E. Michael, Anitha Gollamudi, Jay Bosamiya, Evan Johnson, Aidan Denlinger, Craig Disselkoen, Conrad Watt, Bryan Parno, Marco Patrignani, Marco Vassena, and Deian Stefan. Mswasm: Soundly enforcing memory-safe execution of unsafe code. *Proc. ACM Program. Lang.*, 7(POPL):425–454, 2023. doi: 10.1145/3571208. URL https://doi.org/10.1145/3571208. 4, 12, 14, 17, 27, 52, 74, 77, 78, 80, 82, 83, 84, 104, 105

[70] Greg Morrisett, Amal J. Ahmed, and Matthew Fluet. L^3: A linear language with locations. In Pawel Urzyczyn, editor, *Typed Lambda Calculi and Applications, 7th International Conference, TLCA 2005, Nara, Japan, April 21-23, 2005, Proceedings*, volume 3461 of *Lecture Notes in Computer Science*, pages 293–307. Springer, 2005. doi: 10.1007/11417170_22. URL https://doi.org/10.1007/11417170_22. 19

[71] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. Rocksalt: better, faster, stronger SFI for the x86. In Jan Vitek, Haibo Lin, and Frank Tip, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 395–404. ACM, 2012. doi: 10.1145/2254064.2254111. URL https://doi.org/10.1145/2254064.2254111. 6, 52

[72] J. Gregory Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. In David B. MacQueen and Luca Cardelli, editors, *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*, pages 85–97. ACM, 1998. doi: 10.1145/268946.268954. URL https://doi.org/10.1145/268946.268954. 14

[73] Ike Mulder, Robbert Krebbers, and Herman Geuvers. Diaframe: automated verification of fine-grained concurrent programs in iris. In Ranjit Jhala and Isil Dillig, editors, *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, pages 809–824. ACM, 2022. doi: 10.1145/3519939.3523432. URL https://doi.org/10.1145/3519939.3523432. 48

[74] Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. Lem: reusable engineering of real-world semantics. In Johan Jeuring and Manuel M. T. Chakravarty, editors, *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pages 175–188. ACM, 2014. doi: 10.1145/2628136.2628143. URL https://doi.org/10.1145/2628136.2628143. 7, 106

[75] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. Retrofitting fine grain isolation in the firefox renderer. In *Proceedings of the 29th USENIX Conference on Security Symposium*, USA, 2020. USENIX Association. ISBN 978-1-939133-17-5. 27, 52

[76] George C. Necula. Proof-carrying code. In Peter Lee, Fritz Henglein, and Neil D. Jones, editors, *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*, pages 106–119. ACM Press, 1997. doi: 10.1145/263699.263712. URL https://doi.org/10.1145/263699.263712. 14

[77] Zhaozhong Ni and Zhong Shao. Certified assembly programming with embedded code pointers. *SIGPLAN Not.*, 41(1):320–333, January 2006. ISSN 0362-1340. doi: 10.1145/1111320.1111066. URL https://doi.org/10.1145/1111320.1111066. 6, 52

[78] Kyndylan Nienhuis, Alexandre Joannou, Thomas Bauereiss, Anthony Fox, Michael Roe, Brian Campbell, Matthew Naylor, Robert M. Norton, Simon W. Moore, Peter G. Neumann, Ian Stark, Robert N. M. Watson, and Peter Sewell. Rigorous engineering for hardware security: Formal modelling and proof in the CHERI design and implementation process. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (SP)*, pages 1003–1020. IEEE, May 2020. doi: 10.1109/SP40000.2020.00055. URL https://doi.org/10.1109/SP40000.2020.00055. 74

[79] Peter W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007. doi: 10.1016/J.TCS.2006.12.035. URL https://doi.org/10.1016/j.tcs.2006.12.035. 8

[80] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In Laurent Fribourg, editor, *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2001. doi: 10.1007/3-540-44802-0_1. URL https://doi.org/10.1007/3-540-44802-0_1. 7

[81] Scott Owens, Peter Böhm, Francesco Zappa Nardelli, and Peter Sewell. Lem: A lightweight tool for heavyweight semantics. In Marko C. J. D. van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors, *Interactive Theorem Proving - Second International Conference, ITP 2011, Berg en Dal, The Netherlands, August 22-25, 2011. Proceedings*, volume 6898 of *Lecture Notes in Computer Science*, pages 363–369. Springer, 2011. doi: 10.1007/978-3-642-22863-6_27. URL https://doi.org/10.1007/978-3-642-22863-6_27. 106

[82] Daniel Patterson, Noble Mushtak, Andrew Wagner, and Amal Ahmed. Semantic soundness for language interoperability. In Ranjit Jhala and Isil Dillig, editors, *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, pages 609–624. ACM, 2022. doi: 10.1145/3519939.3523703. URL https://doi.org/10.1145/3519939.3523703. 19

[83] Luna Phipps-Costin, Andreas Rossberg, Arjun Guha, Daan Leijen, Daniel Hillerström, K. C. Sivaramakrishnan, Matija Pretnar, and Sam Lindley. Continuing webassembly with effect handlers. *Proc. ACM Program. Lang.*, 7 (OOPSLA2):460–485, 2023. doi: 10.1145/3622814. URL https://doi.org/10.1145/3622814. 16, 120, 123, 132, 136

[84] Andrew M. Pitts. Parametric polymorphism and operational equivalence. *Math. Struct. Comput. Sci.*, 10(3):321–359, 2000. URL http://journals.cambridge.org/action/displayAbstract?aid=44651. 131

[85] Gordon Plotkin. Lambda-definability and logical relations, October 1972. URL https://homepages.inf.ed.ac.uk/gdp/publications/logical_relations_1973.pdf. 8

[86] Gordon D. Plotkin and Matija Pretnar. Handlers of algebraic effects. In Giuseppe Castagna, editor, *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*, volume 5502 of *Lecture Notes in Computer Science*, pages 80–94. Springer, 2009. doi: 10.1007/978-3-642-00590-9_7. URL https://doi.org/10.1007/978-3-642-00590-9_7. 16, 120

[87] Gordon D. Plotkin and Matija Pretnar. Handling algebraic effects. *Log. Methods Comput. Sci.*, 9(4), 2013. doi: 10.2168/LMCS-9(4:23)2013. URL https://doi.org/10.2168/LMCS-9(4:23)2013. 16, 120

[88] Andrew Prout, William Arcand, David Bestor, Bill Bergeron, Chansup Byun, Vijay Gadepally, Michael Houle, Matthew Hubbell, Michael Jones, Anna Klein, Peter Michaleas, Lauren Milechin, Julie Mullen, Antonio Rosa, Siddharth Samsi, Charles Yee, Albert Reuther, and Jeremy Kepner. Measuring the

impact of spectre and meltdown. In *2018 IEEE High Performance Extreme Computing Conference, HPEC 2018, Waltham, MA, USA, September 25-27, 2018*, pages 1–5. IEEE, 2018. doi: 10.1109/HPEC.2018.8547554. URL https://doi.org/10.1109/HPEC.2018.8547554. 5

[89] Xiaojia Rao, Aïna Linn Georges, Maxime Legoupil, Conrad Watt, Jean Pichon-Pharabod, Philippa Gardner, and Lars Birkedal. Iris-wasm: Robust and modular verification of webassembly programs. *Proc. ACM Program. Lang.*, 7(PLDI): 1096–1120, 2023. doi: 10.1145/3591265. URL https://doi.org/10.1145/3591265. 19, 74, 75, 86, 87, 95, 102, 121, 126, 129, 130, 131, 139, 140, 141, 144

[90] Xiaojia Rao, Aïna Linn Georges, Maxime Legoupil, Conrad Watt, Jean Pichon-Pharabod, Philippa Gardner, and Lars Birkedal. Iris-wasm: Robust and modular verification of webassembly programs (artefact), 2023. URL https://doi.org/10.5281/zenodo.7808708. 54

[91] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 55–74. IEEE Computer Society, 2002. doi: 10.1109/LICS.2002.1029817. URL https://doi.org/10.1109/LICS.2002.1029817. 7

[92] Andreas Rossberg. Webassembly with garbage collection. Technical report, . URL https://webassembly.github.io/gc/. 4

[93] Andreas Rossberg. Tail call optimisation proposal for webassembly. Technical report, . URL https://github.com/WebAssembly/tail-call/blob/master/proposals/tail-call/Overview.md. 4

[94] Andreas Rossberg. Webassembly (release 1.0). https://webassembly.github.io/spec/, 2019. Accessed 2020-01-01. 120

[95] Andreas Rossberg. WebAssembly core specification W3C recommendation. Technical report, W3C, December 2019. URL https://www.w3.org/TR/wasm-core-1/. 4, 5, 26, 53, 77

[96] Andreas Rossberg. Issue #1359: Typed continuations to model stacks, July 2020. URL https://github.com/WebAssembly/design/issues/1359. 16, 120

[97] Andreas Rossberg. Multiple memories proposal for webassembly. Technical report, 2022. URL https://github.com/WebAssembly/multi-memory/blob/main/proposals/multi-memory/Overview.md. 4, 17

[98] Andreas Rossberg. Webassembly (release 2.0). https://webassembly.github.io/spec/, 2023. Accessed 2023-20-02. 120, 130

[99] Andreas Rossberg. WebAssembly specification release 2.0 (draft 2024-04-02). Technical report, WebAssembly Community Group, April 2024. URL https://webassembly.github.io/spec/core/. 4, 5

[100] Andreas Rossberg. Webassembly specification release 2.0 + tail calls + function references + gc (draft 2024-03-19). Technical report, March 2024. URL https://webassembly.github.io/gc/core/syntax/types.html. 106

[101] Andreas Rossberg. Spectec has been adopted, March 2025. URL https://webassembly.org/news/2025-03-27-spectec/. 159

[102] Andreas Rossberg, Daan Leijen, Daniel Hillerström, KC Sivaramakrishnan, Matija Pretnar Sam Lindley, and Stephen Dolan. Stacks and continuations for wasm — idea sketch, February 2020. URL https://github.com/WebAssembly/meetings/blob/master/main/2020/presentations/2020-02-rossberg-continuations.pdf. Accessed Jul 4 2020. 16, 120

[103] Michael Sammler, Simon Spies, Youngju Song, Emanuele D'Osualdo, Robbert Krebbers, Deepak Garg, and Derek Dreyer. Dimsum: A decentralized approach to multi-language semantics and verification. *Proc. ACM Program. Lang.*, 7 (POPL):775–805, 2023. doi: 10.1145/3571220. URL https://doi.org/10.1145/3571220. 7, 53

[104] Fred B. Schneider, J. Gregory Morrisett, and Robert Harper. A language-based approach to security. In Reinhard Wilhelm, editor, *Informatics - 10 Years Back. 10 Years Ahead*, volume 2000 of *Lecture Notes in Computer Science*, pages 86–101. Springer, 2001. doi: 10.1007/3-540-44577-3_6. URL https://doi.org/10.1007/3-540-44577-3_6. 13

[105] Jan Schwinghammer, Lars Birkedal, Bernhard Reus, and Hongseok Yang. Nested hoare triples and frame rules for higher-order store. In Erich Grädel and Reinhard Kahle, editors, *Computer Science Logic, 23rd international Workshop, CSL 2009, 18th Annual Conference of the EACSL, Coimbra, Portugal, September 7-11, 2009. Proceedings*, volume 5771 of *Lecture Notes in Computer Science*, pages 440–454. Springer, 2009. doi: 10.1007/978-3-642-04027-6_32. URL https://doi.org/10.1007/978-3-642-04027-6_32. 122

[106] Remy Seassau, Irene Yoon, Jean-Marie Madiot, and François Pottier. Osiris: Towards formal semantics and reasoning for ocaml, February 2025. 17, 159

[107] Remy Seassau, Irene Yoon, Jean-Marie Madiot, and François Pottier. Formal semantics & program logics for a fragment of ocaml. In *ICFP*, 2025. 17, 159

[108] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Tom Ridge, Susmit Sarkar, and Rok Strnisa. Ott: effective tool support for the

working semanticist. In Ralf Hinze and Norman Ramsey, editors, *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*, pages 1–12. ACM, 2007. doi: 10.1145/1291151.1291155. URL https://doi.org/10.1145/1291151.1291155. 106

[109] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-tso: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, 2010. doi: 10.1145/1785414.1785443. 123

[110] Hovav Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson, editors, *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*, pages 552–561. ACM, 2007. doi: 10.1145/1315245.1315313. URL https://doi.org/10.1145/1315245.1315313. 13

[111] Robbert Gurdeep Singh and Christophe Scholliers. Warduino: a dynamic webassembly virtual machine for programming microcontrollers. In Antony L. Hosking and Irene Finocchi, editors, *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes, MPLR 2019, Athens, Greece, October 21-22, 2019*, pages 27–36. ACM, 2019. doi: 10.1145/3357390.3361029. URL https://doi.org/10.1145/3357390.3361029. 3

[112] K. C. Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. Retrofitting effect handlers onto ocaml. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 206–221. ACM, 2021. doi: 10.1145/3453483.3454039. URL https://doi.org/10.1145/3453483.3454039. 16

[113] Lau Skorstengaard. An introduction to logical relations. *CoRR*, abs/1907.11133, 2019. URL http://arxiv.org/abs/1907.11133. 8

[114] Lau Skorstengaard. *Formal Reasoning about Capability Machines*. PhD thesis, Aarhus University, 2019. 75, 86

[115] Lau Skorstengaard. *Formal Reasoning about Capability Machines*. PhD thesis, Aarhus University, 2019. 14

[116] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. Reasoning about a machine with local capabilities - provably safe stack and return pointer management. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European*

Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, pages 475–501, 2018. doi: 10.1007/978-3-319-89884-1_17. URL https://doi.org/10.1007/978-3-319-89884-1_17. 75, 86

[117] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. Stktokens: Enforcing well-bracketed control flow and stack encapsulation using linear capabilities. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. doi: 10.1145/3290332. URL https://doi.org/10.1145/3290332.

[118] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. Reasoning about a Machine with Local Capabilities: Provably Safe Stack and Return Pointer Management. *ACM Transactions on Programming Languages and Systems*, 42 (1):5:1–5:53, December 2019. ISSN 0164-0925. doi: 10.1145/3363519. 75, 86

[119] Thomas Solodkov and Brayton Noll. The state of webassembly 2023, 2023. URL https://www.cncf.io/reports/the-state-of-webassembly-2023/. 3

[120] David Swasey, Deepak Garg, and Derek Dreyer. Robust and compositional verification of object capability patterns. *Proc. ACM Program. Lang.*, 1(OOPSLA): 89:1–89:26, 2017. doi: 10.1145/3133913. URL https://doi.org/10.1145/3133913. 49, 75, 98

[121] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 48–62, 2013. doi: 10.1109/SP.2013.13. URL https://doi.org/10.1109/SP.2013.13. 13

[122] The Bytecode Alliance. Component model design and specification (github repository), 2023. URL https://github.com/WebAssembly/component-model. 17, 74, 106

[123] The Bytecode Alliance. The WebAssembly component model, 2023. URL https://component-model.bytecodealliance.org/. 74, 106

[124] Gavin Thomas. A proactive approach to more secure code, 2019. URL https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/. 12

[125] Amin Timany and Lars Birkedal. Mechanized relational verification of concurrent programs with continuations. *Proc. ACM Program. Lang.*, 3(ICFP), July 2019. doi: 10.1145/3341709. URL https://doi.org/10.1145/3341709. 16, 121, 142

[126] Amin Timany, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. A logical approach to type soundness. 2022. URL https://cs.au.dk/~timany/publications/files/2022-submitted-logical-type-soundness.pdf. 9, 95

[127] Thomas Van Strydonck, Frank Piessens, and Dominique Devriese. Linear capabilities for fully abstract compilation of separation-logic-verified code. *Proc. ACM Program. Lang.*, ICFP(ICFP), jul 2019. doi: 10.1145/3341688. URL https://doi.org/10.1145/3341688. 75

[128] Andew Wagner, Zachary Eisbach, and Amal Ahmed. An adjoint separation logic for the wasm call stack. presentation at Discussions at Dagstuhl Seminar 25241: Utilising and Scaling the WebAssembly Semantics, June 2025. URL https://www.andrewwagner.io/assets/slides/adj-wasm-dagstuhl.pdf. 160

[129] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *IEEE Symposium on Security and Privacy*, pages 20–37, 2015. doi: 10.1109/SP.2015.9. 14, 52

[130] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, Graeme Barnes, David Chisnall, Jessica Clarke, Brooks Davis, Lee Eisen, Nathaniel Wesley Filardo, Franz A. Fuchs, Richard Grisenthwaite, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alexander Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 9). Technical Report UCAM-CL-TR-987, University of Cambridge, Computer Laboratory, September 2023. URL https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-987.pdf. 83

[131] Conrad Watt. Mechanising and verifying the webassembly specification. In June Andronick and Amy P. Felty, editors, *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*, pages 53–65. ACM, 2018. doi: 10.1145/3167082. URL https://doi.org/10.1145/3167082. 7

[132] Conrad Watt, Petar Maksimovic, Neelakantan R. Krishnaswami, and Philippa Gardner. A program logic for first-order encapsulated webassembly. In Alastair F. Donaldson, editor, *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom*, volume 134 of *LIPIcs*, pages 9:1–9:30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi: 10.4230/LIPIcs.ECOOP.2019.9. URL https://doi.org/10.4230/LIPIcs.ECOOP.2019.9. 7, 26, 33, 42, 52, 53

[133] Conrad Watt, Xiaojia Rao, Jean Pichon-Pharabod, Martin Bodin, and Philippa Gardner. Two mechanisations of webassembly 1.0. In Marieke Huisman, Corina S. Pasareanu, and Naijun Zhan, editors, *Formal Methods - 24th International Symposium, FM 2021, Virtual Event, November 20-26,*

2021, Proceedings, volume 13047 of *Lecture Notes in Computer Science*, pages 61–79. Springer, 2021. doi: 10.1007/978-3-030-90870-6_4. URL https://doi.org/10.1007/978-3-030-90870-6_4. 7, 26, 40, 47, 48, 53, 75, 126

[134] M. V. Wilkes and R. M. Needham. *The Cambridge CAP Computer and Its Operating System*. Elsevier, January 1979. URL https://www.microsoft.com/en-us/research/publication/the-cambridge-cap-computer-and-its-operating-system/. 74

[135] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert M. Norton, and Michael Roe. The CHERI capability model: Revisiting RISC in an age of risk. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*, pages 457–468. IEEE Computer Society, 2014. doi: 10.1109/ISCA.2014.6853201. URL https://doi.org/10.1109/ISCA.2014.6853201. 74

[136] Jonathan Woodruff, Paul Metzger, Robert N. M. Watson, Brooks Davis, Wes Filardo, Jessica Clarke, and John Baldwin. SOSP 2023 CHERI exercises, 2023. URL https://www.cl.cam.ac.uk/~pffm2/sosp2023_cheri_tutorial/cover/README.html. 76

[137] Ningning Xie and Daan Leijen. Generalized evidence passing for effect handlers: efficient compilation of effect handlers to C. *Proc. ACM Program. Lang.*, 5(ICFP):1–30, 2021. doi: 10.1145/3473576. URL https://doi.org/10.1145/3473576. 16

[138] Jeremy Yallop. A collaborative bibliography of work related to the theory and practice of computational effects, 2023. URL https://github.com/yallop/effects-bibliography. 16

[139] Dongjun Youn, Wonho Shin, Jaehyun Lee, Sukyoung Ryu, Joachim Breitner, Philippa Gardner, Sam Lindley, Matija Pretnar, Xiaojia Rao, Conrad Watt, and Andreas Rossberg. Bringing the webassembly standard up to speed with spectec. *Proc. ACM Program. Lang.*, 8(PLDI):1559–1584, 2024. doi: 10.1145/3656440. URL https://doi.org/10.1145/3656440. 7, 159

[140] Dachuan Yu and Zhong Shao. Verification of safety properties for concurrent assembly code. In Chris Okasaki and Kathleen Fisher, editors, *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming, ICFP 2004, Snow Bird, UT, USA, September 19-21, 2004*, pages 175–188. ACM, 2004. doi: 10.1145/1016850.1016875. URL https://doi.org/10.1145/1016850.1016875. 6, 52

[141] Dachuan Yu, Nadeem Abdul Hamid, and Zhong Shao. Building certified libraries for PCC: dynamic storage allocation. In Pierpaolo Degano, editor,

Programming Languages and Systems, 12th European Symposium on Programming, ESOP 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings, volume 2618 of *Lecture Notes in Computer Science*, pages 363–379. Springer, 2003. doi: 10.1007/3-540-36575-3_25. URL https://doi.org/10.1007/3-540-36575-3_25. 6, 52

[142] Alon Zakai. Pause and resume webassembly with binaryen's asyncify. https://kripken.github.io/blog/wasm/2019/07/16/asyncify.html, July 2019. 120

[143] Vadim Zaliva, Kayvan Memarian, Ricardo Almeida, Jessica Clarke, Brooks Davis, Alex Richardson, David Chisnall, Brian Campbell, Ian Stark, Robert N. M. Watson, and Peter Sewell. Formal mechanised semantics of CHERI C: Capabilities, provenance, and undefined behaviour. April 2024. URL http://www.cl.cam.ac.uk/users/pes20/asplos24spring-paper110.pdf. 74, 106