An Indexed Type System for Faster and Safer WebAssembly

by

Adam T. Geller

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Computer Science)

The University Of British Columbia
(Vancouver)

August 2020

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

An Indexed Type System for Faster and Safer WebAssembly

submitted by **Adam T. Geller** in partial fulfillment of the requirements for the degree of **Master of Science** in **Computer Science**.

Examining Committee:

William J. Bowman, Computer Science
Co-supervisor

Ivan Beschastnikh, Computer Science
Co-supervisor

Ronald Garcia, Computer Science
Additional Examiner

Abstract

Downloading and executing untrusted code requires careful considerations to ensure that it is safe, but also something that happens often on the internet. Therefore, untrusted code often requires run-time checks to ensure safety during execution. These checks compromise performance and may be unnecessary. We present the Wasm-prechk language, an assembly language based on WebAssembly that is intended to ensure the same level of safety as WebAssembly while justifying the static elimination of run-time checks.

Lay Summary

Downloading and running code from unknown sources is unsafe, but also quite common. Ill-behaved or erroneous code can expose your data or crash your browser. Unfortunately, mitigating these risks often slows down programs. WebAssembly is a language supported by many browsers and designed to be both fast and safe, but the language still requires potentially unnecessary safety check which make websites slower and less efficient.

We have designed a programming language based on WebAssembly, called Wasm-prechk, that is meant to be as safe as WebAssembly without requiring as many safety check. Wasm-prechk uses a more expressive type system then WebAssembly to help the computer reason about when this extra work may be unnecessary. We contribute the design of Wasm-prechk, as well as a reference implementation, and prove that Wasm-prechk is in fact as safe as WebAssembly.

Preface

This dissertation is original, unpublished, independent work by the author, Adam T. Geller, except for Chapter 4 and Appendix B. The subject reduction proof in Chapter 4 and Appendix B, and formulation of administrative typing rules in Chapter 4 was joint work between Adam. T. Geller and Justin Frank.

Table of Contents

List of Figures

Acknowledgements

This section proceeds through my life in reverse chronological order.

University of British Columbia

First and foremost I would like to thank my wonderful advisors, who made this whole process much easier and significantly contributed to my edification.

- William Bowman - Despite being a new professor, William has been a wonderful advisor and provided a large amount of guidance when I was struggling. William has been a fantastic resource for all of the work here.

- Ivan Beschastnikh - Ivan helped me navigate through the system when I was a wide-eyed first-year and learn from various mistakes I made along the way. Ivan was a great source of wisdom and gave me the freedom to pursue the topics I found most interesting.

I would also like to thank various professors for providing guidance and advice, some of whom are listed below.

- Ron Garcia - Ron is definitely one of the nicest people I have met and has forgotten way more about PL than I have ever learned. He also provided great feedback on this thesis (part of which led to much clearer definitions).

- Alex Summers - Alex spent time to give me helpful feedback that greatly improved this thesis.

- Margo Seltzer - Margo is a kickass professor who is intelligent and cares a lot about students.

- Alan Hu - Alan has provided me with many nuggets of wisdom.

Thank you to Justin Frank, for helping to prove type safety and develop the reference implementation.

Finally, all of my fellow graduate students and lab mates who have become my friends and helped me in various ways. There are way too many awesome people to thank them all, but I include some notable names below.

- Anny Gakhokidze, Puneet Mehrotra, Nico Ritschel, Chris Chen, Felipe Bañados Schwerter, Paulette Koronkevich, Aarti Kashyap, Clement Fung, Vaastav Anand, and countless others.

My Parents

Last but certainly not least, I'd like to thank my awesome parents for everything they have done and continue to do for me. I'm not sure one could ask for better parents than I have been given (even if I did not ask for them). They have both always been there for me, especially my mom who spent a huge amount of effort and time taking care of every issue that came up in my early years, as well as homeschooling me. My Dad has always been my role model and is the smartest person I know. He tried to introduce me to set theory when I was around 4 and calculus when I was about 13 (the first one stuck pretty well, the second one not so much).

Dedications

This is dedicated to Tabby, Koko, and Socrates. These have been my best (feline) friends who have always helped me maintain my mental health.

Chapter 1

Introduction

1.1 Unsafe Code

Browsers and the Internet-of-Things (IoT) require running untrusted code (*i.e.,* unknown code from an unknown source that could do anything), that may have been downloaded from anywhere. It is crucial to ensure the safety of the code being executed in these contexts. Otherwise, one website may crash your entire browser/IoT device, read secret data in your browser/IoT device, or attempt to take over control browser/IoT device. Many such exploits have originated due to unsafe code being downloaded and run. Typically, *sandboxing* and/or *dynamic safety checks* are used to ensure the safety of untrusted code.

Sandboxing involves placing untrusted code into a secure environment to contain the damage caused by unsafe behavior [3]. For example, Google Chrome places the running scripts for each website in separate processes so that unsafe code cannot access the address space of other websites or the browser [7]. This prevents websites from crashing, stealing data, or taking control of other websites and the browser. However, sandboxing in this way requires more run time resources than running scripts in one process, as processes require overhead in most OSes.

Dynamic safety checks are run time checks that catch any attempted unsafe operations. For example, WebAssembly (Wasm) is a low-level language

designed to be both safe and fast to execute in place of JavaScript for performance-critical applications in browsers. While Wasm is type safe, memory safe, and its semantics enforce the separation of control flow and data (which together work like sandboxing to, *e.g.,* prevent websites from crashing each other), it still relies on dynamic checks to ensure these properties at run time. These dynamic checks potentially slow down programs by introducing unnecessary instructions to perform the checks.

We have designed an extension to Wasm, called Wasm-prechk, that adds new instructions that do not require dynamic safety checks. However, under the existing Wasm model the new Wasm-prechk instructions have potentially unsafe semantics, as they require stronger static guarantees than Wasm can provide to ensure the same level of safety as Wasm. These instructions are likely to be faster than their Wasm counterparts because they do not require the addition of instructions by the compiler/interpreter to perform checks. To provide these additional static guarantees, we equip Wasm-prechk with a more advanced type system than Wasm.

1.2 Type Systems

Types systems are useful for reasoning about programs. They can be used to reason about the behavior of programs, usually in the form of safety guarantees. For example, type safety is the property that a well-typed program will never become *stuck*; that is, an expression will always be a well-formed irreducible value, reduce to an error, or reduce to another well-formed expression. The safety guarantees of type systems provide a degree of trust in programs, as a well-typed program implicitly contains a checkable proof that it will only exhibit well-defined behavior (which, in the case of Wasm, has several safety guarantees as discussed above).

Generally, low-level languages are either untyped or have simple type systems that provide minimal guarantees. More expressive type systems can encode richer invariants, enabling ruling out more bad behaviors with static checks alone. Generally, such type systems are attached to high-level languages, where explicit abstractions make it easier to reason about programs

compared to low-level languages. Conversely, using expressive type systems in low-level languages often requires reasoning about program state and unstructured control flow (*e.g.,* `goto`), which introduces more complexity into the type system. However, prior work has attached expressive type systems that permit rich correctness guarantees to simple low-level languages.

1.3 Related Work

Using type information to improve compiler optimizations is not a new idea. Tarditi et al. [8] used strongly typed intermediate languages (TIL) to improve optimizations of SML code. Compiling SML involves many translations among intermediate languages, and by preserving type information across those translations Tarditi et al. [8] were able to safely perform additional compiler operations. Using TIL in the compilation of programs led to up to 50% faster programs. TIL focuses on compiler optimizations and eventually translates into untyped languages and finally runnable assembly, so the ensured guarantees of the type system are lost along the way (they may still be present, but it is no longer possible to statically check them).

Proof-carrying code (PCC) was introduced by Necula [6]. PCC attached explicit proofs that low-level code satisfies some safety properties. The proof can then be checked to ensure the safety of the code before it is run. While typed intermediate languages require types to be considered as part of the language, PCC adds a separate logical framework on top. Thus, PCC can be used with any language, and requires no support from the language. However, because PCC has no support from the language, it has a higher proof burden than using types inside a language.

Necula [6] provides a detailed example of invariants for extensions to TIL to ensure type safety of compiled code. The example uses the Edinburgh Logical Framework (LF) to encode the proof. A type safety proof of a LF program is a proof of correctness (with respect to some specification). Encoded proofs may be quite large, requiring extra time to transmit and check compared to type annotations and typechecking.

Morrisett et al. [5] showed how types could provably be preserved during

five different compilation passes to get from System F all the way down to a typed assembly language (TAL). The purpose of TAL was focused on safety. Although Morrisett et al. [5] argued that the type-preserving compilation passes would permit similar optimizations to TIL, they didn't include further optimizations based on TAL. However, Morrisett et al. [5] did argue that the guarantees of TAL were sufficient to allow untrusted code to be safely executed.

Xi and Harper [9] created a much more expressive type system for an assembly language which had the potential to allow more compiler optimizations. Their language, a dependently typed assembly language (DTAL), used a limited dependent type system, which enabled safely removing some run time checks, including array bounds checks. The goal of DTAL, similar to TAL, was to support type-preserving compilation from a high-level language for both optimizations and safety. DTAL intended to support type-preserving compilation from Dependent ML as well as SML.

After almost two decades of JavaScript being the dominant language in browsers, it was decided that an alternative was necessary for performance-critical code. The alternative that was jointly created by the major browser developers was WebAssembly (Wasm) [4]. Wasm is a stack-based assembly language with structured control flow. It is designed to be safe as well as performant, with a small binary footprint. The Wasm type system is simple, only encoding primitive types, but strong enough to ensure certain safety properties (*i.e.,* that arbitrary code cannot be executed). Memory safety in Wasm is enforced using run time checks. Wasm is supported by most major browsers, and can work well for IoT devices due to its portability and safety.

1.4 Contributions

We want to use types to improve performance while ensuring safety in real-world low-level programs. While prior work has used expressive type systems for low-level languages, we want to show the feasibility of doing this with a language commonly used in contexts that are both performance-critical and untrusted. Towards that goal, we introduce Wasm-prechk, an extension of

the WebAssembly (Wasm) language.

We chose Wasm to build on because it is used in browsers and IoT devices. Both browsers and IoT device require strong safety guarantees (such as preventing programs from crashing the whole browser/IoT device) since they download and execute code from unknown sources. Further, the use case for Wasm is often performance-critical applications such as in-browser games and cryptographic algorithms.

Wasm-prechk introduces new versions of Wasm instructions which require no dynamic checks, but also require stronger type-level safety guarantees than their Wasm counterparts (section 3.2). To facilitate type-checking these new instructions, Wasm-prechk uses an indexed type system which is able to encode linear constraints on program variables and therefore ensure safety properties of the new Wasm-prechk instructions (section 3.3). We ensure that Wasm-prechk is as safe as Wasm by providing a type safety proof of the Wasm-prechk indexed type system (section 4.3). Together, these additions mean that Wasm-prechk is as safe as Wasm while potentially improving performance by moving run-time checks to compile time.

We show that Wasm-prechk is backwards compatible with Wasm by showing that Wasm programs can be turned into Wasm-prechk programs (subsection 4.2.1). This means that there is no additional proof burden on the programmer to use Wasm-prechk. We also show that we can go the other way by turning Wasm-prechk programs into well-typed Wasm programs (subsection 4.2.2).

1.5 Thesis Statement

> An indexed type system can be used in an existing low-level language to reduce the number of dynamic checks required, without sacrificing safety and security guarantees or increasing the programmer's proof burden.

Chapter 2

Background: Wasm

Here we present an overview of Wasm so readers have some familiarity with it for when we present Wasm-prechk. We do not cover the entirety of the Wasm language as presented in the 2017 paper [4], but rather present selected important facets of the syntax, semantics, and type system. It is recommended that the reader first skim this chapter to understand the basics and then refer back while reading chapter 3 and chapter 4.

2.1 Wasm Syntax

Figure 2.1 shows the types of Wasm. Primitive Wasm types, represented as t, include 32- and 64-bit floats and integers. Packed types, tp, include 8-, 16-, and 32- bit integers, are used in memory operations to load/store a smaller payload (*e.g.,* i8 loads/stores just one byte). Wasm is a stack-based language, so the type of an instruction in Wasm consists of a precondition and postcondition on the shape of the stack, which is what a Wasm function type tf is encoding. This can be viewed as though instructions *consume* certain values from the stack and then *produce* values to be pushed on the stack. Thus, function types, tf, are just syntax used in certain instructions, function declarations, and the Wasm typing judgment, not function types in the traditional sense. Lastly, global types consist of a primitive type t and an optional mutable flag (the $^?$ form is explained more below).

$$
\begin{array}{rcl}
t & ::= & \mathsf{i32} \mid \mathsf{i64} \mid \mathsf{f32} \mid \mathsf{f64} \\
tp & ::= & \mathsf{i8} \mid \mathsf{i16} \mid \mathsf{i32} \\
tf & ::= & t^* \to t^* \\
tg & ::= & \text{mut}^? \; t
\end{array}
$$

Figure 2.1: Wasm Types

The Wasm syntax uses the Kleene star within its BNF (*e.g.,* t^*) to denote possibly empty sequences. For example, t^* matches ϵ (the empty sequence, which is an empty sequence of anything), i32 i32, and i32 i64 i32. Instructions, represented by the metavariable e, are usually grouped into sequences e^* which are possibly empty ϵ. As a further point on metavariables, e_1 and e_2, both instruction metavariables, may happen to be the same instruction, or not, we do not know. Similarly, e_1^* and e_2^* refer to different sequence metavariables that may or may not be the same; we can make no assumptions about them.

We can use different annotations in place of the Kleene star to add additional information. The Kleene star may be replaced with an exact value n when we know that the sequence has length n (*e.g.,* a sequence of 3 types be phrased as t^3). We can also use a question mark to represent either an empty sequence (ϵ), or a sequence with exactly one item (*e.g.,* $v^? = v' \vee v^? = \epsilon$).

There is no requirement that a sequence of non-terminals, e_1^*, be made up of entirely the same pattern, unless it is explicitly written out as in $(t.\mathsf{const}\ c)^*$. For example, e_1^* matches $(t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ (t.binop)$. Further, we may separate out subsequences: from $(t.\mathsf{const}\ c)^*$ we may separate out t^* and c^* to refer to the sequences of types and constant values respectively.

With this notation in mind, we can now look over the Wasm instructions in Figure 2.2 (we will discuss the instructions in section 2.2). Syntax written in a blue sans serif font denotes a keyword, while text written in *italics* represents a metavariable. Throughout the Wasm syntax there are many metavariables used to represent natural numbers: n and m are usually used for the table and memory sizes, i and j are often used as indexes (*e.g.,* to reference a local variable), o and $align$ are used within memory operations (we replace a with $align$ for clarity and since we use a elsewhere), and lastly

$$
\begin{array}{rcl}
unop_{iN} & ::= & \mathsf{clz} \mid \mathsf{ctz} \mid \mathsf{popcnt} \\
testop_{iN} & ::= & \mathsf{eqz} \\
binop_{iN} & ::= & \mathsf{add} \mid \mathsf{sub} \mid \mathsf{shl} \mid \mathsf{or} \mid \ldots \\
relop_{iN} & ::= & \mathsf{eq} \mid \mathsf{ne} \mid \mathsf{gt} \mid \mathsf{ge} \mid \ldots \\
cvtop & ::= & \mathsf{convert} \mid \mathsf{reinterpret} \\
e & ::= & \mathsf{unreachable} \mid \mathsf{nop} \mid \mathsf{drop} \mid \mathsf{select} \mid \\
 & & \mathsf{block}\ tf\ e^*\ \mathsf{end} \mid \mathsf{loop}\ tf\ e^*\ \mathsf{end} \mid \mathsf{if}\ tf\ e^*\ \mathsf{else}\ e^*\ \mathsf{end} \mid \\
 & & \mathsf{br}\ i \mid \mathsf{br\_if}\ i \mid \mathsf{br\_table}\ i^+ \mid \mathsf{return} \mid \mathsf{call}\ i \mid \mathsf{call\_indirect}\ tf \mid \\
 & & \mathsf{get\_local}\ i \mid \mathsf{set\_local}\ i \mid \mathsf{tee\_local}\ i \mid \mathsf{get\_global}\ i \mid \\
 & & \mathsf{set\_global}\ i \mid t.\mathsf{load}\ (tp\_sx)^?\ align\ o \mid t.\mathsf{store}\ tp^?\ align\ o \mid \\
 & & \mathsf{current\_memory} \mid \mathsf{grow\_memory} \mid t.\mathsf{const}\ c \mid \\
 & & t.unop_t \mid t.binop_t \mid t.testop_t \mid t.relop_t \mid t.cvtop\ t\_sx^? \mid \ldots
\end{array}
$$

Figure 2.2: Wasm Instructions

c is used as a constant metavariable (which could also be a float). iN is used to annotate operations that support integers, and fN is used to annotate operations that support floats.

Some instructions, such as loop tf e^* end include a sequence of instructions e^*. We refer to such instructions as block instructions, since they define control flow blocks for the instructions inside (not to be confused with the block instruction, which is a block instruction). In a block instruction, you will see one or more instruction sequences e^* as part of the syntax before end, we refer to this as the body. Further, many block instructions also include an explicit type annotation tf declaring their precondition and postcondition.

Wasm has modules that include functions (f), global variables ($glob$), an optional function table (tab), and an optional linear memory chunk (mem), as seen in Figure 2.3. Functions, globals, the table, and memory can be imported, using import "$name_1$" "$name_2$", which imports $name_2$ from the file $name_1$. Similarly, they can also be exported under any number of names using export "$name$".

Functions include a list of local variable declarations to use within the body (a sequence of instructions). Additionally, function arguments are accessible as local variables within the body of functions. Global variables may be mutable (although, exported global variables, which are accessible in

(imports)	im	::=	import "$name$" "$name$"
(exports)	ex	::=	export "$name$"
(functions)	f	::=	ex^* func tf local t^* e^* | ex^* func tf im
(globals)	$glob$	::=	ex^* global tg e^* | ex^* global tg im
(table)	tab	::=	ex^* table n i^* | ex^* table n im
(memory)	mem	::=	ex^* memory n | ex^* memory n im
(modules)	$module$	::=	module f^* $glob^*$ $tab^?$ $mem^?$

Figure 2.3: Wasm Module Definitions

other modules, cannot be mutable, as we will see later), and are initialized via a sequence of instructions. Function tables store references to functions that can be called using indirect function calls; they are used to more safely represent function pointers. Indirect function calls take an index and use it to lookup a function in the function table and call it. They must supply a function type annotation, tf that gets checked against the function that ends up being called at run-time. Linear memory, mem, is a continuous chunk of memory. Memory load and store operations operate within the linear memory chunk.

2.2 Wasm Dynamic Semantics

Wasm is a stack-based assembly language specified using reduction semantics [1]. Before we introduce the Wasm semantics, we first must introduce some administrative structures and instructions that are used in the reduction relation to keep track of information. Administrative instructions are not part of the surface syntax of a language (*e.g.,* you cannot put a local block in a Wasm-prechk program), and can only appear as an intermediate term during reduction. Figure 2.4 shows the new administrative instructions and run-time structures.

The runtime store, s, includes runtime instances for every module ($inst^*$), as well as all of the tables ($tabinst^*$), and memory chunks ($meminst^*$). In

[1] For those unfamiliar with reduction semantics, I highly recommended these notes by Ron Garcia: `https://www.cs.ubc.ca/~rxg/cpsc509/05-reduction.pdf`

(closures)	cl	::=	$\{\text{inst } i, \text{func } f\}$
(bytes)	b	::=	$0x00, 0x01, ..., 0x\texttt{ff}$
(table instances)	$tabinst$	::=	cl^*
(memory instances)	$meminst$	::=	b^*
(modules instances)	$inst$	::=	$\{\text{func } cl^*, \text{glob } v^*, \text{tab } i^?, \text{mem } i^?\}$
(stores)	s	::=	$\{\text{inst } inst^*, \text{tab } tabinst^*,$ $\text{mem } meminst^*\}$
(values)	v	::=	$t.\mathsf{const}\ c$
(admin. instrs.)	e	::=	$... \mid \mathsf{trap} \mid \mathsf{call}\ cl \mid \mathsf{label}_n\{e^*\}\ e^*\ \mathsf{end} \mid$ $\mathsf{local}_n\{i;\ v^*\}\ e^*\ \mathsf{end}$
(reduction contexts)	L^0	::=	$v^* \square e^*$
	L^{k+1}	::=	$v^*\ \mathsf{label}_n\{e^*\}\ L^k\ \mathsf{end}\ e^*$

Figure 2.4: Wasm Administrative Instructions and Run-Time Structures

other words, s includes an instantiation of every module. Module instances, $inst$, represent Wasm modules after linking. They refer to their table and memory (if they have either), by indexing into the list of runtime instances of tables and memory chunks in the store s. A table instance $tabinst$ contains a list of closures that can be called. b represents a byte. A memory instance $meminst$ is a sequence of bytes representing a contiguous memory chunk. Wasm closures, cl, intuitively represents a function closed under linking. Closures include the module instance that the function is defined in, as well as the function definition (which cannot be an import) with any exports erased.

There are a few final notational digressions we must make before describing the reduction relation. Firstly, objects such as $s = \{\text{inst } inst^*, \ldots\}$ can be dereferenced using their keywords (*e.g.,* "inst"). For example, $s_{\text{inst}} = inst^*$ given the above definition of s. Secondly, we can index into a sequence to get a specific element (*e.g.,* $inst^*(i)$ returns the ith $inst$ in $inst^*$). Lastly, Wasm uses several shorthands to get information out of module instances in s: $s_{\text{func}}(i, j) = s_{\text{inst}}(i)_{\text{func}}(j)$. Essentially, this allows us to implicitly dereference the ith module instance to get the jth function inside of the instance. This shorthand is used similarly for glob, tab, and mem.

Constant instructions $t.\mathsf{const}\ c$ represent values, and are denoted by the

metavariable v, when they should be interpreted that way. They produce a constant value (known statically). This leads to a particular representation of the stack, as discussed in subsection 2.2.1. A *trap* (trap) is the Wasm term for a run-time error. call cl is a function call on a closure. As we will see, it is an intermediate step for performing both direct and indirect function calls.

Two types of block instructions are introduced. The first, the label block, is used in handling control flow. Specifically, they are used to handle branching. All block instructions (block, loop, and if) reduce to label blocks. Label blocks can store instructions (e^* inside the curly braces), and the annotation n is equal to the expected number of inputs to those instructions. This is explained more when we describe how branching works.

The second block instruction is the local block. A local is the result of reducing a closure call; it is used to reduce a function body within the closed environment of the closure. They introduce an environment consisting of the module instance and local variables inside which their body is reduced.

Finally, we introduce reduction contexts, L^k, where k is the nesting depth. Reduction contexts are defined using label blocks, so L^k contains k nested label blocks. As well as nested label blocks, reduction contexts contain preceding values v^* (*i.e.,* a stack), and proceeding instructions e^* that are next to be executed after the nested label block finishes reducing.

2.2.1 The Wasm Reduction Relation

The Wasm Reduction Relation works on *configurations* that include the store s, local variables (represented as a sequence of values v^*), and the instruction sequence e^*. Reduction is relative the the current module index i, which is used to know which module instance in the store to look at when dereferencing the store. The store, local variables, and module index are omitted when not used. We present all the reduction rules below.

$$\boxed{s;\, v^*;\, e^* \hookrightarrow_i s';\, v'^*;\, e'^*}$$

Instructions are reduced in place by decomposing the program using

reduction contexts. Intuitively, we pull out the next instruction to execute, reduce it, and push the result on top of the stack. The "stack" is just the sequence of values (*i.e.,* constant instructions) preceding the first reducible instruction. When an instruction reduces to a value, that value becomes the new top of the stack and the next instruction is reduced. *This method of decomposing ensures that all of the instructions preceding the instruction currently being reduced have already been reduced to values.*

Binary and relation operations consume two values from the stack and either push back onto the stack the specified operation applied to those values, or trap if the operation on the values is not defined (in the case of dividing by zero). Test operators only consume one value, and do not trap, but are otherwise similar. The reduction rules for these operators use metafunctions (*e.g.,* $testop_t(c)$) to compute the result of applying the operator for the produced value.

The instruction unreachable causes a trap (it is similar to *e.g.,* `assert false`), nop reduces to the empty sequence, and drop consumes one value and reduces to the empty sequence (*i.e.,* it discards the value on top of the stack). select is a ternary operator (like ? : in C) that consumes three values and produces either the first or the second depending on the third value. The true/non-zero case of select returns the first value consumed ($k + 1$ is a common shorthand for a non-zero value), and the false/zero case returns the second value consumed.

$$
\begin{array}{rcl}
(t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ t.binop & \hookrightarrow & t.\mathsf{const}\ c \\
 & & \text{if } c = binop(c_1, c_2) \\
(t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ t.binop & \hookrightarrow & \mathsf{trap} \\
 & & otherwise \\
(t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ t.relop & \hookrightarrow & t.\mathsf{const}\ relop(c_1, c_2) \\
(t.\mathsf{const}\ c)\ t.testop & \hookrightarrow & \mathsf{i32}.testop_t(c) \\
\mathsf{unreachable} & \hookrightarrow & \mathsf{trap} \\
\mathsf{nop} & \hookrightarrow & \epsilon \\
v\ \mathsf{drop} & \hookrightarrow & \epsilon \\
v_1\ v_2\ (\mathsf{i32}.\mathsf{const}\ 0)\ \mathsf{select} & \hookrightarrow & v_2 \\
v_1\ v_2\ (\mathsf{i32}.\mathsf{const}\ k+1)\ \mathsf{select} & \hookrightarrow & v_1
\end{array}
$$

Block instructions define a control flow environment used by branching instructions inside which their bodies are reduced. The true case of an if block reduces to the first body inside of a block; the false case does the same but with the second body. Both block and loop reduces to label blocks. Stored instructions are only added when reducing a loop, in which case it stores the loop code so it can run the loop again. If the body of a label block is a trap or a sequence of values then the trap/values replace the block. Since decomposition happens on label blocks, we have included the inductive reduction rule, which intuitively pulls instructions out of the context, reduces them outside the context, and then plugs them back in.

$$\frac{s;\ v^*;\ e^* \hookrightarrow s';\ v'^*;\ e'^*}{s;\ v^*;\ L^k[e^*] \hookrightarrow s';\ v'^*;\ L^k[e'^*]}$$

$$\begin{array}{rcl}
L^0[\mathsf{trap}] & \hookrightarrow & \mathsf{trap} \\
v^n\ \mathsf{block}\ (t_1^n \to t_2^m)\ e^*\ \mathsf{end} & \hookrightarrow & \mathsf{label}_m\{\}v^n\ e^*\ \mathsf{end} \\
v^n\ \mathsf{loop}\ (t_1^n \to t_2^m)\ e^*\ \mathsf{end} & \hookrightarrow & \mathsf{label}_n\{\mathsf{loop}\ (t_1^n \to t_2^m)\ e^* \\
 & & \quad \mathsf{end}\} \\
 & & \quad v^n\ e^* \\
 & & \mathsf{end} \\
(\mathsf{i32.const}\ 0)\ \mathsf{if}\ tf\ e_1^*\ \mathsf{else}\ e_2^*\ \mathsf{end} & \hookrightarrow & \mathsf{block}\ tf\ e_2^*\ \mathsf{end} \\
(\mathsf{i32.const}\ k+1)\ \mathsf{if}\ tf\ e_1^*\ \mathsf{else}\ e_2^*\ \mathsf{end} & \hookrightarrow & \mathsf{block}\ tf\ e_1^*\ \mathsf{end} \\
\mathsf{label}_n\ \{e_0^*\}\ v^*\ \mathsf{end} & \hookrightarrow & v^* \\
\mathsf{label}_n\ \{e_0^*\}\ \mathsf{trap}\ \mathsf{end} & \hookrightarrow & \mathsf{trap}
\end{array}$$

Branching (br j) intuitively jumps to the $j+1$th outer control flow block (*i.e.,* a label block). More concretely, a br j inside a label block (which, you may recall, are used as control flow blocks) jumps to the surrounding label block with nesting depth $j+1$ (essentially peeling back j layers). After branching, execution continues with the values v^n consumed by the br and the stored instructions e_0^* of the $j+1$th outer label block (this is in place to support loops, as jumping to the label block introduced by loop is what causes the next iteration to be performed). Extra stack values beyond those consumed are discarded. Figure 2.5 has several examples of branching in action.

Wasm also has a conditional branch instruction. This instruction, br_if j, consumes a value and reduces to br j if the value is non-zero, otherwise it reduces to the empty sequence. Table branches, br_table , has a list of one or more numbers, i^+ that may be used for a branch. It consumes a i32 k and reduces to br with the kth number, or the last number if there is no kth number.

$\mathsf{label}_0\{\mathsf{loop}\ldots\ \mathsf{end}\}\ \mathsf{br}\ 0\ \mathsf{end}$
$\hookrightarrow\ \mathsf{loop}\ldots\ \mathsf{end}$

$\mathsf{label}_0\{\}$
  $\mathsf{label}_0\{\mathsf{loop}\ldots\ \mathsf{end}\}\ \mathsf{br}\ 1\ \mathsf{end}$
end
$\hookrightarrow\ \epsilon$

$\mathsf{label}_0\{\}$
  $\mathsf{label}_0\{\}$
    $\mathsf{label}_0\{\}\ \mathsf{br}\ 1\ \mathsf{end}$
  end
end
$\hookrightarrow\ \mathsf{label}_0\{\}\ \ \mathsf{end}$

Figure 2.5: Branching Examples

$$
\begin{array}{rcl}
\mathsf{label}_n\ \{e_0^*\}\ L^j[v^n\ \mathsf{br}\ j]\ \mathsf{end} & \hookrightarrow & v^n\ e_0^* \\
(\mathsf{i32.const}\ 0)\ \mathsf{br\_if}\ j & \hookrightarrow & \epsilon \\
(\mathsf{i32.const}\ k+1)\ \mathsf{br\_if}\ j & \hookrightarrow & \mathsf{br}\ j \\
(\mathsf{i32.const}\ k)\ \mathsf{br\_table}\ j_1^k\ j\ j_2^* & \hookrightarrow & \mathsf{br}\ j \\
(\mathsf{i32.const}\ k+n)\ \mathsf{br\_table}\ j_1^k j & \hookrightarrow & \mathsf{br}\ j
\end{array}
$$

Direct and indirect function calls are expanded in two steps. First, the associated closure is fetched either from the current module instance (for direct calls) or from the table (for indirect calls, which traps if the type of the fetched closure doesn't match the expected type). This step reduces a direct or indirect call to a $\mathsf{call}\ cl$. Then, the closure body is placed into a local block with the arguments from the stack and locals declared by the function (t^k), which are zero-initialized, being used as the local variables.

$$
\begin{array}{rcl}
s;\ \mathsf{call}\ j & \hookrightarrow_i & \mathsf{call}\ s_{\mathrm{func}}(i,j) \\
s;\ \mathsf{call\_indirect}\ j & \hookrightarrow_i & s_{\mathrm{tab}}(i,j) \\
 & & \text{if } s_{\mathrm{tab}}(i,j)_{\mathrm{code}} = (\mathsf{func}\ tf\ \mathsf{local}\ t^*\ e^*) \\
s;\ \mathsf{call\_indirect}\ j & \hookrightarrow_i & \mathsf{trap} \\
 & & \text{otherwise} \\
v^n\ (\mathsf{call}\ cl) & \hookrightarrow_i & \mathsf{local}_m\{cl_{\mathrm{inst}};\ v^n\ (t.\mathsf{const}\ 0)^k\} \\
 & & \quad \mathsf{block}\ (\epsilon \to t_2^m)\ e^* \\
 & & \quad \mathsf{end} \\
 & & \mathsf{end} \\
 & & \text{where } cl_{\mathrm{func}} = (\mathsf{func}\ tf\ \mathsf{local}\ t^*\ e^*)
\end{array}
$$

The local block has the same module index, i, as the closure, so the body of the local block is reduced within the module that the closure is defined in and thus uses the global variables, table, and memory of that module instance. This is handled by the inductive reduction rule (which has much more of a structural operational small-step semantics flavor). In general, return can be thought of as br k, where k is the context depth. A label block is added inside of the local block when expanded a function call, so at the top level of a function br 0 is essentially equivalent to return, except with an additional reduction step. Returning, somewhat similarly to branching, replaces the local block with the arguments to the return instruction, except that it skips over any label blocks. If the body of a local block is a trap or sequence of values equal to the number annotation on the local block, then that is what the local block reduces to, similar to branching (also similar to branching, any extra values on the stack are discarded).

$$\frac{s;\ v^*;\ e^* \hookrightarrow_i s';\ v'^*;\ e'^*}{s;\ v_0^*;\ \mathsf{local}_n\{i;\ v^*\}e^*\ \mathsf{end} \hookrightarrow_j s';\ v_0^*;\ \mathsf{local}_n\{i;\ v'^*\}e'^*\ \mathsf{end}}$$

$$\begin{array}{rcl} \mathsf{local}_n\{i;\ v_l^*\}v^n & \hookrightarrow & v^n \\ \mathsf{local}_n\{i;\ v_l^*\}\mathsf{trap} & \hookrightarrow & \mathsf{trap} \\ \mathsf{local}_n\{i;\ v_l^*\}L^k[v^n\mathsf{return}] & \hookrightarrow & v^n \end{array}$$

Local variables are represented as a list of values at run time. They are get/set by indexing into them, like everything else in Wasm. The same is true of global variables, except there is an extra step since they are stored in the current module instance inside the store s.

$$\begin{array}{rcl} v_1^j\ v\ v_2;\ \mathsf{get\_local}\ j & \hookrightarrow & v \\ v_1^j\ v\ v_2;\ v'\ (\mathsf{set\_local}\ j) & \hookrightarrow & v_1^j\ v'\ v_2;\ \epsilon \\ v_1^j\ v\ v_2;\ v'\ (\mathsf{tee\_local}\ j) & \hookrightarrow & v_1^j\ v'\ v_2;\ v' \\ s;\ \mathsf{get\_global}\ j & \hookrightarrow_i & s_{\mathrm{glob}}(i,j) \\ s;\ v;\ \ (\mathsf{set\_global}\ j) & \hookrightarrow_i & s';\ \epsilon \\ & & \text{where } s' = s \text{ with } glob(i,j) = v' \end{array}$$

Finally, there are the memory instructions. One can load or store a value from or to memory, get the current memory size, or try to grow the memory. $|t|$ is used to represent the size of the type (*e.g.,* $|\mathsf{i64}| = 8$ bytes). We omit two rules, one each for store and load, that include the ability to use packed types to load/store smaller values and to load signed/unsigned. There is a lot of minutiae detail, but none of it is particularly important. For example, *tp* is an optional packed type which allows storing values smaller than the normal size of the type of the value (*e.g.,* storing eight bits i8 of a thirtytwo bit integer i32). Loading from memory can optionally be signed or unsigned using sx, which represented signed or unsigned. The "alignment exponent" *align* is a mysterious variable that is not used during reduction, and is only used during typechecking without any explanation. Two metafunctions, $const_t$

and $bits_t$, are used to convert bits to values and vice versa. The key high level takeaway is that load and store will trap if the supplied index k plus the static offset o is out of bounds.

$$
\begin{array}{rcl}
s;\ (\mathsf{i32.const}\ k) & & \\
(t.\mathsf{load}\ tp\_sx\ align\ o) & \hookrightarrow_i & s;\ (t.\mathsf{const}\ const_t(b^*)) \\
 & & \text{if } s_{\text{mem}}(i, k+o, |t|) = b^* \\
s;\ (\mathsf{i32.const}\ k) & & \\
(t.\mathsf{load}\ tp\_sx\ align\ o) & \hookrightarrow_i & \mathsf{trap} \\
 & & \text{otherwise} \\
s;\ (\mathsf{i32.const}\ k)\ (t.\mathsf{const}\ c) & & \\
(t.\mathsf{store}\ tp\_sx\ align\ o) & \hookrightarrow_i & s';\ \epsilon \\
 & & \text{if } s' = s \text{ with } \text{mem}(i, k+o, |t|) = bits_t(c) \\
s;\ (\mathsf{i32.const}\ k)\ (t.\mathsf{const}\ c) & & \\
(t.\mathsf{store}\ tp\_sx\ align\ o) & \hookrightarrow_i & \mathsf{trap} \\
 & & \text{otherwise}
\end{array}
$$

$$
\begin{array}{rcl}
s;\ \mathsf{current\_memory} & \hookrightarrow_i & \mathsf{i32.const}\ |s_{\text{mem}}(i, *)|/64\text{Ki} \\
s;\ (\mathsf{i32.const}\ k) & & \\
\mathsf{grow\_memory} & \hookrightarrow_i & s';\ \mathsf{i32.const}\ |s'_{\text{mem}}(i, *)|/64\text{Ki} \\
\text{if } s' = s & \text{with} & \text{mem}(i, *) = s_{\text{mem}}(i, *)(0)^{k*64\text{Ki}} \\
s;\ (\mathsf{i32.const}\ k) & & \\
\mathsf{grow\_memory} & \hookrightarrow_i & \mathsf{i32.const}\ (-1) \\
\text{otherwise} & &
\end{array}
$$

2.3 The Wasm Type System

Instructions in Wasm are typed under a module type context C. C keeps track of various module-level types: functions, globals, the table, memory, locals, the label stack (*i.e.,* the expected types for branching instructions), and the return stack (*i.e.,* the expected type of the return instruction).

$$C ::= \{\text{func } tf^*,\ \text{global } tg^*,\ \text{table } n^?,\ \text{memory } m^?,\\ \text{local } t^*,\ \text{label } (t^*)^*,\ \text{return } (t^*)^?\}$$

Here is an example of a Wasm typing rule, a binary operation of some type t consumes two values of the given type t on the stack and produces a value of type t:

$$\frac{}{C \vdash t.binop : t\ t \to t}$$

The above example shows what a typical Wasm typing rule looks like. The type associated with the instruction $t.binop$ is a Wasm function type, which is just the precondition ($t\ t$ on the left of the $\to$) and postcondition (t on the right of the $\to$) on the stack. In the precondition, the top of the stack is the rightmost type (for example, in $t_1\ t_2\ t_3$, t_3 is the top of the stack), since that represents the value closest to the instruction getting reduced. The precondition and postcondition represent the shape of the stack before and after executing a sequence of instructions. Intuitively, they represent the "state of the world" before and after the instruction sequence is executed: they require the world to be in a certain state, and then transform it into some other state. Thus, the static Wasm typing judgement is as follows:

$$\boxed{C \vdash e^* : tf}$$

In addition to this typing judgment, Wasm also includes typing judgments for administrative instructions (which require additional type information about runtime structures, so the judgment has a different form) and a typing judgment in the form of the reduction relation for the Wasm type safety proof. Wasm also has typing judgments for modules and module-level declarations.

We reproduce and explain a few selected typing rules from Wasm using the static typing judgement. Most typing rules are for a single instruction and there are a few rules which can combine rules. The rule for typing a block, Rule Wasm-Block typechecks the body e^* under the module type context with the postcondition t_2^m appended to the label stack. This is yet

another common notational shorthand where x, y means x extended with y. The branch rule, Rule Wasm-Br, accepts any precondition, extended with the ith postcondition on the label stack (counting backwards), and returns to any postcondition. A branch will return the n values before it, so it is ok if there are more values on the stack, as they will be discarded. Execution does not proceed after branching, so the postcondition can be anything. For function calls we lookup the type of the function in the context (Rule Wasm-Call). Recall that local variables are represented by a list of values at runtime. Thus, the typing rule for set_local checks that the value consumed by set_local , which will replace the ith local in the list, has the correct type that is given by looking up the type of the ith local in the context (Rule Wasm-Set-Local).

$$\frac{}{C \vdash t.binop : t\ t \to t}\ \textsc{Wasm-Binop}$$

$$\frac{tf = t_1^n \to t_2^m \qquad C, \text{label}(t_2^m) \vdash e^* : tf}{C \vdash \mathsf{block}\ tf\ e^*\ \mathsf{end} : tf}\ \textsc{Wasm-Block}$$

$$\frac{C_{\text{label}}(i) = t^n}{C \vdash \mathsf{br}\ i : t_1^*\ t^n \to t_2^*}\ \textsc{Wasm-Br} \qquad \frac{C_{\text{func}}(i) = tf}{C \vdash \mathsf{call}\ i : tf}\ \textsc{Wasm-Call}$$

$$\frac{C_{\text{local}}(i) = t}{C \vdash \mathsf{set\_local}\ i : t \to \epsilon}\ \textsc{Wasm-Set-Local}$$

The empty instruction sequence has an empty precondition and postcondition (Rule Wasm-Empty). An instruction e_2 can be appended to a sequence of instructions e_1^* if the precondition of e_2 is the same as the postcondition of e_1^* (Rule Wasm-Composition). Then, the precondition of the full sequence $e_1^*\ e_2$ is the precondition of e_1^* and the postcondition of $e_1^*\ e_2$ is the postcondition of e_2.

$$\frac{}{C \vdash \epsilon : \epsilon \to \epsilon}\ \textsc{Wasm-Empty}$$

$$\frac{C \vdash e_1^* : t_1^* \to t_2^* \qquad C \vdash e_2 : t_2^* \to t_3^*}{C \vdash e_1^*\ e_2 : t_1^* \to t_3^*}\ \textsc{Wasm-Composition}$$

2.3.1 Stack Polymorphism

To compose together the types of many instructions, it is necessary to carry around extra type information about the rest of the stack while type-checking instructions. *Stack polymorphism* allows extending the precondition and postcondition with the same data to thread unmodified parts of the stack through a list of instructions. Intuitively, this allows you to "forget" the rest of the stack and focus only on the part being manipulated by the instruction being checked, after which point the "forgotten" part can be re-added.

For example, if the stack has the shape i64 i32 i32, then stack polymorphism allows us to ignore i64 and typecheck i32.*binop* with i32 i32. Then the stack would look like i32, at which point we add i64 back to the postcondition to get i64 i32 after executing i32.*binop*.

Chapter 3

Wasm-prechk

The goal of Wasm-prechk is to eliminate unnecessary dynamic checks. To accomplish this, it must (1) have instructions that do not require dynamic checks and (2) statically prove that the assumptions of those instructions are met. Wasm-prechk extends Wasm with new instructions that explicitly do not require dynamic checks, and an indexed type system to reason about the safety of omitting checks. Intuitively, we are replacing dynamic checks with static checks whenever possible.

Consider the example of a Wasm program with an unnecessary dynamic check in Figure 3.1. The program consists of a block instruction that consumes two arguments from the stack and produces onto the stack their quotient if the second argument is non-zero, and the first argument otherwise. In Wasm, this program would have a dynamic division-by-zero check inserted for the

```
block (i32 i32 → i32)
      (tee_local 0)
      (i32.const 1)
      (get_local 0)
      (select)
      (i32.div)
end
```

Figure 3.1: An Example of an Unnecessary Wasm check

division instruction (i32.div). However, this check would be unnecessary since the instructions preceding the division instruction ensures that the second argument is non-zero. This is guaranteed because the select instruction selects the first value (the local) if the local is non-zero, and, otherwise, selects the second value (i32.const 1).

In the above example, it is possible to check the necessary precondition of the division instruction (that the second argument is non-zero) statically, rather than dynamically. Wasm-prechk performs such static checks using an indexed type system. An indexed type language uses an index language in the type system to encode information within types. Wasm-prechk's index language must be capable of capturing enough information about a Wasm-prechk program to statically verify the preconditions of *prechk*-tagged instructions.

The Wasm-prechk index language is designed to encode linear constraints on program values (the details of how they are encoded is discussed in subsection 3.1.1). To do this, we "shadow" Wasm-prechk program values using what we call *index variables* to track constraints on and relationships between program values. Many of these constraints/relationships are written using Wasm operators (*e.g., binop*), since they are the predominant way that Wasm values end up being related to each other. Index variables are associated with program values using *indexed types*, which combine the type information from Wasm: the primitive type t of the value, with the indexed type variable that represents the value in the index language. Finally, we also use index variables to track local variables (specifically the current value of local variables, since they are mutable and may change) via an *index local store*.

3.1 Wasm-prechk Syntax

The syntax of Wasm-prechk has the same structure as Wasm, but different instructions and richer types. First, Wasm-prechk introduces four additional instructions, which are referred to as "*prechk*-tagged" instructions. Second, Wasm-prechk does not support floating point values or unary operators on

$$
\begin{array}{rcl}
testop_{iN} & ::= & \mathsf{eqz} \\
binop_{iN} & ::= & \mathsf{add} \mid \mathsf{sub} \mid \mathsf{shl} \mid \mathsf{or} \mid \ldots \\
relop_{iN} & ::= & \mathsf{eq} \mid \mathsf{ne} \mid \mathsf{gt} \mid \mathsf{ge} \mid \ldots \\
e & ::= & \mathsf{unreachable} \mid \mathsf{nop} \mid \mathsf{drop} \mid \mathsf{select} \mid \\
 & & \mathsf{block}\ \mathit{tfi}\ e^*\ \mathsf{end} \mid \mathsf{loop}\ \mathit{tfi}\ e^*\ \mathsf{end} \mid \mathsf{if}\ \mathit{tfi}\ e^*\ \mathsf{else}\ e^*\ \mathsf{end} \mid \\
 & & \mathsf{br}\ i \mid \mathsf{br\_if}\ i \mid \mathsf{br\_table}\ i^+ \mid \mathsf{return} \mid \mathsf{call}\ i \mid \mathsf{call\_indirect}\ \mathit{tfi} \mid \\
 & & \mathsf{get\_local}\ i \mid \mathsf{set\_local}\ i \mid \mathsf{tee\_local}\ i \mid \mathsf{get\_global}\ i \mid \\
 & & \mathsf{set\_global}\ i \mid t.\mathsf{load}\ (tp\_sx)^?\ align\ o \mid t.\mathsf{store}\ tp^?\ align\ o \mid \\
 & & \mathsf{current\_memory} \mid \mathsf{grow\_memory} \mid t.\mathsf{const}\ c \mid \\
 & & t.binop_t \mid t.testop_t \mid t.relop_t \mid \\
\hline
 & & t.\mathsf{div}_{\mathsf{prechk}} \mid t.\mathsf{call\_indirect}_{\mathsf{prechk}} \mid \\
 & & t.\mathsf{load}_{\mathsf{prechk}}\ (tp\_sx)^?\ align\ o \mid t.\mathsf{store}_{\mathsf{prechk}}\ tp^?\ align\ o
\end{array}
$$

Figure 3.2: Wasm-prechk syntax including the four *prechk*-tagged instructions

integers since they are difficult to reason about (this is explained in more detail in chapter 5). While it would be possible to support them, we would have no more type information about them than Wasm, and the focus of this work is on the type information. Wasm-prechk uses a different representation of types within instructions and functions, as we see in subsection 3.1.1.

Recall from section 2.2 that four Wasm instructions require run-time checks: integer division, indirect function calls, and memory loads and stores. "*prechk*-tagged" instructions refer to four Wasm-prechk instructions, listed in Figure 3.2, that are counterparts to these four Wasm instructions. Intuitively, we add a tag to the instruction to show that it doesn't require run-time checks. Formally, however, different instructions have different semantics and typing rules, as explained below.

3.1.1 The Wasm-prechk Index Language

Wasm-prechk uses an indexed type system. We use the Wasm-prechk index language to encode constraints on program values within types. Figure 3.3 shows the syntax for the index type language. Remember, syntax written in a blue sans serif font denotes a Wasm keyword. Below is a quick overview of each metavariable.

$$
\begin{array}{rcl}
t & ::= & \mathsf{i32} \mid \mathsf{i64} \\
a & ::= & IndexVariable \\
x\ y & ::= & a \mid (t\ c) \mid (\mathsf{binop}\ x\ y) \mid (\mathsf{testop}\ x) \mid (\mathsf{relop}\ x\ y) \\
P & ::= & (=\ x\ y) \mid (\text{if}\ P\ P\ P) \mid \neg P \mid P \wedge P \mid P \vee P \\
\phi & ::= & \circ \mid \phi, (t\ a) \mid \phi, P
\end{array}
$$

Figure 3.3: Syntax of the Wasm-prechk index type language

$$
\begin{array}{rcl}
ti & ::= & (t\ a) \\
l & ::= & ti^* \\
tfi & ::= & ti^*;\ l;\ \phi \rightarrow ti^*;\ l;\ \phi \\
C & ::= & \{\text{func}\ \ tfi^*,\ \text{global}\ \ (\text{mut}^?\ t)^*,\ \text{table}\ n^?,\ \text{memory}\ m^?, \\
 & & \quad \text{local}\ \ t^*,\ \text{label}(ti^*;\ l;\ \phi)^*,\ \text{return}\ (ti^*;\ l;\ \phi)^?\}
\end{array}
$$

Figure 3.4: Wasm-prechk indexed function types

- t represents a primitive Wasm type. We do not reason about floating point values, so it is either a 32-bit integer (i32) or a 64-bit integer (i64).
- a is a type index variable, which is used to track constraints on program values.
- x and y are type indices; they can be an index type variable, a constant with an explicit type, or a Wasm operation on a type index.
- P is a proposition about type indices which can encode equality constraints on type indices, or combine propositions using common first-order logic operators.
- ϕ is the type index context which stores index type variable declarations and propositions. Essentially, it contains all of the knowledge we have about all of the index variables.

Indexed types are used to associate index variables a with values in the program. Figure 3.4 shows the form of an indexed type, ti, which includes both the type t and an index variable a. In Wasm-prechk, we represent the shape of the stack as a sequence of indexed types ti^*.

The index local store associates index variables with local variables. It has an identical form to the stack: a sequence of indexed types to associate index variables with local variables. We use the shorthand l to refer to the index local store. The index type context ϕ is used to reason about the possible values of computations. It stores constraints on and between program values tracked by indexed types representing the stack and index local store.

Wasm-prechk uses indexed "function" types tfi, which, similar to Wasm's function types, are just a precondition and postcondition. However, indexed function types include much more information in their precondition and postcondition! They represent the stack using a sequence of indexed types and track local variables using the index local store, and include ϕ which contains constraints about those values. We see how this information is used in subsection 3.3.2.

We retain C to refer to the module type context in Wasm-prechk, although the representation of module types is slightly different. Wasm function types are replaced with Wasm-prechk indexed function types. Further, the postconditions in the label stack and return stack are replaced with Wasm-prechk indexed postconditions including indexed types, the local index store, and the index type context.

We can now introduce the Wasm-prechk typing judgement for instructions. It is similar to the Wasm typing judgment, but uses indexed function types which include much more information by tracking constraints about program values.

$$\boxed{C \vdash e^* : tfi}$$

Recall that certain Wasm instructions (such as block and call_indirect) include Wasm function types to declare the expected types of their bodies. In Wasm-prechk, we replace those function types with indexed function types.

$$\boxed{s;\ v^*;\ e^* \rightarrow s';\ v'^*;\ e'^*}$$

$$
\begin{array}{rll}
(t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ t.binop & \hookrightarrow & t.\mathsf{const}\ c \\
 & & \text{if } c = binop(c_1, c_2) \\
(t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ t.binop & \hookrightarrow & \mathsf{trap} \\
 & & otherwise \\
s;\ \mathsf{call\_indirect}\ j & \hookrightarrow_i & s_{\text{tab}}(i, j) \\
 & & \text{if } s_{\text{tab}}(i, j)_{\text{code}} = (\mathsf{func}\ tf\ \mathsf{local}\ t^*\ e^*) \\
s;\ \mathsf{call\_indirect}\ j & \hookrightarrow_i & \mathsf{trap} \\
 & & \text{otherwise} \\
s;\ (\mathsf{i32.const}\ k) & & \\
(t.\mathsf{load}\ tp\_sx\ align\ o) & \hookrightarrow_i & s;\ (t.\mathsf{const}\ const_t(b^*)) \\
 & & \text{if } s_{\text{mem}}(i, k + o, |t|) = b^* \\
s;\ (\mathsf{i32.const}\ k) & & \\
(t.\mathsf{load}\ tp\_sx\ align\ o) & \hookrightarrow_i & \mathsf{trap} \\
 & & \text{otherwise} \\
s;\ (\mathsf{i32.const}\ k)\ (t.\mathsf{const}\ c) & & \\
(t.\mathsf{store}\ tp\_sx\ align\ o) & \hookrightarrow_i & s';\ \epsilon \\
 & & \text{if } s' = s \text{ with} \\
 & & \quad \text{mem}(i, k + o, |t|) = bits_t(c) \\
s;\ (\mathsf{i32.const}\ k)\ (t.\mathsf{const}\ c) & & \\
(t.\mathsf{store}\ tp\_sx\ align\ o) & \hookrightarrow_i & \mathsf{trap} \\
 & & \text{otherwise}
\end{array}
$$

Figure 3.5: Wasm instructions that have preconditions for reduction

3.2 Wasm-prechk Dynamic Semantics

Wasm-prechk uses the same reduction relation with the same structure as Wasm (explained in detail in section 2.2). All the reduction rules for all of the Wasm-prechk instructions are the same as they are for Wasm, as presented in section 2.2, except that indexed function types are used instead of Wasm function types. We also have four new instructions, for which we introduce new reduction rules.

The formal reason why certain Wasm instructions require run-time checks is because they have preconditions as part of their semantics. If the preconditions are not met then those instructions trap to avoid undefined behavior

$$\boxed{s;\ v^*;\ e^* \to s';\ v'^*;\ e'^*}$$

$$
\begin{array}{rll}
(t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2) & & \\
\qquad t.\mathsf{div}_{\mathsf{prechk}} & \hookrightarrow\ c & \\
& \quad \text{where } c_2 \neq 0 \wedge c = c_1/c_2 & \\
s;\ (t.\mathsf{const}\ j) & & \\
t.\mathsf{call\_indirect}_{\mathsf{prechk}}\ tfi & \hookrightarrow_i\ \mathsf{call}\ s_{tab}(i,j) & \\
& \quad \text{where } s_{tab}(i,j) = & \\
& \quad \mathsf{func}\ tfi_2\ \mathsf{local}\ t^*\ e^* & \\
& \quad \text{and } tfi_2 <: tfi & \\
s;\ (\mathsf{i32.const}\ k) & & \\
(t.\mathsf{load}_{\mathsf{prechk}}\ (tp\_sx)^?\ a\ o) & \hookrightarrow_i\ t.\mathsf{const}\ const_t(b^*) & \\
& \quad \text{where } s_{mem}(i, k+o, |t|) = b^* & \\
s;\ (\mathsf{i32.const}\ k)\ (t.\mathsf{const}\ c) & & \\
(t.\mathsf{store}_{\mathsf{prechk}}\ tp^?\ a\ o) & \hookrightarrow_i\ s';\ \epsilon & \\
& \quad \text{where } s' = s & \\
& \quad \text{with } \mathrm{mem}(i, k+o, |t|) = bits_t^{|t|}(c) &
\end{array}
$$

Figure 3.6: Behavior of new *prechk*-tagged instructions

(we've reproduced the reduction rules for those instructions in Figure 3.5). The Wasm type system is not expressive enough to ensure these preconditions statically, so they instead must be checked at run-time. However, the Wasm-prechk type system is capable of statically checking these preconditions.

In Wasm-prechk, "*prechk*-tagged" instructions can assume that the preconditions on their behavior hold because it is enforced by the Wasm-prechk type system. This can be seen in the reduction rules for the "*prechk*-tagged" instructions in Figure Figure 3.6, where they do not have rules to trap when their preconditions do not hold. For example, in the $\mathsf{div}_{\mathsf{prechk}}$ rule, the second argument c_2 is guaranteed to be non-zero, so there will be no trap on division-by-zero. $\mathsf{call\_indirect}_{\mathsf{prechk}}$ can assume that the function that gets pulled from the table tfi_2 has a subtype of the expected type tfi, so it is a valid type for the indirect call (we will go over subtyping in more detail in subsection 3.3.1). The *prechk*-tagged memory operations $\mathsf{load}_{\mathsf{prechk}}$ and $\mathsf{store}_{\mathsf{prechk}}$ can assume that the memory operation takes place inside the memory bounds.

Now, we can rewrite the example from Figure 3.1 to use the *prechk*-tagged division instruction, as seen in Figure 3.7. Remember that we know the second argument to $\mathsf{div}_{\mathsf{prechk}}$ is guaranteed to be non-zero, so we know the assumption of the $\mathsf{div}_{\mathsf{prechk}}$ instruction ($c_2 \neq 0$) holds. Since $\mathsf{div}_{\mathsf{prechk}}$ does not require a dynamic check, this program will presumably be faster than the version with the dynamic check. We still have not shown how we statically ensure that this assumption holds, which we will do in subsection 3.3.2.

3.3 The Wasm-prechk Indexed Type System

The Wasm-prechk type system is designed to provide sufficient information to safely eliminate dynamic checks (*i.e.,* to ensure that the required preconditions are met to *prechk*-tag an instruction). As explained in 3.1.1, the Wasm-prechk type system can encode constraints on program values in the preconditions and postconditions of instructions. We will now show how these constraints are added and used.

Recall the form of the Wasm-prechk typing judgement for instructions.

$$\boxed{C \vdash e^* : ti_1^*;\, l_1;\, \phi_1 \rightarrow ti_2^*;\, l_2;\, \phi_2}$$

Under C, the module type context, e^* has the precondition $ti_1^*;\, l_1;\, \phi_1$ and postcondition $ti_2^*;\, l_2;\, \phi_2$. We sometimes use the metavariable abbreviation $tfi ::= ti_1^*;\, l_1;\, \phi_1 \rightarrow ti_2^*;\, l_2;\, \phi_2$ as shorthand for the precondition and postcondition of an instruction.

```
block (i32 i32 → i32)
      (tee_local 0)
      (i32.const 1)
      (get_local 0)
      (select)
      (i32.div_prechk)
end
```

Figure 3.7: An Example of Using a Wasm-prechk *prechk*-tagged Instruction

As in Wasm, Wasm-prechk generally has two kinds of typing rules. Most rules are for inferring or checking the types of instructions (in which case e^* will be a single instruction). There are also a few rules to compose together instruction sequences. We present the typing rules mixed with discussion of those rules. The typing judgment definition in its entirety is reproduced in the appendix Appendix A.

Here are some of the simpler rules. These rules don't use or modify index type information. Rule Unreachable accepts any precondition and guarantees any postcondition since it just causes a trap. In Rule Nop, no changes are made from the precondition to the post condition because the instruction does nothing. Rule Drop consumes the top value from the stack (without caring about its type) and does not change the local index store or index type context.

$$\frac{}{C \vdash \mathsf{unreachable} : ti_1^*; \, l_1; \, \phi_1 \to ti_2^*; \, l_2; \, \phi_2} \textsc{Unreachable}$$

$$\frac{}{C \vdash \mathsf{nop} : \epsilon; \, l; \, \phi \to \epsilon; \, l; \, \phi} \textsc{Nop} \qquad \frac{}{C \vdash \mathsf{drop} : (t \; a); \, l; \, \phi \to \epsilon; \, l; \, \phi} \textsc{Drop}$$

The constant instruction is a simple example of how indexed types work. Rule Const adds a new indexed type onto the stack to track the new program variable $(t \; a)$, declares the new indexed type in the index type context ϕ (the $(t \; a)$ part of $\phi, (t \; a), (= a \; (t \; c))$), and constrains that indexed type to be equal to the constant in ϕ (the $(= a \; (t \; c))$ part of $\phi, (t \; a), (= a \; (t \; c))$). We require a to be fresh (*i.e.,* that it is not present in any types in the program up to this point), so that we know a is not constrained/referenced anywhere in the precondition. This is a common pattern to see in rules which introduce new index variables. Since const does not change or reference the local variables, the local index store l is unchanged between the precondition and postcondition.

$$\frac{a \text{ is fresh}}{C \vdash t.\mathsf{const} \; c : \epsilon; \, l; \, \phi \to (t \; a); \, l; \, \phi, (t \; a), (= a (t \; c))} \textsc{Const}$$

There are several different kinds of operations, but they all work similarly. The binary operator instruction adds constraints between new and old program values, since the result of the instruction is a new program value, while the consumed values may already be constrained. A binary operation consumes two values from the stack, which have associated indexed types $(t\ a_1)$ and $(t\ a_2)$, and produces a value which is associated with the fresh indexed type $(t\ a_3)$. The index type declaration $(t\ a_3)$ is added to the index type context ϕ and a_3 is constrained to be equal to the binary operator applied to the index variables that correspond to the input $(= a_3\ (\|binop\|\ a_1\ a_2)$. As a side note, we use $\|binop\|$ to indicate that we are moving the *binop* (or relop or testop) from Wasm-prechk to the index language, where it will be interpreted by the semantics of the index language. Binary operators do not affect or use local variables, so the local index store. l, is the same in the precondition and postcondition.

$$\frac{a_3 \text{ is fresh}}{\begin{array}{l} C \vdash t.binop : (t\ a_1)\ (t\ a_2); l; \phi \\ \qquad \rightarrow (t\ a_3); l; \phi, (t\ a_3), (= a_3\ (\|binop\|\ a_1\ a_2)) \end{array}}\ \textsc{Binop}$$

$$\frac{a_3 \text{ is fresh}}{\begin{array}{l} C \vdash t.testop : (t\ a_1)\ l; \phi \\ \qquad \rightarrow (\mathsf{i32}\ a_2); l; \phi, (t\ a_2), (= a_2\ (\|testop\|\ a_1)) \end{array}}\ \textsc{Testop}$$

$$\frac{a_3 \text{ is fresh}}{\begin{array}{l} C \vdash t.relop : (t\ a_1)\ (t\ a_2); l; \phi \\ \qquad \rightarrow (t\ a_3); l; \phi, (t\ a_3), (= a_3\ (\|relop\|\ a_1\ a_2)) \end{array}}\ \textsc{Relop}$$

Rule Select constrains indexed types in a rather complex way. Select consumes three values from the stack, it returns the second value if the third value is zero, and otherwise returns the first value (similar to C's ternary operator). We use the type-level "if" to allow the constraint on the result to depend on the third value consumed: $(\text{if } (= a\ (\mathsf{i32}\ 0))\ (= a_3\ a_2)\ (= a_3\ a_1))$.

$$\frac{a_3 \text{ is fresh}}{\begin{array}{l} C \vdash \mathsf{select} : (t\ a_1)\ (t\ a_2)\ (i32\ a);\ l;\ \phi \\ \qquad \to (t\ a_3);\ l;\ \phi, (t\ a_3), \\ \qquad\qquad (\text{if } (= a\ (\mathsf{i32}\ 0))\ (= a_3\ a_2)\ (= a_3\ a_1)) \end{array}}\ \textsc{Select}$$

The rules for the three different kinds of blocks (block, loop, and if) are similar to Wasm. They simply ensure that the interior instruction sequence has the expected type under the context with the expected postcondition (or precondition in the case of loop) appended to the local stack. In Wasm-prechk, if blocks make extra assumptions about the consumed value in the subsequences (that it is non-zero in the first sequence and zero in the second), because those constraints must be true for that sequence to be executed. While if and block append their postcondition to the label stack for typechecking branching instructions within the block, loop appends its precondition because branching to a loop means running the loop again.

$$\frac{C_2, \text{label}\ (ti_2^*;\ l_2;\ \phi_2) \vdash e^* : ti_1^*;\ l_1;\ \phi_1 \to ti_2^*;\ l_2;\ \phi_2}{C \vdash \mathsf{block}\ (ti_1^*;\ l_1;\ \phi_1 \to ti_2^*;\ l_2;\ \phi_2)\ e^*\ \mathsf{end} : ti_1^*;\ l_1;\ \phi_1 \to ti_2^*;\ l_2;\ \phi_2}\ \textsc{Block}$$

$$\frac{C_2, \text{label}\ (ti_1^*;\ l_1;\ \phi_1)^* \vdash e^* : ti_1^*;\ l_1;\ \phi_1 \to ti_2^*;\ l_2;\ \phi_2}{C \vdash \mathsf{loop}\ (ti_1^*;\ l_1;\ \phi_1 \to ti_2^*;\ l_2;\ \phi_2)\ e^*\ \mathsf{end} : ti_1^*;\ l_1;\ \phi_1 \to ti_2^*;\ l_2;\ \phi_2}\ \textsc{Loop}$$

$$\frac{\begin{array}{l} C_2, \text{label}\ (ti_2^*;\ l_2;\ \phi_2) \vdash e_1^* : ti_1^*;\ l_1;\ \phi_1, \neg(= a\ (\mathsf{i32}\ 0)) \to ti_2^*;\ l_2;\ \phi_2 \\ C_2, \text{label}\ (ti_2^*;\ l_2;\ \phi_2) \vdash e_2^* : ti_1^*;\ l_1;\ \phi_1, (= a\ (\mathsf{i32}\ 0))) \to ti_2^*;\ l_2;\ \phi_2 \end{array}}{C \vdash \mathsf{if}\ (ti_1^*;\ l_1;\ \phi_1 \to ti_2^*;\ l_2;\ \phi_2)\ e_1^*\ \mathsf{else}\ e_2^*\ \mathsf{end} : ti_1^*;\ l_1;\ \phi_1 \to ti_2^*;\ l_2;\ \phi_2}\ \textsc{If}$$

One thing to note is that all three of these rules include their expected preconditions and postconditions as part of their syntax. We consider the index variables in these indexed function types to be unification variables rather than literals, allowing them to match any literal as long as the types unify. Intuitively, this is very similar to alpha equivalence, where the precon-

dition matches any preceding postcondition with the same structure as long as the variable can be renamed to match. The postcondition appended to the label stack also has unification variables instead of the supplied literals.

The rules for branching instructions and return are similar to Wasm. However, br_if adds the assumption that the consumed value is zero to its postcondition. This assumption can be safely added because the value must be zero for execution to continue without a branch occurring. If the consumed value is constrained to be non-zero in the indexed type system, then this will cause a contradiction in the constraints of the index type context ϕ. However, that is fine since this means that no instructions following the br_if will be executed. Also remember the above note that the postconditions on the label stack contain unification variables, not literals.

Recall from section 2.2 that br_table branches to one of many different labels. Thus, we must ensure that every possible branching postcondition to which it might branch is implied by the precondition.

$$\frac{C_{\text{label}}(i) = ti^*;\ l_1;\ \phi_1}{C \vdash \mathsf{br}\ i : ti_1^*\ ti^*;\ l_1;\ \phi_1 \to ti_2^*;\ l_2;\ \phi_2}\ \textsc{Br}$$

$$\frac{C_{\text{return}} = ti^*;\ l_1;\ \phi_1}{C \vdash \mathsf{return} : ti_1^*\ ti^*;\ l_1;\ \phi_1 \to ti_2^*;\ l_2;\ \phi_2}\ \textsc{Return}$$

$$\frac{C_{\text{label}}(i) = ti^*;\ l_1;\ \phi_1, \neg(=\ a\ (\mathsf{i32}\ 0))}{C \vdash \mathsf{br\_if}\ i : ti^*\ a;\ l_1;\ \phi_1 \to ti^*;\ l_1;\ \phi_1, (=\ a\ (\mathsf{i32}\ 0))}\ \textsc{Br-If}$$

$$\frac{(C_{\text{label}}(i) = ti^*;\ l_1;\ \phi_i)^+ \qquad (\phi_1 \implies \phi_i)^n}{C \vdash \mathsf{br\_table}\ i^+ : ti_1^*\ ti^*\ a;\ l_1;\ \phi_1 \to ti_2^*;\ l_2;\ \phi_2}\ \textsc{Br-Table}$$

Recall that functions are declared within the module with a specific indexed function type *tfi*, that is a precondition and postcondition. These declared indexed function types are placed inside the module type context C. Direct function calls call i have the same type as the declared indexed

function type of the function they are calling with two differences. First, the local index store is unchanged, since the called function will have been turned into a closure that operates on separate local variables in a local block. Second, the index type context in the postcondition of the call is extended with the declarations and constraints from the precondition of the call. The precondition and postcondition of a function can only contain constraints about the arguments supplied to that function, so simply copying the postcondition of the function would result in the loss of information about all other index variables.

Indirect function calls call_indirect $ti_1^*; l_1; \phi_1 \rightarrow ti_2^*; l_2; \phi_2$ include the expected indexed function type $ti_1^*; l_1; \phi_1 \rightarrow ti_2^*; l_2; \phi_2$ provided as part of their syntax (the same note about index variables being unification variables from above holds). Remember that indirect function calls perform a run time typecheck against the closure that they end up calling, so we assume statically that the check will proceed because if it does not the program will trap (trap satisfies any *tfi*, including the one for the indirect call) and not be able to do any harm. The same two differences described above between the expected indexed function type *tfi* and the type of the call_indirect *tfi* instruction also hold.

$$\frac{C_{\text{func}}(i) = ti_1^*; l_1; \phi_1 \rightarrow ti_2^*; l_2; \phi_2}{C \vdash \text{call } i : ti_1^*; l; \phi_1 \rightarrow ti_2^*; l; \phi_1, \phi_2} \textsc{Call}$$

$$\frac{C_{\text{table}}(i) = (j, tfi_2^*)}{\begin{array}{l} C \vdash \text{call\_indirect } ti_1^*; l_1; \phi_1 \rightarrow ti_2^*; l_2; \phi_2 \\ \quad : ti_1^* \ (i32\ a); l; \phi_1 \rightarrow ti_2^*; l; \phi_1, \phi_2 \end{array}} \textsc{Call-Indirect}$$

The only instructions that actually mutate the local index store are those that operate on local variables. get_local produces a fresh indexed type $(t\ a_2)$ that is constrained to be equal to the index variable associated with the local being retrieved. set_local works in the reverse direction, replacing the index variable associated with the local being set. Because set_local reasons about

local variables, which are not part of the instruction sequence (unlike values on the stack), we can copy the index variable instead of creating a fresh one and constraining it to be equal like for get_local Finally, tee_local is effectively a combined set_local and get_local that consumes and immediately regurgitates a value, like the Unix tool "tee". Thus, the typing rule is similarly a combination of the set_local and get_local rules, where the indexed type from the stack replaces the local variable indexed type, and a fresh indexed type is produced that is constrained to be equal to the consumed index variable.

$$\frac{C_{\text{local}}(i) = t \qquad l(i) = (t\ a) \qquad a_2 \text{ is fresh}}{C \vdash \text{get\_local } i : \epsilon;\ l;\ \phi \to (t\ a_2);\ l;\ \phi, (t\ a_2), (=\ a_2\ a)}\ \textsc{Get-Local}$$

$$\frac{C_{\text{local}}(i) = t \qquad l_2 = l_1[i := (t\ a)]}{C \vdash \text{set\_local } i : (t\ a);\ l_1;\ \phi \to \epsilon;\ l_2;\ \phi}\ \textsc{Set-Local}$$

$$\frac{C_{\text{local}}(i) = t \qquad l_2 = l_1[i := (t\ a)] \qquad a_2 \text{ is fresh}}{C \vdash \text{tee\_local } i : (t\ a);\ l_1;\ \phi \to (t\ a_2);\ l_2;\ \phi, (t\ a_2), (=\ a_2\ a)}\ \textsc{Tee-Local}$$

Instructions for getting and setting globals produce and consume unconstrained values respectively. Global variables are difficult to reason about in the type system since they are different between modules. At compile-time, before linking, a module has no information about globals from another module which would be necessary for reasoning about the types of functions imported from the other module. Therefore, we do not track index variables for globals (we just treat them as unconstrained values when they are introduced onto the stack by get_global). We do still statically ensure the same properties as Wasm: that the value is of the correct Wasm type and in the case of setting a global variable that the global variable is mutable (has the

mut flag in its type).

$$\frac{C_{\text{global}}(i) = \text{mut}^? \ t \qquad a \text{ is fresh}}{C \vdash \mathsf{get\_global}\ i : \epsilon;\ l;\ \phi \to (t\ a);\ l;\ \phi, (t\ a)}\ \textsc{Get-Global}$$

$$\frac{C_{\text{global}}(i) = \text{mut}\ t}{C \vdash \mathsf{set\_global}\ i : (t\ a);\ l;\ \phi \to \epsilon;\ l;\ \phi}\ \textsc{Set-Global}$$

The typing rules for memory instructions are very similar to Wasm as we do not reason about the contents of memory or how its size can change throughout a program. As in Wasm, there are many small details related to how exactly values are loaded and stored that are not particularly important to the understanding of the type system, but they are explained with the reduction rules for these values in section 2.2. One thing that does not appear in the Wasm reduction rules but mysteriously appears in the typing rules without much explanation is *align*. It is checked against the size of the type of the value being stored/loaded $|t|$ (or optionally $|tp|$, which should be less than $|t|$) in the premise $2^{align} \leq (|tp| <)^?|t|$.

$$\frac{C_{\text{memory}} = n \qquad 2^{align} \leq (|tp| <)^?|t| \qquad a_2 \text{ is fresh}}{C \vdash t.\mathsf{load}\ (tp\_sx)^?\ align\ o : (\mathsf{i32}\ a_1);\ l;\ \phi \to (t\ a_2);\ l;\ \phi, (t\ a_2)}\ \textsc{Mem-Load}$$

$$\frac{C_{\text{memory}} = n \qquad 2^{align} \leq (|tp| <)^?|t|}{C \vdash t.\mathsf{store}\ tp^?\ align\ o : (\mathsf{i32}\ a_1)\ (t\ a_2);\ l;\ \phi \to \epsilon;\ l;\ \phi}\ \textsc{Mem-Store}$$

$$\frac{C_{\text{memory}} = n \qquad a \text{ is fresh}}{C \vdash \mathsf{current\_memory}\ : \epsilon;\ l;\ \phi \to (\mathsf{i32}\ a);\ l;\ \phi, (\mathsf{i32}\ a)}\ \textsc{Current-Memory}$$

$$\frac{C_{\text{memory}} = n \qquad a_2 \text{ is fresh}}{C \vdash \mathsf{grow\_memory}\ : (\mathsf{i32}\ a_1);\ l;\ \phi \to (\mathsf{i32}\ a_2);\ l;\ \phi, (\mathsf{i32}\ a_2)}\ \textsc{Grow-Memory}$$

The last rules are the ones that can be used to compose sequences of

instructions. The first rule is for the empty instruction sequence ϵ, which, similar to Wasm, simply has the same precondition and postcondition $\epsilon;\ l;\ \phi$. Second, we have Rule Stack-Poly to add stack polymorphism (see subsection 2.3.1). Third, there is a rule to compose a sequence of instructions e_1^* with another instruction e_2.

$$\frac{}{C \vdash \epsilon : \epsilon;\ l;\ \phi \to \epsilon;\ l;\ \phi}\ \textsc{Empty}$$

$$\frac{C \vdash e^* : ti_1^*;\ l_1;\ \phi_1 \to ti_2^*;\ l_2;\ \phi_2}{C \vdash e^* : ti^*\ ti_1^*;\ l_1;\ \phi_1 \to ti^*\ ti_2^*;\ l_2;\ \phi_2}\ \textsc{Stack-Poly}$$

$$\frac{\begin{array}{c} C \vdash e_1^* : ti_1^*;\ l_1;\ \phi_1 \to ti_2^*;\ l_2;\ \phi_2 \\ C \vdash e_2 : ti_2^*;\ l_2;\ \phi_2 \to ti_3^*;\ l_3;\ \phi_3 \end{array}}{C \vdash e_1^*\ e_2 : ti_1^*;\ l_1;\ \phi_1 \to ti_3^*;\ l_3;\ \phi_3}\ \textsc{Composition}$$

3.3.1 Subtyping, Implication, and Constraint Satisfaction

One issue with adding the index type context ϕ to preconditions and postconditions is that the postcondition of one instruction and the precondition of the next instruction might not match up exactly. For example, one instruction may ensure a value is greater than ten, but the next just wants the value to be greater than zero. Intuitively, if a value, "x", is greater than ten it must also be greater than zero, and we want the Wasm-prechk type system to be able to figure this out as well. However, computers as of yet are unable to use intuition, so we must instead formalize this.

Our formalization of this problem is to allow *strengthening* preconditions and *weakening* postconditions. Strengthening and weakening is based on implication ($\implies$). We say that $\phi_1 \implies \phi_2$ when the following holds: if ϕ_1 is satisfied, then ϕ_2 must also be satisfied. If $\phi_1 \implies \phi_2$, then we consider ϕ_1 to be stronger than ϕ_2, and ϕ_2 to be weaker than ϕ_1. This solves the aforementioned problem because we can weaken "x is greater than 10" to "x is greater than 0" (or equivalently strengthen "x is greater than 0" to "x is greater than 10").

To fit strengthening and weakening into the type system, we define a subtyping judgment based on implication. We essentially parameterize our typing judgment with the implication relation. As an aside, we show that this is a practical thing to do and that such a relation exists by implementing such an implication relation using Z3 (see subsection 5.2.1). The Rule Implies says that if an indexed function type tfi_1 has a stronger precondition and weaker postcondition than some other indexed function type tfi_2, and is otherwise equivalent, then tfi_1 is a subtype of tfi_2 since it can safely be used in place of tfi_2.

$$\frac{\phi_0 \implies \phi_1 \qquad \phi_2 \implies \phi_3}{ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2 <: ti_1^*; l_1; \phi_0 \to ti_2^*; l_2; \phi_3}\ \textsc{Implies}$$

We then use this in the Wasm-prechk type system by adding a typing rule that allows the indexed function type for a list of instructions to be replaced by a subtype of that indexed function type.

$$\frac{ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2 <: ti_1^*; l_1; \phi_0 \to ti_2^*; l_2; \phi_3 \qquad C \vdash e^* \to ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2}{C \vdash e^* \to ti_1^*; l_1; \phi_0 \to ti_2^*; l_2; \phi_3}\ \textsc{Subtyping}$$

3.3.2 Using Types for Check Elimination

In Section 3.2 we explained that *prechk*-tagged instructions do not need dynamic checks because of the static guarantees of the Wasm-prechk type system. Here, we see how the Wasm-prechk type system provides those guarantees by looking at the typing rules for each of the *prechk*-tagged instructions.

Integer division simply requires that the second argument is non-zero. The premise $\phi \implies \neg(= a_2\ 0)$ requires that the index constraints satisfy the proposition $a_2 \neq 0$ for the pre-checked instruction to be safe. Therefore, since a divide-by-zero is provably absent, it is safe to use the *prechk*-tagged

division instruction. As an aside, recall that requiring a_3 is fresh ensures that it is not declared, constrained, or referenced by the precondition.

$$\frac{\phi \implies \neg(= \ a_2 \ 0) \qquad a_3 \text{ is fresh}}{\begin{array}{l} C \vdash t.\mathsf{div}_{\mathsf{prechk}} : (t\ a_1)\ (t\ a_2);\ l;\ \phi \\ \qquad\qquad \to (t\ a_3);\ l;\ \phi, (t\ a_3), (= a_3\ (\|div\|\ a_1\ a_2)) \end{array}} \textsc{Div-Prechk}$$

Tagging memory loads and stores with *prechk* requires ensuring that the memory index is valid. Since Wasm and Wasm-prechk use linear memory, which is a contiguous block of memory, we simply have to ensure that the index is within those bounds. The initial memory size is the number of 64 Ki pages (65,536 bytes), so we check that the constraints in the index type context ensure that the memory index plus the static offset is between 0 and $65,536 - width$. We use $width$ as a shorthand to denote the number of bytes that is being stored/loaded, it is equal to $|t|/8$ if $tp^? = \epsilon$, and otherwise equal to $|tp|/8$.

Unfortunately, while the size of memory may be grown during program execution, we are currently unable to reason about changing memory size. Therefore, we just use the initial memory size.

$$\frac{\begin{array}{l} C_{\text{memory}} = n \qquad 2^{align} \leq (|tp| <)^? |t| \qquad a_3 \text{ is fresh} \\ \phi \implies (\mathsf{ge}\ (\mathsf{add}\ a_1\ (\mathsf{i32}\ o))(\mathsf{i32}\ 0)), \\ \qquad (\mathsf{le}\ (\mathsf{add}\ a_1\ (\mathsf{add}\ (\mathsf{i32}\ o + width)))\ (\mathsf{i32}\ n * 64\text{Ki})) \end{array}}{\begin{array}{l} C \vdash t.\mathsf{load}_{\mathsf{prechk}}\ (tp\_sx)^?\ align\ o : (\mathsf{i32}\ a_1);\ l;\ \phi \\ \qquad\qquad \to (t\ a_2);\ l;\ \phi, (t\ a_2) \end{array}} \textsc{Load-Prechk}$$

$$\frac{\begin{array}{l} C_{\text{memory}} = n \qquad 2^{align} \leq (|tp| <)^? |t| \\ \phi \implies (\mathsf{ge}\ (\mathsf{add}\ a_1\ (\mathsf{i32}\ o))\ (\mathsf{i32}\ 0)), \\ \qquad (\mathsf{le}\ (\mathsf{add}\ a_1\ (\mathsf{add}\ (\mathsf{i32}\ o + width)))\ (\mathsf{i32}\ n * 64\text{Ki})) \end{array}}{C \vdash t.\mathsf{store}_{\mathsf{prechk}}\ tp^?\ align\ o : (\mathsf{i32}\ a_1)\ (t\ a_2);\ l;\ \phi \to \epsilon;\ l;\ \phi} \textsc{Store-Prechk}$$

Indirect function calls in Wasm require a dynamic check to ensure that the index into the table points to a function of a suitable type (recall the

explanation of tables and call_indirect from section 2.2). Proving the safety of an indirect function call involves showing that every possible function that could be called will not cause a run-time type error. We ensure this by requiring that the type of every function at every possible index value has a subtype of the expected type: $\forall 0 < i \leq n.(\phi \implies \neg(= (\mathsf{i32}\ i)\ a)) \vee \mathit{tfis}(i) <: \mathit{tfi}$ where $\mathit{tfis} = (\mathit{tfi}_2...)$. The $\forall$ and $\vee$ (as in $(\phi \implies \neg(= (\mathsf{i32}\ i)\ a)) \vee \mathit{tfis}(i) <: \mathit{tfi}$) in this case are at the meta level and not within the index language. Further, we must show that the provided table index is within the table boundaries: $\phi \implies (\mathsf{gt}\ n\ a) \wedge (\mathsf{le}\ (\mathsf{i32}\ 0)\ a)$.

$$\frac{\begin{array}{c} C_{table}(i) = (n, (\mathit{tfi}_2...)) \\ \phi \implies (\mathsf{gt}\ n\ a) \wedge (\mathsf{le}\ (\mathsf{i32}\ 0)\ a) \\ \mathit{tfis} = (\mathit{tfi}_2...) \qquad \mathit{tfi} = ti_1^*;\ l_1;\ \phi_1 \to ti_2^*;\ l_2;\ \phi_2 \\ \forall 0 < i \leq n.\ (\phi \implies \neg(= (\mathsf{i32}\ i)\ a)) \vee\ \mathit{tfis}(i) <: \mathit{tfi} \end{array}}{\begin{array}{l} C \vdash \mathsf{call\_indirect}_{\mathsf{prechk}}\ \mathit{tfi} : ti_1^*\ (i32\ a);\ l;\ \phi_1 \\ \qquad\qquad \to ti_2^*;\ l;\ \phi_1, \phi_2 \end{array}}\ \textsc{Call-Indirect-Prechk}$$

An Example of Using Types for Check Elimination Here we present a short contrived example of using types for check elimination, based on the example from Figure 3.7. We are typechecking a $\mathsf{div}_{\mathsf{prechk}}$. First, we give the module type context C_1, which contains one local variable, which is an i32, and the instruction sequence we are typing, which is a safe division happening inside of a block (Figure 3.7). We also use the shorthand C_2 for C_1 extended with the label postcondition for the instructions inside the block.

$$\begin{array}{rcl} C_1 & = & \{\text{func } \epsilon,\ \text{global } \epsilon,\ \text{table } \epsilon,\ \text{memory } \epsilon, \\ & & \quad \text{local i32},\ \text{label } \epsilon,\ \text{return } \epsilon\} \\ C_2 & = & \{\text{func } \epsilon,\ \text{global } \epsilon,\ \text{table } \epsilon,\ \text{memory } \epsilon, \\ & & \quad \text{local i32},\ \text{label } ((\mathsf{i32}\ a_0);\ (\mathsf{i32}\ a_2);\ \circ),\ \text{return } \epsilon\} \end{array}$$

The block takes two integers, and, using the local as temporary storage, either divides the first by the second, or the first by 1 if the second is 0. In the example we build up type derivations to reach a typing derivation for the whole block. We first state the rule that we will use, than give the derivation (in many cases we apply Rule Stack-Poly inline for brevity). Typechecking

the example relies on select, where the second argument (i32.const 1) is chosen if the first argument (and the conditional) get_local 0 is equal to zero, so the result must be non-zero.

1. Rule Tee-Local and Rule Stack-Poly

   Note that $\phi_1 = \circ, (\text{i32}\ a_1), (\text{i32}\ a_2)$, and $\phi_2 = \phi_1, (\text{i32}\ a_4), (= a_4\ a_2)$.

$$\frac{\begin{array}{c} C_{2\text{local}}(0) = \text{i32} \\ (\text{i32}\ a_2) = (\text{i32}\ a_3)[0 := (\text{i32}\ a_2)] \qquad a_4 \text{ is fresh} \end{array}}{\begin{array}{l} C_2 \vdash \text{tee\_local}\ 0 : (\text{i32}\ a_1)\ (\text{i32}\ a_2); (\text{i32}\ a_3); \phi_1 \\ \qquad\qquad \to (\text{i32}\ a_1)\ (\text{i32}\ a_4); (\text{i32}\ a_2); \phi_2 \end{array}}$$

2. Rule Const and Rule Stack-Poly

   Note that $\phi_3 = \phi_2, (= a_4\ a_2), (\text{i32}\ a_5), (= a_5\ (\text{i32}\ 1))$.

$$\frac{a_5 \text{ is fresh}}{\begin{array}{l} C_2 \vdash \text{i32.const}\ 1 : (\text{i32}\ a_1)\ (\text{i32}\ a_4); (\text{i32}\ a_2); \phi_2 \\ \qquad\qquad \to (\text{i32}\ a_1)\ (\text{i32}\ a_4)\ (\text{i32}\ a_5); (\text{i32}\ a_2); \phi_3 \end{array}}$$

3. Rule Composition

$$\frac{1. \qquad 2.}{\begin{array}{l} C_2 \vdash (\text{tee\_local}\ 0)\ (\text{i32.const}\ 1) \\ \quad : (\text{i32}\ a_1)\ (\text{i32}\ a_2); (\text{i32}\ a_3); \phi_1 \\ \quad\ \to (\text{i32}\ a_1)\ (\text{i32}\ a_4)\ (\text{i32}\ a_5); (\text{i32}\ a_2); \phi_3 \end{array}}$$

4. Rule Get-Local and Rule Stack-Poly

   Note that $\phi_4 = \phi_3, (\text{i32}\ a_6), (= a_6\ a_2)$.

$$\frac{a_6 \text{ is fresh}}{\begin{array}{l} C_2 \vdash \text{get\_local}\ 0 : (\text{i32}\ a_1)\ (\text{i32}\ a_4)\ (\text{i32}\ a_5); (\text{i32}\ a_2); \phi_3 \\ \qquad\qquad \to (\text{i32}\ a_1)\ (\text{i32}\ a_4)\ (\text{i32}\ a_5)\ (\text{i32}\ a_6); (\text{i32}\ a_2); \phi_4 \end{array}}$$

5. Rule Composition

Note that $\phi_4 = \phi_3, (\mathsf{i32}\ a_6), (= a_6\ a_2)$.

$$\frac{3. \qquad 4.}{\begin{array}{l} C_2 \vdash (\mathsf{tee\_local}\ 0)\ (\mathsf{i32.const}\ 1)\ (\mathsf{get\_local}\ 0) \\ \quad : (\mathsf{i32}\ a_1)\ (\mathsf{i32}\ a_2);\ (\mathsf{i32}\ a_3);\ \phi_1 \\ \quad\ \to (\mathsf{i32}\ a_1)\ (\mathsf{i32}\ a_4)\ (\mathsf{i32}\ a_5)\ (\mathsf{i32}\ a_6);\ (\mathsf{i32}\ a_2);\ \phi_4 \end{array}}$$

6. Rule SELECT and Rule STACK-POLY

   Note that $\phi_5 = \phi_4, (\mathsf{i32}\ a_7), (\text{if } (= a_6\ (\mathsf{i32}\ 0))\ (= a_7\ a_5)(= a_7\ a_4))$.

$$\frac{a_7 \text{ is fresh}}{\begin{array}{l} C_2 \vdash \mathsf{select} \\ \quad : \to (\mathsf{i32}\ a_1)\ (\mathsf{i32}\ a_4)\ (\mathsf{i32}\ a_5)\ (\mathsf{i32}\ a_6);\ (\mathsf{i32}\ a_2);\ \phi_4 \\ \quad\ \to ((\mathsf{i32}\ a_1)\ (\mathsf{i32}\ a_7);\ (\mathsf{i32}\ a_2);\ \phi_5 \end{array}}$$

7. Rule COMPOSITION

$$\frac{5. \qquad 6.}{\begin{array}{l} C_2 \vdash (\mathsf{tee\_local}\ 0)\ (\mathsf{i32.const}\ 1)\ (\mathsf{get\_local}\ 0)\ (\mathsf{select}) \\ \quad : (\mathsf{i32}\ a_1)\ (\mathsf{i32}\ a_2);\ (\mathsf{i32}\ a_3);\ \phi_1 \\ \quad\ \to ((\mathsf{i32}\ a_1)\ (\mathsf{i32}\ a_7);\ (\mathsf{i32}\ a_2);\ \phi_5 \end{array}}$$

8. Rule DIV-PRECHK

   Note that $\phi_5 = \phi_5, (\mathsf{i32}\ a_8), (= a_8(\|\mathsf{div}\|\ a_1\ a_7))$.

$$\frac{\phi_5 \implies \neg(= a_7\ 0) \qquad a_8 \text{ is fresh}}{\begin{array}{l} C_2 \vdash (\mathsf{i32.div}_{\mathsf{prechk}}) \\ \quad : ((\mathsf{i32}\ a_1)\ (\mathsf{i32}\ a_7);\ (\mathsf{i32}\ a_2);\ \phi_5 \\ \quad\ \to (\mathsf{i32}\ a_8);\ (\mathsf{i32}\ a_2);\ \phi_6 \end{array}}$$

9. Rule COMPOSITION

$$\frac{7. \qquad 8.}{\begin{array}{l} C_2 \vdash (\mathsf{tee\_local}\ 0)\ (\mathsf{i32.const}\ 1)\ (\mathsf{get\_local}\ 0)\ (\mathsf{select})\ (\mathsf{i32.div}_{\mathsf{prechk}}) \\ \quad : (\mathsf{i32}\ a_1)\ (\mathsf{i32}\ a_2);\ (\mathsf{i32}\ a_3);\ \phi_1 \\ \quad\ \to (\mathsf{i32}\ a_8);\ (\mathsf{i32}\ a_2);\ \phi_6 \end{array}}$$

10. Rule Block

$$\frac{9.}{\begin{array}{l} C_1 \vdash\ \mathsf{block}\ (\mathsf{i32}\ a_1)\ (\mathsf{i32}\ a_2);\ (\mathsf{i32}\ a_3);\ \phi_1 \\ \qquad\qquad \to (\mathsf{i32}\ a_8);\ (\mathsf{i32}\ a_2);\ \phi_6 \\ \qquad (\mathsf{tee\_local}\ 0) \\ \qquad (\mathsf{i32.const}\ 1) \\ \qquad (\mathsf{get\_local}\ 0) \\ \qquad (\mathsf{select}) \\ \qquad (\mathsf{i32.div}) \\ \quad \mathsf{end} : (\mathsf{i32}\ a_1)\ (\mathsf{i32}\ a_2);\ (\mathsf{i32}\ a_3);\ \phi_1 \\ \quad \to (\mathsf{i32}\ a_8);\ (\mathsf{i32}\ a_2);\ \phi_6 \end{array}}$$

3.3.3 Module Types

The complete module typing rules are in Figure 3.8 (note that im is an import and ex is an export). Functions f, typecheck their body e^* under the module type context C with the expected postcondition $ti_2^*;\ l_2;\ \phi_2$ in the label stack and return position, and with the local index store $(t_1\ a_1)^*\ (t\ a_2)^*$ constructed from the function's arguments $(t_1\ a_1)^*$ and declared locals $(t\ a_2)^*$. Global variables $glob$ must ensure that their initialization instructions e^* produce a value of the proper type t. Exported global variables cannot be mutable, if there are any exports defined, the global cannot have the mutable tag mut: $ex^* = \epsilon \vee tg = t$. Tables tab ensure that the indices i^n refer to well-typed functions and there are exactly as many indices as the expected size n. Memory mem simply has its declared initial size n from which it can only grow bigger. All imported functions, globals, tables, and memories are

expected to have their declared type. They are typechecked during linking.

Typechecking a module involves typechecking every component of the module. Functions, f, are typechecked under the module type context, C, containing the entirety of the module. This means that functions can refer to themselves, other functions, all globals, the table, and memory. This may seem to be a circular definition, but the type of the module is declared statically (as the combined declared types of all the module components), so it is just checking against the expected module index type context. Globals, *glob*, are typechecked under the module index context containing only the global variable declarations preceding the current declaration.

$$\frac{\begin{array}{c} tfi = (t_1\ a_1)^*;\ \epsilon;\ \phi_1 \to ti_2^*;\ l_2;\ \phi_2 \\ C_2 = C, \text{local}\ t_1^*\ t^*, \text{label}\ (ti_2^*;\ l_2;\ \phi_2), \text{return}\ (ti_2^*, l_2, \phi_2) \\ C_2 \vdash e^* : \epsilon;\ (t_1\ a_1)^*\ (t\ a_2)^*;\ \phi_1 \to ti_2^*;\ l_2;\ \phi_2 \end{array}}{C \vdash ex^*\ \mathsf{func}\ tfi\ \mathsf{local} t^*\ e^* : ex^*\ tfi}\ \textsc{Func}$$

$$\frac{}{C \vdash ex^*\ \mathsf{func}\ tfi\ im : ex^*\ tfi}$$

$$\frac{tg = mut^?\ t \qquad ex^* = \epsilon \vee tg = t \qquad C \vdash e^* : \epsilon;\ \epsilon;\ \phi_1 \to (t\ a);\ \epsilon;\ \phi_2}{C \vdash ex^*\ \mathsf{global}\ tg\ e^* : ex^*\ tg}$$

$$\frac{tg = t}{C \vdash ex^*\ \mathsf{global}\ tg\ im : ex^*\ tg}$$

$$\frac{(C_{\text{func}}(i) = tfi)^n}{C \vdash ex^*\ \mathsf{table}\ n\ i^n : ex^*\ (n, tfi^n)}\ \textsc{Table}$$

$$\frac{}{C \vdash ex^*\ \mathsf{table}\ (n, tfi^n)\ im : ex^*\ (n, tfi^n)} \qquad \frac{}{C \vdash ex^*\ \mathsf{memory}\ n : ex^*\ n}$$

$$\frac{}{C \vdash ex^*\ \mathsf{memory}\ n\ im : ex^*\ n}$$

$$\frac{\begin{array}{c} (C \vdash f : ex_f^*\ tfi)^* \qquad (C_i \vdash glob_i : ex_g^*\ tg_i)^* \\ (C \vdash tab : ex_t^*\ (n, tfi^n))^? \qquad (C \vdash mem : ex_m^*\ n)^? \\ (C_i = \{\text{global}\ tg^{i-1}\})_i^* \qquad ex_f^{*\,*}\ ex_g^{*\,*}\ ex_t^{*\,?}\ ex_m^{*\,?}\ \text{distinct} \\ C = \{\text{func}\ tfi^*, \text{global}\ tg^*, \text{table}\ (n, tfi^n)^?, \text{memory}\ n^?\} \end{array}}{\vdash \mathsf{module}\ f^*\ glob^*\ tab^?\ mem^?}$$

Figure 3.8: Indexed Module Typing Rules

Chapter 4

Metatheory

Now that we have introduced Wasm-prechk and shown how it can be used for reasoning, it is time to reason about Wasm-prechk itself. First, we will take a look at the relationship between Wasm and Wasm-prechk, by showing methods to translate Wasm programs to Wasm-prechk programs and vice versa. Then, we will prove the type safety of Wasm-prechk, to ensure that our claim that Wasm-prechk is as safe as Wasm is valid. However, before we can do any of that, we must "complete" our reasoning ability by creating a way to connect the reduction relation form with the type system.

4.1 Administrative Typing Rules

While we have shown the Wasm-prechk typing rules for instructions within a static context, we still need typing rules for administrative instructions and the store used in reduction. *Administrative instructions* are introduced for reduction to keep track of information during reduction. For example, local is the result of reducing a closure call; it is used to reduce a function body within the closed environment of the closure. They are not part of the surface syntax of a language (*e.g.,* you cannot put a local block in a Wasm-prechk program), and can only appear as an intermediate term during reduction. Figure 4.2 shows the Wasm-prechk typing rules for module instances $inst$, the run time store s, and various data structures contained within s. There

$$S ::= \{\text{inst } C^*,\ \text{tab } n^*,\ \text{mem } m^*\}$$

$\boxed{\vdash s; v^*; e^*}$

$$\frac{\vdash s : S \qquad S; \epsilon \vdash_i v^*; e^* : ti^*; l; \phi}{\vdash_i s; v^*; e^* : ti^*; l; \phi}\ \textsc{Program}$$

$\boxed{S; (ti^*; l; \phi)^? \vdash_i v^*; e^* : ti^*; l; \phi}$

$$\frac{(\vdash v : (t\ a); \phi_v)^* \qquad C = S_{\text{inst}}(i), \text{local } t^*, \text{return } (ti^n; l; \phi)^? \qquad S; C \vdash e^* : \epsilon : (t\ a)^*; \phi_v^* \to ti^n; l; \phi}{S; (ti^n; l; \phi)^? \vdash_i v^*; e^* : ti^n; l; \phi}\ \textsc{Code}$$

Figure 4.1: Wasm-prechk Program Typing Rules

are many different judgments being introduced, so we explicitly state the form of the judgment before stating the rule for that judgment.

During reduction, we use Rule Program (Figure 4.1) to ensure that a Wasm-prechk program state (consisting of the store s, local variables v^*, and instruction sequence e^*) is well typed (notice that it has the same form as the reduction relation). It uses Rule Code and relies on the store being well-typed (Rule Store in Figure 4.2), to ensure that a reducible Wasm-prechk program is well typed. Rule Code checks that a sequence of instructions is well typed with an empty stack, the indexed types and constraints for the given local variables in the precondition, and an optional return postcondition (not used by Rule Program). Since local variables are values, we know that each one of them is equal to some constant, so Rule Code is really just checking that the sequence of instructions has some postcondition reachable from the given local variables. There is an optional return postcondition for Rule Code because the typing rule for local blocks (as seen in Rule Local in Figure 4.3) has as a premise a judgment of the exactly same form, except with a return postcondition.

In addition to getting the type of the instructions being reduced, we also

need to know the type of the store s since it is part of the reduction relation. Rule STORE checks that a run-time store, s is well typed by the store context S. The store context S is to s as C is to $inst$. That is, it contains the type information for everything in s. Rule STORE ensures that every module instance $inst$ in s has the type of the index module context C in S using Rule INSTANCE. Further, Rule STORE ensures that all of the closures in all of the tables in s are well typed, and the the sizes of all the tables and memory chunks in S do not exceed the actual size of their implementations.

To get the type of the store, we in turn have to know the types of each of the various run-time data structures. Rule INSTANCE checks that a module instance is well-typed by the index module context under the store context S. It checks all of the closures cl^* against their expected types tfi^* in C, and similarly for all of the globals (v^* and $(\text{mut}^?\ t)^*$). The table and memory indices (i and j, respectively) are used to look up the the relevant types ((n, tfi^*) and m, respectively) in the store context S. Closures are typechecked by Rule CLOSURE, which falls back on the module typing rules from Figure 3.8 to typecheck the function definition inside of the closure. Rule ADMIN-CONST gets the postcondition indexed types and constraints on values; it is used to typecheck local and global variables.

Now we will introduce the typing rules for administrative instructions, and the administrative typing judgment in Figure 4.3. The administrative typing judgment $S; C \vdash e^* : tfi$ extends the Wasm-prechk typing rules for instructions to include administrative instructions and the store context S. Every rule of the judgment $C \vdash e^* : tfi$ (recall the rules enumerated in section 3.3) is implicitly added to the administrative judgment by accepting any S.

Most of the rules for administrative instructions check against extra information provided by the administrative typing judgment. Rule LOCAL typechecks a local block using Rule CODE to ensure that the body e^* is well typed with the indexed types and constraints for local variables provided by the local block as the precondition and any postcondition. Since local blocks are inline expansions of function calls, we use the optional return postcondition functionality of Rule CODE to ensure that returning from

$\boxed{\vdash s : S}$

$$\frac{\begin{array}{c} S = \{\text{inst } C^*, \text{tab } n^*, \text{mem } m^*\} \\ (S \vdash inst : C)^* \qquad ((S \vdash cl : tfi)^*)^* \qquad (n \leq |cl^*|)^* \qquad (m \leq |b^*|)^* \end{array}}{\vdash \{\text{inst } inst^*, \text{tab } (cl^*)^*, \text{mem } (b^*)^*\} : S} \textsc{Store}$$

$\boxed{S \vdash inst : C}$

$$\frac{\begin{array}{cc} (S \vdash cl : tfi)^* & (\vdash v : (t\ a), \phi_v)^* \\ (S_{\text{tab}}(i) = n)^? & (S_{\text{mem}}(j) = m)^? \end{array}}{\begin{array}{l} S \vdash \{\text{func } cl^*, \text{glob } v^*, \text{tab } i^?, \text{mem } j^?\} \\ \quad : \{\text{func } tfi^*, \text{global } (\text{mut}^?\ t)^*, \text{table } n^?, \text{memory } m^?\} \end{array}} \textsc{Instance}$$

$\boxed{\vdash v : ti; \phi}$

$$\frac{}{\vdash t.\mathsf{const}\ c : (t\ a); \circ, (t\ a), (\mathsf{eq}\ a\ (t\ c))} \textsc{Admin-Const}$$

$\boxed{S \vdash cl : tfi}$

$$\frac{S_{\text{inst}}(i) \vdash f : tfi}{S \vdash \{\text{inst } i, \text{code } f\} : tfi} \textsc{Closure}$$

Figure 4.2: Wasm-prechk Store Typing Rules

inside the local block will be well typed. Rule Call-Cl typechecks calling a closure by ensuring that the closure cl being called has the same type as the call instruction call cl in S. Rule Trap is always well typed under any precondition and postcondition. Rule Label typechecks the body of the label block with the precondition of the saved instructions pushed onto the label stack. If the label was generated by a loop, then the precondition of the saved values is the precondition of the loop, and we know the loop is well typed. Otherwise, the saved instructions will be an empty sequence and will be well typed from the precondition.

$\boxed{S; C \vdash e^* : tfi}$

$$\frac{S; (ti^n; l_2; \phi_2) \vdash_i v_l^*; e^* : ti^n; l_2; \phi_2}{S; C \vdash \mathsf{local}\{i; v_l^*\}\ e^*\ \mathsf{end} : \epsilon; l_1; \phi_1 \to ti^n; l_1; \phi_1, \phi_2}\ \textsc{Local}$$

$$\frac{S \vdash cl : tfi}{S; C \vdash \mathsf{call}\ cl : tfi}\ \textsc{Call-Cl} \qquad \frac{}{S; C \vdash \mathsf{trap} : tfi}\ \textsc{Trap}$$

$$\frac{S; C \vdash e_0^* : ti_3^*; l_3; \phi_3 \to ti_2^*; l_2; \phi_2 \quad S; C, \text{label}\ (ti_3^*; l_3; \phi_3) \vdash e^* : \epsilon; l_1; \phi_1 \to ti_2^*; l_2; \phi_2}{S; C \vdash \mathsf{label}\{e_0^*\}\ e^*\ \mathsf{end} : \epsilon; l_1; \phi_1 \to ti_2^*; l_2; \phi_2}\ \textsc{Label}$$

Figure 4.3: Wasm-prechk Administrative Instruction Rules

Given these additional typing judgments and rules, we can now show the metatheoretic properties mentioned above.

4.2 Relationship Between Wasm and Wasm-prechk

We want to show two properties about the relationship between Wasm and Wasm-prechk. First, we want Wasm-prechk to be backwards compatible with Wasm. It should be possible to convert well-typed Wasm programs into well-typed Wasm-prechk programs with no additional developer effort. We demonstrate a simple yet naive way of embedding Wasm programs into Wasm-prechk in subsection 4.2.1. Second, we want to show that well-typed Wasm-prechk programs can be turned into Wasm programs. This is accomplished in subsection 4.2.2 using an erasure function that turns Wasm-prechk programs and types into Wasm programs and types.

4.2.1 Embedding Wasm in Wasm-prechk

We present a way to embed Wasm programs in Wasm-prechk. The embedding function takes a Wasm program and replaces all of the type annotations with indexed function types that have no constraints on the variables. Intuitively,

the type annotations are the only part of the surface syntax of Wasm that isn't in Wasm-prechk, so we must figure out a way to bring it over. While this embedding requires no additional developer effort, it provides no information to the indexed type system beyond what can be inferred from the instructions in the program. We conjecture that a well typed Wasm program embedded in Wasm-prechk is also well typed, but we have not proved it.

We typeset Wasm-prechk instructions in a blue sans serif font and Wasm instruction in a **bold red font** to set them apart.

Conjecture 1. Well Typed Wasm Programs Embedded in Wasm-prechk are Well Typed

If $\vdash$ **module** $f^*\ glob^*\ tab^?\ mem^?$,
then $\vdash embed_{module}($**module** $f^*\ glob^*\ tab^?\ mem^?)$

Embedding works purely over the surface syntax of the languages. As such, we define embedding over modules: the pinnacle syntactic objects of both the Wasm and Wasm-prechk surface syntax hierarchies. Embedding a module *module* means embedding all of the functions f^* in the module, and embedding the table *tab* parameterized with all of the function definitions f^*. We do not have to embed globals $glob^*$ or the memory $mem^?$ as they have the same syntax in both Wasm and Wasm-prechk. We explain how to embed tables *tab* in Definition 2, and functions f in Definition 3.

Definition 1. $\boxed{embed_{module}(module)^C = \text{module}}$

$$
\begin{aligned}
embed_{module}(\textbf{module}\ f^*\ glob^*\ tab^?\ mem^?) \quad = \quad & \text{module}\ embed_f(f)^* \\
& \quad glob^* \\
& \quad embed_{tab}(tab^?)^{f^*} \\
& \quad mem^?
\end{aligned}
$$

Tables in Wasm-prechk must also provide the indexed function types of all the functions they contain, so to embed them we must include those types. We do this by parameterizing the embedding of the table *tab* with all of the declared functions f^*. Then, we retrieve the indexed function type $ti_1; l_1; \phi_1 \to ti_2; l_2; \phi_2$ of the function pointed to by the function index i in

f^* for every function index i in the table. We cannot embed imported tables because we have no way of accessing the types of the functions included in the table.

Definition 2. $\boxed{embed_{tab}(tab)^{f^*} = \mathsf{tab}}$

$$\begin{array}{rcl} embed_{tab}(\textbf{tab}\ n\ i^n) & = & \mathsf{tab}\ n\ (ti_1;\ l_1;\ \phi_1 \to ti_2;\ l_2;\ \phi_2)^n \\ & & \text{where } \forall i. f^*(i) = \mathsf{func}\ ti_1;\ l_1;\ \phi_1 \to ti_2;\ l_2;\ \phi_2 \\ & & \qquad\qquad \mathsf{local}\ t^*\ e^* \end{array}$$

The embedding of functions, Definition 3, both must construct an indexed function type for itself and embed its body. Function bodies have their local variables defined by the function that they are enclosed in. Thus, when the function body is embedded we pass the local types $(t_1^*\ t^*)$ so the body knows how to constrain local variables. We construct an indexed function type that has the precondition of the expected values on the stack turned into indexed types using fresh index variables and the types t_1^* from the Wasm type, and do the same with the postcondition and t_2^*. We cannot embed imported functions because we have no way of accessing the types of the local variables of the function.

Definition 3. $\boxed{embed_f(f) = \mathsf{f}}$

$$\begin{array}{rcl} embed_f(\textbf{func}\ (t_1^* \to t_2^*) & = & \mathsf{func}\ ((t_1\ a_1)^*;\ \epsilon;\ (\circ, (t_1\ a_1)^*) \\ \qquad \textbf{local}\ t^*\ e^*) & & \qquad \to (t_2\ a_2)^*;\ (t_1\ a_3)^*\ (t\ a_4)^*; \\ & & \qquad\qquad (\circ, (t_2\ a_2)^*, (t_1\ a_3)^* (t\ a_4)^*)) \\ & & \qquad t^*\ (embed_e(e)^{(t_1^*\ t^*)})^* \\ & & \mathsf{end} \end{array}$$

Embedding instructions replaces all function types used within the Wasm syntax with Wasm-prechk indexed function types, and adds the function types for all of the functions in a table to the table's type declaration. This occurs within blocks and indirect function calls, as shown in Definition 4. The indexed types simply have fresh index variables that are different in the precondition and postcondition, and the primitive types for the stack are

known from the Wasm type $t_1^* \to t_2^*$. To know what the local variables are, we parameterize the embedding over the types of local variables (t^*).

Definition 4. $\boxed{embed_e(e)^{t^*} = \mathsf{e}}$

$$
\begin{aligned}
embed_{e^*}(\textbf{block}\ (t_1^* \to t_2^*)\ e^*\ \textbf{end})^{t^*} &= \mathsf{block}\ ((t_1\ a_1)^*; (t\ a_3)^*; (\circ, (t_1\ a_1)^*, (t\ a_3)^*) \to (t_2\ a_2)^*; (t\ a_4)^*; (\circ, (t_2\ a_2)^*, (t\ a_4)^*))\ embed_{e^*}(e^*)^{t^*}\ \mathsf{end} \\
embed_{e^*}(\textbf{loop}\ (t_1^* \to t_2^*)\ e^*\ \textbf{end})^{t^*} &= \mathsf{loop}\ ((t_1\ a_1)^*; (t\ a_3)^*; (\circ, (t_1\ a_1)^*, (t\ a_3)^*) \to (t_2\ a_2)^*; (t\ a_4)^*; (\circ, (t_2\ a_2)^*, (t\ a_4)^*))\ embed_{e^*}(e^*)^{t^*}\ \mathsf{end} \\
embed_{e^*}(\textbf{if}\ (t_1^* \to t_2^*)\ e^*\ \textbf{end})^{t^*} &= \mathsf{if}\ ((t_1\ a_1)^*; (t\ a_3)^*; (\circ, (t_1\ a_1)^*, (t\ a_3)^*) \to (t_2\ a_2)^*; (t\ a_4)^*; (\circ, (t_2\ a_2)^*, (t\ a_4)^*))\ embed_e(e_1^*)^{t^*}\ embed_e(e_2^*)^{t^*}\ \mathsf{end} \\
embed_{e^*}(\textbf{call\_indirect}\ (t_1^* \to t_2^*))^{t^*} &= \mathsf{call\_indirect}\ ((t_1\ a_1)^*; (t\ a_3)^*; (\circ, (t_1\ a_1)^*, (t\ a_3)^*) \to (t_2\ a_2)^*; (t\ a_4)^*; (\circ, (t_2\ a_2)^*, (t\ a_4)^*)) \\
embed_{e^*}(e)^{t^*} &= e, \text{ otherwise} \\
embed_{e^*}(e^*)^{t^*} &= (embed_{e^*}(e)^{t^*})^*
\end{aligned}
$$

These are not the only differences in the surface syntax between Wasm and Wasm-prechk: we also introduced four new instructions (the *prechk*-tagged instructions). The definition of embedding we have introduced has been entirely syntactic, but that will not work for replacing non-*prechk*-tagged instructions with *prechk*-tagged versions during embedding since we must be able to ensure that stronger guarantees are met. Thus, we do not have an explicit embedding that provides *prechk*-tagged instructions, though we do posit the existence of a trivial embedding that would provide *prechk*-tagged

instructions. One could, for example, check at every div, call_indirect , load , and store whether the *prechk*-tagged version of the instruction is well typed, and only if it is well typed replace the instruction with the *prechk*-tagged version. However, a more sophisticated static analysis could provide more precise type annotations and therefore potentially allow even more check eliminations.

4.2.2 Erasing Wasm-prechk to Wasm

We provide an erasure function for Wasm-prechk that transforms Wasm-prechk programs into Wasm programs by discarding the extra information from the Wasm-prechk type system and replacing *prechk*-tagged instructions with their non-tagged counterparts. Erasure is useful in the type safety proof because it lets us reuse much of the proof of progress from Wasm (see subsection 4.3.2). Therefore, we define erasure not just for the surface syntax, like we did for embedding, but also for typing constructs (such as the module type context), administrative instructions, and runtime data structures (such as the store). We show that erasing a well-typed Wasm-prechk program produces a well-typed Wasm program.

As with the presentation of the embedding, we typeset Wasm-prechk instructions in a blue sans serif font and Wasm instruction in a **bold red font**.

Erasing Surface Syntax As with embedding, we start by defining erasure with the pinnacle syntactic object: the module. Defining and erasure for modules relies on the erasure of tables and functions, and therefore instructions and indexed function types. Keep in mind that the proofs of sound erasure work over the typing rules for these constructs, so we also define erasure of module type contexts since they are used in the typing rules for modules.

Erasing a module erases all of the functions f^* and the table $tab^?$. The globals $glob^*$ and optional memory $mem^?$ both have the same syntax in Wasm-prechk as in Wasm.

Definition 5. $\boxed{erase_{module}(module) = \textbf{module}}$

$$
\begin{array}{lcl}
erase_{module}(\text{module } f^* \; glob^* \; tab^? \; mem^?) & = & \textbf{module } erase_f(f)^* \\
& & \quad glob^* \\
& & \quad erase_{tab}(tab)^? \\
& & \quad mem^?
\end{array}
$$

We show that erasing a well-typed Wasm-prechk module yields a well-typed Wasm module.

Theorem 1. Sound Module Erasure

If $\vdash \text{module } f^* \; glob^* \; (n, tfi^n)^? \; mem^?$,
then $\vdash \textbf{module } erase_f(f)^* glob^* n^? mem^?$

Proof. Note that the globals $glob^*$ and memory $mem^?$ are not affected by erasure, and have the same module typing rules in Wasm as in Wasm-prechk. Thus, we only need to reason about the functions f^* and table $tab^?$.

Then, by Lemma Sound Function Typing Erasure and Lemma Sound Table Erasure, we have that $\vdash \textbf{module } erase_f(f)^* glob^* n^? mem^?$. □

Erasing a table definition $\text{table } n \; i^n$ does nothing, since a table definition has the same syntax in Wasm-prechk and in Wasm. However, erasing an imported table declaration $\text{table } (n, tfi^n) \; im$ must get rid of the indexed function types tfi^n. We do not use or care about the exports, since they are unchanged and only used for linking, so we omit them.

Definition 6. $\boxed{erase_{tab}(tab) = \textbf{tab}}$

$$
\begin{array}{rcl}
\text{table } n \; i^n & = & \textbf{table } n \; i^n \\
\text{table } (n, tfi^n) \; im & = & \textbf{table } n \; im
\end{array}
$$

We show that erasure on well typed Wasm-prechk tables tab is sound with respect to Wasm's type system. This proof relies on the definition of $erase_C(C)$: Definition 10.

Lemma 1. Sound Table Erasure

If $C \vdash tab$,

then $erase_C(C) \vdash erase_{tab}(tab)$

Proof. By case analysis on $erase_{tab}(tab)$.

- Case: $tab = \mathsf{table}\ n\ i^n$

  We know $(C_{\text{func}}(i) = tfi)^n$ because it is a premise of Rule Table.

  Then, $(erase_C(C)_{\text{func}}(i) = tfi)^n$ by definition of $erase_C(C)$.

  Therefore, $erase_C(C) \vdash \mathbf{table}\ n\ i^n$ because Wasm accepts any module type context in that rule.

- Case: $tab = \mathsf{table}\ (n, tfi^n)\ im$

  Trivially $erase_C(C) \vdash \mathbf{table}\ n\ im$ because Wasm accepts any module type context and imported table.

□

To erase a function definition f, we erase both the type declaration $ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$ and the body e^*. We can also erase an imported function by erasing the declared type tfi.

Definition 7. $\boxed{erase_f(f) = \mathbf{f}}$

$$
\begin{array}{lcl}
erase_f(\mathsf{func}\ ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2\ \mathsf{local}\ t^*\ e^*) & = & \mathbf{func}\ erase_{tfi}(ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2)\ \mathbf{local}\ t^*\ erase_{e^*}(e^*) \\
erase_f(\mathsf{func}\ ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2\ im) & = & \mathbf{func}\ erase_{tfi}(ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2)\ im
\end{array}
$$

We show that erasing a Wasm-prechk function f, that is well typed under a module type context C, produces a Wasm function $erase_f(f)$ that is well typed under the erased module type context $erase_C(C)$. This is useful not just for erasing the surface syntax, but also because functions are a part of

closures which are used at run time (as part of module instances and tables). The proof relies on Lemma Sound Static Typing Erasure to prove that the body is still well typed. The case of imported functions is trivial because an imported function is well typed under absolutely any context and with any function type, so it is omitted.

Lemma 2. Sound Function Typing Erasure

If

$$C \vdash \mathsf{func}\ (t_1\ a_1)^*;\ l_1;\ \phi_1 \to (t_2\ a_2);\ l_2;\ \phi_2 \\ \mathsf{local}\ t^*\ e^* : ex^*\ (t_1\ a_1)^*;\ l_1;\ \phi_1 \to (t_2\ a_2);\ l_2;\ \phi_2$$

then

$$erase_C(C) \vdash erase_f(\mathsf{func}\ erase_{tfi}((t_1\ a_1)^*;\ l_1;\ \phi_1 \to (t_2\ a_2);\ l_2;\ \phi_2) \\ \mathsf{local}\ t^*\ erase_{e^*}(e^*)) \\ : ex^*\ erase_{tfi}((t_1\ a_1)^*;\ l_1;\ \phi_1 \to (t_2\ a_2);\ l_2;\ \phi_2)$$

Proof. We must show that

$$erase_C(C, \mathrm{local}(t_1^*\ t^*), \mathrm{label}((t_2\ a_2);\ l_2;\ \phi_2), \mathrm{return}((t_2\ a_2);\ l_2;\ \phi_2)) \\ \vdash erase_{e^*}(e^*) \\ : erase_{tfi}((t_1\ a_1)^*;\ l_1;\ \phi_1 \to (t_2\ a_2);\ l_2;\ \phi_2)$$

since it is the only premise of typechecking a function definition in Wasm.

We know the following because it is a premise of Rule Func which we have assumed to hold.

$$C, \mathrm{local}(t_1^*\ t^*), \mathrm{label}((t_2\ a_2);\ l_2;\ \phi_2), \mathrm{return}((t_2\ a_2);\ l_2;\ \phi_2) \\ \vdash e^* \\ : (t_1\ a_1)^*;\ l_1;\ \phi_1 \to (t_2\ a_2);\ l_2;\ \phi_2$$

Then, by Lemma Sound Static Typing Erasure, we have that

$$erase_C(C, \mathrm{local}(t_1^*\ t^*), \mathrm{label}((t_2\ a_2);\ l_2;\ \phi_2), \mathrm{return}((t_2\ a_2);\ l_2;\ \phi_2)) \\ \vdash erase_{e^*}(e^*) \\ : erase_{tfi}((t_1\ a_1)^*;\ l_1;\ \phi_1 \to (t_2\ a_2);\ l_2;\ \phi_2)$$

□

Erasing an indexed type function keeps only the primitive Wasm types (t_1^* and t_2^*) from the indexed types representing the stack ($(t_1\ a_1)^*$ and $(t_2\ a_2)^*$), and discards everything else.

Definition 8. $\boxed{erase_{tfi}(tfi) = \mathbf{tf}}$

$erase_{tfi}((t_1\ a_1)^*;\ l_1;\ \phi_1 \to (t_2\ a_2)^*;\ l_2;\ \phi_2) = t_1^* \to t_2^*$

Erasing instructions involves erasing the indexed function types for every instruction that includes it as part of their syntax (blocks and indirect function calls). We must also remove the *prechk* tag from *prechk*-tagged instructions to turn them into instructions that exist in Wasm.

Definition 9. $\boxed{erase_{e^*}(e) = \mathbf{e}}$

$$
\begin{array}{rcl}
erase_{e^*}(\mathsf{block}\ tfi\ e^*\ \mathsf{end}) & = & \mathbf{block}\ erase_{tfi}(tfi) \\
 & & \quad erase_{e^*}(e^*) \\
 & & \mathbf{end} \\
erase_{e^*}(\mathsf{loop}\ tfi\ e^*\ \mathsf{end}) & = & \mathbf{loop}\ erase_{tfi}(tfi) \\
 & & \quad erase_{e^*}(e^*) \\
 & & \mathbf{end} \\
erase_{e^*}(\mathsf{if}\ tfi\ e_1^*\ e_2^*\ \mathsf{end}) & = & \mathbf{if}\ erase_{tfi}(tfi) \\
 & & \quad erase_{e^*}(e_1^*) \\
 & & \quad erase_{e^*}(e_2^*) \\
 & & \mathbf{end} \\
erase_{e^*}(\mathsf{label}_n\ \{e_0^*\}\ e^*\ \mathsf{end}) & = & \mathbf{label}_n\ \{erase_{e^*}(e_0^*)\} \\
 & & \quad erase_{e^*}(e^*) \\
 & & \mathbf{end} \\
erase_{e^*}(\mathsf{local}_n\ \{i;\ v^*\}\ e^*\ \mathsf{end}) & = & \mathbf{local}_n\ \{i;\ v^*\} \\
 & & \quad erase_{e^*}(e^*) \\
 & & \mathbf{end} \\
erase_{e^*}(\mathsf{call\_indirect}\ tfi) & = & \mathbf{call\_indirect}\ erase_{tfi}(tfi) \\
erase_{e^*}(t.\mathsf{div}_{\mathsf{prechk}}) & = & t.\mathbf{div} \\
erase_{e^*}(t.\mathsf{call\_indirect}_{\mathsf{prechk}}) & = & t.\mathbf{call\_indirect} \\
erase_{e^*}(t.\mathsf{store}_{\mathsf{prechk}}\ tp^?\ align\ o) & = & t.\mathbf{store}\ tp^?\ align\ o \\
erase_{e^*}(t.\mathsf{load}_{\mathsf{prechk}}\ (tp\_sx)^?\ align\ o) & = & t.\mathbf{load}\ (tp\_sx)^?\ align\ o \\
erase_{e^*}(e) & = & e, \text{otherwise} \\
erase_{e^*}(e^*) & = & erase_{e^*}(e)^*
\end{array}
$$

Erasing Typing Constructs Here, we prove that erasing a Wasm-prechk static typing derivation is sound with respect to Wasm's type system. This means that erasure on the Wasm-prechk static typing judgment is sound with respect to Wasm's type system. Specifically, a Wasm-prechk instruction sequence e^*, that is well typed under a module type context C, produces a Wasm instruction sequence $e'^* = erase_{e^*}(e^*)$ that is well typed under the erased module type context $C' = erase_C(C)$.

Lemma 3. Sound Static Typing Erasure

If $C \vdash e^* : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$,
then $erase_C(C) \vdash erase_{e^*}(e^*) : erase_{tfi}(ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2)$

Proof. We proceed by induction over typing derivations. Most proof cases are omitted as they are simple, but we provide a few to give an idea of what the proofs look like. Intuitively, we want to show that erasing the typing derivation produces a valid Wasm typing derivation.

For most of the cases, the sequence of instructions e^* contains only a single instruction e_2, so we elide the step of turning $erase_{e^*}(e^*)$ into $erase_{e^*}(e_2)$.

We present one Case below, the rest are in the appendix (B.1).

- Case: $C \vdash t.binop : (t\ a_1)\ (t\ a_2); l_1; \phi_1 \to (t\ a_3); l_1; \phi_1, (t\ a_3), (= a_3\ (binop\ a_1\ a_2))$

  We want to show that

  $$erase_C(C) \vdash erase_{e^*}(t.binop) : erase_{tfi}((t\ a_1)\ (t\ a_2); l_1; \phi_1 \to (t\ a_3); l_1; \phi_1, (t\ a_3), (= a_3\ (binop\ a_1\ a_2)))$$

  By the definition of $erase_e$, we want to show that $erase_C(C) \vdash t.binop : t\ t \to t$ is valid in Wasm.

  Trivially, we have $erase_C(C) \vdash t.binop : t\ t \to t$ by Rule Wasm-Binop, since Rule Wasm-Binop works under any module type context.

□

To erase a module type context, we must erase all of the function types tfi^*, the table type (n, tfi_2^*) if one is present, and the postconditions in the label stack $((t_1\ a_1)^*; l_1; \phi_1)^*$ and the return stack $((t_2\ a_2)^*; l_2; \phi_2)^?$. We erase postconditions the same way we erase the postconditions of indexed function types: by keeping only the primitive Wasm types (t_1^* in the case of a label postcondition). Recall that erasing a table type means discarding the type information about the functions in the table.

Definition 10. $\boxed{erase_C(C) = \mathbf{C}}$

$$
\begin{array}{lcl}
erase_C(\{\text{func } tfi^*,\ \text{global } tg^*, & = & \{\text{func } erase_{tfi}(tfi^*), \\
\quad \text{table } (n, tfi_2^*)^?, & & \quad \text{global } tg^*,\ \text{table } n^?, \\
\quad \text{memory } m^?,\ \text{local } t^*, & & \quad \text{memory } m^?, \text{local } t^*, \\
\quad \text{label } ((t_1\ a_1)^*;\ l_1;\ \phi_1)^*, & & \quad \text{label } (t_1^*)^*,\ \text{return } (t_2^*)^?\} \\
\quad \text{return } ((t_2\ a_2)^*;\ l_2;\ \phi_2)^?\}) & &
\end{array}
$$

Erasing Programs Defining and erasure for programs relies on the erasure of the store and its various structures, as well as the erasure of instructions which we have already defined and proven. Remember, the proofs of sound erasure work over the typing rules for these constructs, so we have to show sound erasure for all of the various typing rules that Rule Program relies on.

Now we will show that erasing a well-typed Wasm-prechk program in reduction form $(s;\ v^*;\ e^*)$ is sound with respect to Wasm's type system. Intuitively, we accomplish this by showing that erasing typing derivations of the Rule Program judgment produce valid Wasm typing derivations, like in Lemma Sound Static Typing Erasure. To do so, we must show sound erasure for Rule Code, as it is a premise of Rule Program; this is done by Lemma Sound Code Typing Erasure. Erasing programs involves erasing many run-time data structures, including the store s and store context S, as well as modules instances $inst^*$ in s, and closures cl in modules instances and the optional table. Erasing the store is shown to be safe by Lemma Sound Store Erasure.

Theorem 2. Sound Program Typing Erasure

If $\vdash_i s;\ v^*\ e^* : (t_2\ a_2)^*;\ l_2;\ \phi_2$,
then $\vdash_i erase_s(s);\ v^*;\ erase_{e^*}(e^*) : t_2^*$

Proof. We must show that $\vdash erase_s(s) : S$ for some Wasm store context S, and that $erase_S(S);\ \vdash_i erase_{e^*}(e^*) : erase_{tfi}(ti_1^*;\ l_1;\ \phi_1 \to ti_2^*;\ l_2;\ \phi_2)$.

We have $\vdash s : S'$, where S' is a Wasm-prechk store context, because it is a premise of Rule Program which we have assumed to hold.

Then, $\vdash erase_s(s) : erase_S(S')$ is valid by Lemma Sound Store Erasure. Since $erase_S(S')$ is a Wasm store context, we have that $\vdash erase(s) : \S$ for some Wasm store context S where $S = erase_S(S')$.

We also have $S;\ \vdash_i v^*\ e^* : \epsilon;\ l_1;\ \phi_1 \to (t_2\ a_2)^*;\ l_2;\ \phi_2$ as a premise of Rule Program.

In which case we have $erase_S(S);\ \vdash_i erase_{e^*}(e^*) : erase_{tfi}(ti_1^*;\ l_1;\ \phi_1 \to ti_2^*;\ l_2;\ \phi_2)$ by Lemma Sound Code Typing Erasure. □

The sound erasure of Rule Code is used in the sound erasure of programs. Thus, we only prove the case when the optional return stack $((t_2\ a_2)^*;\ l_2;\ \phi_2)^?$ is empty because we are only proving this to use later in Rule Program, which never uses the return stack. Lemma Sound Code Typing Erasure relies on Lemma Sound Admin Typing Erasure, which shows a similar property, but for the administrative typing judgment $S;\ C \vdash e^* : tfi$.

Lemma 4. Sound Code Typing Erasure

If $S;\ \vdash_i v^*\ e^* : \epsilon;\ l_1;\ \phi_1 \to (t_2\ a_2)^*;\ l_2;\ \phi_2$,
then $erase_S(S);\ \vdash_i erase_{e^*}(e^*) : erase_{tfi}(ti_1^*;\ l_1;\ \phi_1 \to ti_2^*;\ l_2;\ \phi_2)$

Proof. We must show that $(\vdash v : t_v)^*$ and

$$\begin{aligned} erase_S(S);\ erase_C(S_{\text{inst}}(i), \text{local } t_v^* \\ \vdash erase_{e^*}(e^*) \\ : erase_{tfi}(\epsilon;\ l_1;\ \phi_1 \to (t_2\ a_2)^n;\ l_2;\ \phi_2) \end{aligned}$$

We have $(\vdash v : t_v)^*$ trivially since it is a premise of Rule Code which we have assumed to hold.

We also have $S;\ S_{\text{inst}}(i), \text{local } t_v^* \vdash e^* : ti_1^*;\ l_1;\ \phi_1 \to ti_2^*;\ l_2;\ \phi_2$ because it too is a premise of Rule Code.

Then, by Lemma Sound Admin Typing Erasure, we have that

$$\begin{aligned} erase_S(S);\ erase_C(S_{\text{inst}}(i), \text{local } t_v^* \\ \vdash erase_{e^*}(e^*) \\ : erase_{tfi}(\epsilon;\ l_1;\ \phi_1 \to (t_2\ a_2)^n;\ l_2;\ \phi_2) \end{aligned}$$

□

Lemma Sound Admin Typing Erasure builds on Lemma Sound Static Typing Erasure by adding the store context S and typing rules for administrative instructions. It is necessary to add these rules and extra information because they are used for typechecking programs. Note that while we add S to the judgment used in Lemma Sound Static Typing Erasure to get $S; C \vdash e^*; tfi$, none of the rules previously proven reference S in any way, they simply match any store context.

Lemma 5. Sound Admin Typing Erasure

If $S; C \vdash e^* : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$,
then $erase_S(S); erase_C(C) \vdash erase_{e^*}(e^*) : erase_{tfi}(ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2)$

Proof. We proceed by induction over typing rules. In addition to the prior cases from Lemma Sound Static Typing Erasure, which trivially still hold since the value of S does not matter to those rules, we add proves for a few administrative typing rules, which may refer to S. Again, several proof cases are omitted as they are simple.

We present one Case below, the rest are in the appendix (B.1).

- $S; C \vdash \mathsf{label}_n\{e_0^*\}\ e^*\ \mathsf{end} : \epsilon; l_1; \phi_1 \to (t_2\ a_2)^n; l_2; \phi_2$

  We must show that

  $$erase_S(S); erase_C(C) \vdash erase_{e^*}(e_0^*) : erase_{tfi}((t_3\ a_3)^*; l_3; \phi_3 \to (t_2\ a_2)^n; l_2; \phi_2)$$

  and

  $$erase_S(S); erase_C(C, \text{label}((t_3\ a_3)^*; l_3; \phi_3)) \vdash erase_{e^*}(e^*) : erase_{tfi}(\epsilon; l_1; \phi_1 \to (t_3\ a_3)^*; l_3; \phi_3)$$

  as they are the premises of typechecking a label block in Wasm.

  We have that $S; C \vdash e_0^* : (t_3\ a_3)^*; l_3; \phi_3 \to (t_2\ a_2)^n; l_2; \phi_2$ since it is a premise of Rule Label which we have assumed to hold.

Then, by the inductive hypothesis for the stored instructions e_0^* being well typed, we have that

$$erase_S(S);\ erase_C(C) \vdash erase_{e^*}(e_0^*) : erase_{tfi}((t_3\ a_3)^*;\ l_3;\ \phi_3 \to (t_2\ a_2)^*;\ l_2;\ \phi_2)$$

$S;\ C, \text{label}((t_3\ a_3)^*;\ l_3;\ \phi_3)) \vdash e^* : \epsilon;\ l_1;\ \phi_1 \to (t_3\ a_3)^*;\ l_3;\ \phi_3$, because it is a premise of Rule Label which we have assumed to hold.

By the inductive hypothesis for the body e^* being well typed, we have that

$$erase_S(S);\ erase_C(C, \text{label}((t_3\ a_3)^*;\ l_3;\ \phi_3)) \vdash erase_{e^*}(e^*) : erase_{tfi}(\epsilon;\ l_1;\ \phi_1 \to (t_3\ a_3)^*;\ l_3;\ \phi_3)$$

□

We must prove safe erasure about the store s for use in Theorem 2. First though, we must define erasure for s. Erasing the store erases all of the modules instances and closures in the tables inside the store. Note that in the definition we have expanded the definition of a table instance to $(\{\text{inst } i,\ \text{func } f\}^*)^*$ for extra clarity.

Definition 11. $\boxed{erase_s(s) = \text{s}}$

$$\begin{array}{lcl} erase_s(\{\text{inst } inst^*, & = & \{\text{inst } erase_{inst}(inst)^*, \\ \quad \text{tab } (\{\text{inst } i,\ \text{func } f\}^*)^*, & & \quad \text{tab } (\{\text{inst } i,\ \text{func } erase_f(f)\}^*)^*, \\ \quad \text{mem } meminst^*\}) & & \quad \text{mem } meminst^*\} \end{array}$$

Lemma Sound Store Erasure proves that erasing a well-typed Wasm-prechk store results in a well-typed Wasm store.

Lemma 6. Sound Store Erasure

If $\vdash s : S$, then $\vdash erase_s(s) : erase_S(S)$

Proof. Note that

$$\begin{aligned} s &= \{\text{inst } inst^*, \text{tab } (\{\text{inst } i,\ \text{func } f\}^*)^*,\ \text{mem } meminst^*\} \text{ and} \\ S &= \{\text{inst } C^*,\ \text{tab } (n, tfi^*)^*,\ \text{mem } m^*\} \end{aligned}$$

Then,

$$erase_S(S) = \{\text{func } erase_{tfi}(tfi)^*,\ \text{global } tg^*,\ \text{table } n,\ \text{memory } n^?, ...\}$$

by the definition of $erase_C$.

Then, we must prove the following properties, as they are the premises of $\vdash erase_s(s) : erase_S(S)$:

1. $(erase_S(S) \vdash erase_{inst}(inst) : erase_C(C))^*$

   We have that $(S \vdash inst : C)^*$, because it is a premise of Rule Store that we have assumed to hold.

   Then, we have $erase_S(S) \vdash erase_{inst}(inst) : erase_C(C))^*$ by Lemma Sound Instance Typing Erasure.

2. $((erase_S(S) \vdash \{\text{inst } i,\ \text{func } erase_f(f)\} : erase_{tfi}(tfi))^*)^*$

   We have $((S \vdash cl : tfi)^*)^*)$, because it is a premise of Rule Store that we have assumed to hold.

   Then, $((erase_S(S) \vdash \{\text{inst } i,\ \text{func } erase_f(f)\} : erase_{tfi}(tfi))^*)^*$ by Lemma Sound Closure Typing Erasure

3. $(n \leq |\{\text{inst } i,\ \text{func } erase_f(f)\}|)^*$

   We have that $(n \leq |\{\text{inst } i,\ \text{func } f\}|)^*$, because it is a premise of Rule Store that we have assumed to hold.

   Because the number of closures is not affected by erasure, we can then say that $(n \leq |\{\text{inst } i,\ \text{func } erase_f(f)\}|)^*$

4. $(m \leq |b^*|)^*$

   Trivially, we have that $(m \leq |b^*|)^*$, because it is a premise of Rule Store that we have assumed to hold.

□

Erasing a module instance erases all of the functions f in the closures (which we have expanded inline to $\{\text{inst } i,\ \text{func } f\}$) within the module instance.

Definition 12. $\boxed{erase_{inst}(inst) = \textbf{inst}}$

$$\begin{array}{rcl} erase_{inst}(\{\text{func } \{\text{inst } i,\ \text{func } f\}^*, & = & \{\text{func } \{\text{inst } i,\ \text{func } erase_f(f)\}^*, \\ \quad \text{global } v^*,\ \text{table } i^?, & & \quad \text{global } v^*,\ \text{table } i^?, \\ \quad \text{memory } j^?\}) & & \quad \text{memory } j^?\} \end{array}$$

We now prove that if a Wasm-prechk module instance $inst$ has type C under the store context S, then the erased Wasm instance $erase_{inst}(inst)$ will have the erased type $erase_c(C)$ under the erased store context $erase_S(S)$. To do this, we rely on the above lemmas to safely erase index information from function declarations and table declarations (globals and memory have the same type information in both Wasm-prechk and Wasm). This will be useful for proving that a well-typed Wasm-prechk store s erases to a well-typed Wasm store $erase_s(s)$ since stores contain many instances. To do this, we rely on the above lemmas to safely erase index type information about closures and tables (globals and memory have the same type information in both Wasm-prechk and Wasm).

Lemma 7. Sound Instance Typing Erasure If $S \vdash inst : C$, then $erase_S(S) \vdash erase_{inst}(inst) : erase_C(C)$

Proof. Note that

$$\begin{array}{rcl} S & = & \{\text{inst } C^*,\ \text{tab } (n, tfi^*)^*,\ \text{mem } m^*\} \\ inst & = & \{\text{func } \{\text{inst } i,\ \text{func } f\}^*, \text{global } v^*,\ \text{table } i^?, \text{memory } j^?\} \\ C & = & \{\text{func } tfi^*,\ \text{global } tg^*,\ \text{table } (n, tfi_2)^?,\ \text{memory } n^?, \dots\} \end{array}$$

Then,

$$erase_C(C) = \{\text{func } erase_{tfi}(tfi)^*,\ \text{global } tg^*,\ \text{table } n,\ \text{memory } n^?, ...\}$$

by the definition of $erase_C$.

Then, we must prove the following properties, as they are the premises of $erase_S(S) \vdash erase_{inst}(inst) : erase_C(C)$:

1. $erase_S(S) \vdash \{\text{inst } i,\ \text{func } f\} : erase_{tfi}(tfi))^*$

   We have that $S \vdash \{\text{inst } i,\ \text{func } f\} : tfi$, because it is a premise of Rule Instance that we have assumed to hold.

   Then, we have $erase_S(S) \vdash \{\text{inst } i,\ \text{func } f\} : erase_{tfi}(tfi))^*$ by Lemma Sound Closure Typing Erasure.

2. $(\vdash v : tg)^*$

   Trivially, this is a premise of $S \vdash inst : C$ and is not affected by erasure, so therefore it holds.

3. $erase_S(S)_{tab}(i) = n$

   $erase_S(S)_{tab}(i) = n$ by definition of $erase_S$.

   Therefore, $erase_S(S)_tab(i) = n$.

4. $erase_S(S)_mem(i) = n^?$

   Trivially, this is a premise of $S \vdash inst : C$ and is not affected by erasure, so therefore it holds.

□

We erase store contexts by erasing all of the module type instances C^* and table types $(n, tfi^*)^*$ within.

Definition 13. $\boxed{erase_S(S) = \mathbf{S}}$

$$erase_S(\{\text{inst } C^*,\ \text{tab } (n, tfi^*)^*,\ \text{mem } m^*\}) = \{\text{inst } erase_c(C)^*,\ \text{tab } n^*, \text{mem } m^*\}$$

Finally, we prove that if a Wasm-prechk closure is well typed than the erased closure is well typed.

Lemma 8. Sound Closure Typing Erasure

If $S \vdash \{\text{inst } i,\ \text{func } f\}^* : tfi$,
then $erase_S(S) \vdash \{\text{inst } i,\ \text{func } erase_f(f)\}^* : erase_{tfi}(tfi)$

Proof. We must show that $erase_S(S)_{\text{inst}}(i) \vdash erase_f(f) : erase_{tfi}(tfi)$.

We have $S_{\text{inst}}(i) \vdash f : tfi$ since it is a premise of Rule Closure which we have assumed to hold.

Then, $erase_S(S)_{\text{inst}}(i) \vdash erase_f(f) : erase_{tfi}(tfi)$ by Lemma Sound Function Typing Erasure. □

4.3 Type Safety

Type safety is the property that a well-typed program either reduces to another well-typed program, is an intentionally irreducible expression (in the case of Wasm-prechk, a sequence of values), or throws an error (trap, in the case of Wasm-prechk). Thus, type safety assures us that the behavior of a well-typed program is always well defined. The type safety of Wasm guarantees a number of important properties, including memory safety. Proving the type safety of Wasm-prechk gives us a high degree of assurance that it has the same level of safety as Wasm.

4.3.1 Subject Reduction

Subject reduction, also sometimes referred to as "type preservation", ensures that if a program has a specific type, then the program will have the same type after a reduction step. Before we present the subject reduction proof, we first introduce a number of useful lemmas.

Lemma Inversion tells us what typing rules can apply to a given Wasm-prechk instruction sequence, and therefore lets us reason about what the type of that sequence looks like. For example, if we have a typing derivation, D for $S; C \vdash t.\mathsf{const}\ c : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$, then we know that D must have at its base Rule Const, because that is the only way we have of typing constant instructions. D can also include any number of applications of Rule Subtyping and Rule Stack-Poly, because they can be applied to any well-typed sequence of instructions.

We do not know the exact types of instructions just from them being well typed, since the typing rules are non-deterministic. However, we can reason

about the general shape of the types given the base type on top of which Rule SUBTYPING and Rule STACK-POLY get applied. Additionally, Rule COMPOSITION can be used with the empty sequence and any well-typed single instruction. The addition of Rule COMPOSITION with the empty sequence is trivial because the postcondition of an empty instruction sequence must be immediately reachable from the precondition. Therefore the stack and local index store must be the same in both the precondition and postcondition of the empty sequence in the above case, and the postcondition index type context must be reachable from the precondition index type context.

Most cases of Lemma INVERSION are omitted. The complete definition can be found in the appendix (section B.2).

Lemma 9. INVERSION

- If $S; C \vdash t.\mathsf{const}\ c : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$, then $ti_2^* = ti_1^*\ (t\ a)$, $l_1 = l_2$, and $\phi_1, (t\ a), (=\ a\ (t\ c)) \implies \phi_2$.
- If $S; C \vdash t.binop : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$, then $ti_1^* = ti^*\ (t\ a_1)\ (t\ a_2)$, $ti_2^* = ti^*\ (t\ a_3)$, $l_1 = l_2$, and $\phi_1, (t\ a_3), (=\ a_3\ (binop\ a_1\ a_2)) \implies \phi_2$.
- If $S; C \vdash \mathsf{block}\ (ti_3^*; l_3; \phi_3 \to ti_4^m; l_4; \phi_4)$
  $e^*\ \mathsf{end} : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$
  then $ti_1^* = ti_0^*\ ti_3^*$, $ti_2^* = ti_0^*\ ti_4^m$, $l_1 = l_3$, $l_2 = l_4$, $\phi_1 \implies \phi_3, \phi_4 \implies \phi_2$, and $S; C, \mathrm{label}(ti_4^m; l_4; \phi_4) \vdash e^* : ti_3^*; l_3; \phi_3 \to ti_4^m; l_4; \phi_4$.
- If $S; C \vdash \mathsf{br}\ i : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$, then $ti_1^* = ti_3^*\ ti^*$, $C_{\mathrm{label}}(i) = ti^*; l_1; \phi_3$, and $\phi_1 \implies \phi_3$.
- If $S; C \vdash \mathsf{call\_indirect}\ ti_3^*; l_3; \phi_3 \to ti_4^*; l_4; \phi_4 : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$, then $ti_1^* = ti_0^*\ ti_3^*$, $ti_2^* = ti_0^*\ ti_4^*$, $l_2 = l_1$, $\phi_1 \implies \phi_3$, and $\phi_3, \phi_4 \implies \phi_2$.
- If $S; C \vdash e_1^*\ e_2 : ti_1^*; l_1; \phi_1 \to ti_3^*; l_3; \phi_3$, then $S; C \vdash e_1^* : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$, and $S; C \vdash e_2 : ti_2^*; l_2; \phi_2 \to ti_3^*; l_3; \phi_3$.

Proof. Proof omitted, but follows from induction over typing derivations. □

The next lemma, Lemma Lift-Consts, shows that if a sequence of constants, v^n, has a certain postcondition within a nested context, L^j, then it has the same postcondition outside of that context with the precondition of the context. We use this rule for branching and returning when we have some values v^n inside a reduction context L^j.

The intuition for the proof is that the nature of nested contexts are such that all of the instructions preceding v^n are values and therefore only add fresh index variables which are constrained to be equal to constants. Thus, we can pull v^n outside of the nested context and know that we can still get to the postcondition because we can add back in, using implication, all of the fresh index variables that we would have added from the values preceding.

Lemma 10. Lift-Consts

If $S; C \vdash v^n : \epsilon; l_3; \phi_3 \to ti^n; l_3; \phi_4$ is a subderivation of $S; C \vdash L^j[v^n] : s_1; l_1; \phi_1 \to s_2; l_2; \phi_2$,
then $S; C \vdash v^n : \epsilon; l_1; \phi_1 \to ti^n; l_3; \phi_4$ after reduction

Proof. By induction on j.

- Base case: $j = 0$

  We want to show that $S; C \vdash v^n : \epsilon; l_1; \phi_1 \to ti^n; l_3; \phi_4$ after reduction.

  We have $S; C \vdash v_0^* \; v^n \; e^* \; \mathsf{end} : s_1; l_1; \phi_1 \to s_2; l_2; \phi_2$ for some v_0^* and e^* by expanding L^0.

  Then, $S; C \vdash (t.\mathsf{const}\ c)^* : \epsilon; l_1; \phi_0 \to (t\ a)^*; l_1; \phi_0, (t\ a)^*, (\mathsf{eq}\ a\ (t\ c))$ where $v_0^* = (t.\mathsf{const}\ c)^*$ and $\phi_1 \implies \phi_0$ by Lemma Inversion on Rule Const.

  Further, $S; C \vdash v^n : \epsilon; l_3; \phi_0, (t\ a)^*, (\mathsf{eq}\ a\ (t\ c)) \to (t\ a)^*\ ti^n; l_3; \phi_4$, by Lemma Inversion on Rule Const.

  We now have all the information we need to show what we want to show.

  We know $\phi_0, (t\ a)^*, (\mathsf{eq}\ a\ (t\ c)) \implies \phi_3$.

Recall that $S; C \vdash v^n : \epsilon; l_3; \phi_0, (t\ a)^*, (\mathsf{eq}\ a\ (t\ c)) \to (t\ a)^*\ ti^n; l_3; \phi_4$, then

$$S; C \vdash v^n : (t\ a)^*; l_3; \phi_0, (t\ a)^*, (\mathsf{eq}\ a\ (t\ c)) \to (t\ a)^*, (t\ a)^*\ ti^n; l_3; \phi_4$$

by Rule SUBTYPING.

If v_0^* are not executed (*i.e.,* they are not part of the reduced expression), then a^* are fresh, so $\phi_0 \implies \phi_0, (t\ a)^*, (\mathsf{eq}\ a\ (t\ c))$, and therefore $S; C \vdash v^n : \epsilon; l_1; \phi_0 \to ti^n; l_1; \phi_4$ by Rule SUBTYPING and since $l_1 = l_3$.

Then, $S; C \vdash v^n : \epsilon; l_1; \phi_1 \to ti^n; l_1; \phi_4$ by *subtyping*.

- Induction case: $j = k + 1$

  We want to show that $S; C \vdash v^n : \epsilon; l_1; \phi_1 \to ti^n; l_3; \phi_4$ after reduction.

  We have $S; C \vdash \mathsf{label}_n\{e_0^*\}\ v_0^*\ L^k[v^n]\ e_1^*\ \mathsf{end} : s_1; l_1; \phi_1 \to s_2; l_2; \phi_2$ for some v_0^*, e_0^*, and e_1^* by expanding L^j.

  Then, $S; C \vdash (t.\mathsf{const}\ c)^* : \epsilon; l_1; \phi_0 \to (t\ a)^*; l_1; \phi_0, (t\ a)^*, (\mathsf{eq}\ a\ (t\ c))$ where $v_0^* = (t.\mathsf{const}\ c)^*$ and $\phi_1 \implies \phi_0$ by Lemma INVERSION on Rule CONST.

  Further, $S; C \vdash L^k[v^n] : (t\ a)^*; l_1; \phi_0, (t\ a)^*, (\mathsf{eq}\ a\ (t\ c)) \to s_5; l_5; \phi_5$ for some $s_5; l_5; \phi_5$ by Lemma INVERSION on Rule LABEL.

  Now we can prove want we wanted to show.

  We know $S; C \vdash v^n : \epsilon; l_1; \phi_0, (t\ a)^*, (\mathsf{eq}\ a\ (t\ c)) \to ti^n; l_1; \phi_4$ by the inductive hypothesis.

  If v_0^* are not executed (*i.e.,* after one reduction step), a^* are fresh, so $\phi_0 \implies (t\ a)^*, (\mathsf{eq}\ a\ (t\ c))$, and therefore $S; C \vdash v^n : \epsilon; l_1; \phi_0 \to (t\ a)^*\ ti^n; l_3; \phi_5$ by Rule SUBTYPING and since $l_1 = l_3$.

  Then, $S; C \vdash v^n : \epsilon; l_1; \phi_1 \to (t\ a)^*\ ti^n; l_3; \phi_3$ by Rule SUBTYPING.

□

Theorem 3. Subject Reduction

If $\vdash_i s; v^*; e^* : ti^*; l; \phi$ and $s; v^*; e^* \hookrightarrow_i s'; v'^*; e'^*$ then $\vdash_i s'; v'^*; e'^* : ti^*; l; \phi$.

Proof. We have $\vdash s : S$ and $S; \epsilon \vdash_i v^*; e^* : ti^*; l; \phi$ because they are premises of Rule Program.

Then, by Lemma Subject Reduction for Code, we have that $\vdash s' : S$ and $S; \epsilon \vdash_i v'^*; e'^* : ti^*; l; \phi$.

Thus, $\vdash s'; v'^*; e'^* : ti^*; l; \phi$ by Rule Program. □

Lemma Subject Reduction for Code proves subject reduction of the Rule Code typing rule. That is, it proves that if a sequence of instructions e^* and local variables v^* is typed by the Rule Code typing rule, then after a step of reduction the reduced instructions e'^* and locals v'^* will have the same postcondition $ti^*; l; \phi$. Further, if reduction modifies the store s, than the modified store s' will have the same type S.

In many reduction cases, there are values on the stack that get consumed by reducing an instruction. This creates a bit of a problem because those values represent intermediate state, and as such will introduce new index variables to the index type context in their postcondition. After reduction, the intermediate state is no longer present, so we lose those index variables from the postconditions.

For example, $(t.\mathsf{const}\ c)\ \mathsf{drop}$ could be typed as $\epsilon; l; \phi \to \epsilon; l; \phi, (t\ a), (=\ a\ (t\ c))$ where a represent the value on the stack $t.\mathsf{const}\ c$. This would reduce to ϵ, and then we lose the information about a in the postcondition index type system. However, this can be solved using implication, as we know a is fresh from the Rule Const, and therefore we allow saying $\phi \implies \phi, (t\ a), (=\ a\ (t\ c))$ after reduction. This pattern will appear in any case of the proof that consumes values.

Lemma 11. Subject Reduction for Code

If $S; (ti; l; \phi)^? \vdash v^*; e^* : ti; l; \phi, \vdash s : S$, (we omit this on rules that do not use the store) and $s; v^*; e^* \hookrightarrow s'; v'^*; e'^*$, then $S; (ti; l; \phi)^? \vdash v'^*; e'^* : ti; l; \phi$, and $\vdash s' : S$ (we omit this on rules that do not change the store)

Proof. By induction on reduction.

Most proof cases are omitted, the complete proof can be found in the appendix (B.2).

- Case: $S;\ (ti^*;\ l;\ \phi)^? \vdash_i v_1^j\ (t.\mathsf{const}\ c)\ v_2^k;\ (t.\mathsf{const}\ c')\ (\mathsf{set\_local}\ j) : ti^*;\ l;\ \phi$
  $\wedge\ v_1^j\ (t.\mathsf{const}\ c)\ v_2^k;\ (t.\mathsf{const}\ c')\ \mathsf{set\_local}\ j \hookrightarrow v_1^j\ (t.\mathsf{const}\ c')\ v_2^k;\ \epsilon$

  We want to show that

  $$S;\ (ti^*;\ l;\ \phi)^? \vdash_i v_1^j\ (t.\mathsf{const}\ c')\ v_2^k;\ (t.\mathsf{const}\ c')\ (\mathsf{set\_local}\ j) : ti^*;\ l;\ \phi$$

  We know $\vdash\ (t.\mathsf{const}\ c)\ :\ (t\ \ a);\ (\circ, (t\ \ a), (=\ a\ \ (t\ \ c)))$, $S;\ S_{\text{inst}}(i) \vdash (t.\mathsf{const}\ c')\ (\mathsf{set\_local}\ j) : \epsilon : l_1;\ \phi_1^j, (\circ, (t\ a), (=\ a\ (t\ c))), \phi_2^k \to ti^*;\ l;\ \phi$, and $l_1(j) = (t\ a)$, and $C_{\text{local}}(j) = t$ because they are premises of Rule Code that we have assumed to hold.

  By Lemma Inversion on Rule Composition, $S;\ S_{\text{inst}}(i) \vdash (t.\mathsf{const}\ c') : \epsilon;\ l_1;\ \phi_1^j, (\circ, (t\ a), (=\ a\ (t\ c))), \phi_2^k \to ti_3^*;\ l_3;\ \phi_3$, $S;\ S_{\text{inst}}(i) \vdash \mathsf{set\_local}\ j : ti_3^*;\ l_3;\ \phi_3 \to ti^*;\ l;\ \phi$.

  Recall that $t = C_{\text{local}}(j)$, then by Lemma Inversion on Rule Set-Local we have $ti_3^* = ti^*\ (t\ a')$, $l = l_3[j := (t\ a')]$, and $\phi_3 \implies \phi$.

  Then, by Lemma Inversion on Rule Const, $ti^* = \epsilon$, $l_1 = l_3$, and $\phi_1^j, (\circ, (t\ a), (=\ a\ (t\ c))), \phi_2^k, (t\ a'), (=\ a'\ (t\ c')) \implies \phi_3$.

  Now we have all the information we need to derive the same type for $(t.\mathsf{const}\ c')$

  We have $S;\ S_{\text{inst}}(i) \vdash \epsilon : \epsilon;\ l;\ \phi \to \epsilon;\ l;\ \phi$ by Rule Empty.

  Then, $S;\ S_{\text{inst}}(i) \vdash \epsilon\ :\ \epsilon;\ l;\ \phi_1^j, (\circ, (t\ \ a), (=\ a\ \ (t\ \ c))), \phi_2^k, (t\ \ a'), (=\ a'\ (t\ c')) \to \epsilon;\ l;\ \phi$ by Rule Subtyping.

  Since a is fresh, $\phi_1^j, \phi_2^k, (t\ \ a'), (=\ a'\ \ (t\ \ c')) \implies \phi_1^j, (\circ, (t\ \ a), (=\ a\ (t\ c))), \phi_2^k, (t\ a'), (=\ a'\ (t\ c'))$.

  Then, $S;\ S_{\text{inst}}(i) \vdash \epsilon : \epsilon;\ l;\ \phi_1^j, \phi_2^k, (t\ a'), (=\ a'\ (t\ c')) \to \epsilon;\ l;\ \phi$ by Rule Subtyping.

  Further, $\vdash (t.\mathsf{const}\ c') : (t\ a');\ \circ, (t\ a'), (=\ a'\ (t\ c'))$ by Rule Admin-Const.

  Therefore, $S;\ (ti^*;\ l;\ \phi)^? \vdash_i v_1^j\ (t.\mathsf{const}\ c')\ v_2^k;\ \epsilon : ti^*;\ l;\ \phi$ by Rule Code.

- Case: $S;\ (ti^*;\ l;\ \phi)^?\ \vdash_i\ v^*;\ (\mathsf{i32.const}\ k)\ (t.\mathsf{load}\ align\ o) : ti^*;\ l;\ \phi$
  $\wedge\ s;\ (\mathsf{i32.const}\ k)\ (t.\mathsf{load}\ align\ o) \hookrightarrow_i t.\mathsf{const}\ \mathrm{const}_t(b^*),$
  where $s_{\mathrm{mem}}(i, k+o, |t|) = b^*$

  We want to show that $S;\ (ti^*;\ l;\ \phi)^?\ \vdash_i\ v^*;\ (t.\mathsf{const}\ \mathrm{const}_t(b^*)) : ti^*;\ l;\ \phi$

  We know $(\vdash\ v\ :\ ti_v;\ \phi_v)^*$ and $S;\ C\ \vdash\ (\mathsf{i32.const}\ k)\ (t.\mathsf{load}\ align\ o)\ :\ \epsilon;\ ti_v^*;\ \phi_v^* \to ti^*;\ l;\ \phi$ because they are premises of Rule Code which we have assumed to hold.

  Then, by Lemma Inversion on Rule Composition, Rule Const, Rule Mem-Load, we know $ti^* = (t\ a)$, $ti_v^* = l$, and $\phi_v^*, (t\ a)\implies \phi$.

  We have

  $$S;\ S_{\mathrm{inst}}(i) \vdash t.\mathsf{const}\ \mathrm{const}_t(b^*) : \epsilon;\ ti_v^*;\ \phi_v^* \to (t\ a);\ l;\ \phi_v^*, (t\ a), (=\ a\ (t\ c))$$

  by Rule Const.

  Then, $S;\ S_{\mathrm{inst}}(i) \vdash (t.\mathsf{const}\ \mathrm{const}_t(b^*)) : \epsilon;\ ti_v^*;\ \phi_v^* \to (t\ a);\ l;\ \phi$ by Rule Subtyping.

  Recall $(\vdash\ v\ :\ ti_v;\ \phi_v)^*$, then $S;\ (ti^*;\ l;\ \phi)^?\ \vdash_i\ v^*;\ t.\mathsf{const}\ \mathrm{const}_t(b^*)\ :\ ti^*;\ l;\ \phi$ by Rule Code.

- Case: $S;\ (ti^*;\ l;\ \phi)^?\ \vdash_i\ v^*;\ (\mathsf{i32.const}\ k)\ (t.\mathsf{const}\ c)\ (t.\mathsf{store}\ align\ o)\ :\ ti^*;\ l;\ \phi$
  $\wedge\ s;\ (\mathsf{i32.const}\ k)\ (t.\mathsf{const}\ c)\ (t.\mathsf{store}\ align\ o) \hookrightarrow_i\ s';\ \epsilon,$ where $s' = s$ with $\mathrm{mem}(i, k+o, |t|) = \mathrm{bits}_t^{|t|}(c)$

  We know $(\vdash v : ti_v;\ \phi_v)^*$ and

  $$S;\ S_{\mathrm{inst}}(i) \vdash (\mathsf{i32.const}\ k)\ (t.\mathsf{const}\ c)\ (t.\mathsf{store}\ align\ o) : \epsilon;\ ti_v^*;\ \phi_v^* \to ti^*;\ l;\ \phi$$

  because they are premises of Rule Code which we have assumed to hold.

  Then, by Lemma Inversion on Rule Composition, Rule Const,

and Rule MEM-STORE, we have $ti^* = \epsilon$, $ti_v = l$, and $\phi_v^*, (\mathsf{i32}\ a_1), (= a_1\ (\mathsf{i32}\ k)), (t\ a_2), (= a_2\ (t\ c)) \implies \phi$.

Since a_1 and a_2 are fresh, $\phi_v^* \implies \phi$.

We have $S;\ S_{\text{inst}}(i) \vdash \epsilon : \epsilon;\ l;\ \phi_v^* \to \epsilon;\ l;\ \phi_v^*$ by Rule EMPTY.

Then, $S;\ S_{\text{inst}}(i) \vdash \epsilon : \epsilon;\ ti_v;\ \phi_v^* \to \epsilon;\ l;\ \phi$ by Rule SUBTYPING.

Recall that $(\vdash v : ti_v;\ \phi_v)^*$. Therefore, $S;\ (ti^*;\ l;\ \phi)^? \vdash_i v^*;\ \epsilon : ti^*;\ l;\ \phi$ by Rule CODE.

Now we must ensure that the new store s' is well typed: $\vdash s' : S$.

Recall $\vdash s : S$, then $S_{\text{mem}}(i) = n$ and $s_{\text{mem}}(i) = b^*$ where $n \leq |b^*|$ because it's a premise of Rule STORE.

Since $s' = s$ with $\text{mem}(i, k + o, |t|) = \text{bits}_t^{|t|}(c)$, then $|s'_t extmem(i) \models |s_t extmem(i)|$, and therefore $n \leq |s'_t extmem(i)|$, so $s' : S$ by Rule STORE.

- Otherwise: we have $(\vdash (t.\mathsf{const}\ c) : (t\ a);\ (\circ, (t\ a), (= a\ (t\ c))))^*$, and $S, S_{\text{inst}}(i) \vdash e^* : \epsilon;\ (t\ a)^*;\ (\circ, (t\ a), (= a\ (t\ c)))^* \to ti;\ l;\ \phi$

  By Lemma SUBJECT REDUCTION WITHOUT EFFECTS, we have $S, S_{\text{inst}}(i) \vdash e'^* : \epsilon;\ (t\ a)^*;\ (\circ, (t\ a), (= a\ (t\ c)))^* \to ti;\ l;\ \phi$.

  Then, $S;\ (ti^*;\ l;\ \phi)^? \vdash v^*;\ e'^* : ti;\ l;\ \phi$

□

Lemma SUBJECT REDUCTION WITHOUT EFFECTS is used in the subject reduction proof to separate out cases that do not modify state since it simplifies the reasoning. Further, by separating these cases, we can abstract out the common pattern of building back up to Rule PROGRAM from the instruction typing judgment $S;\ C \vdash tfi$. To avoid needing to do mutual inversion, we do not include the local block case here. We do not include any instructions here that modify state, such as $\mathsf{set\_local}$ or $\mathsf{store}_{\mathsf{prechk}}$, meaning that this is non-exhaustive. That is on purpose because we are using this lemma to handle simple cases, and more complex cases are handled separately. We show that if a sequence of instructions e^* reduces to another sequence of

instructions e'^*, and the reduction does not modify the program state (the store s or the locals v^*), then e'^* has the same precondition $ti_1^*;\ l_1;\ \phi_1$ and postcondition $ti_2^*;\ l_2;\ \phi_2$ as e^*.

Lemma 12. Subject Reduction Without Effects

If $S;\ C \vdash e^* : ti_1^*;\ l_1;\ \phi_1 \to ti_2^*;\ l_2;\ \phi_2$,
$\vdash s : S$ (note: we omit this for cases which do not use s),
and $s;\ v^*;\ e^* \hookrightarrow s;\ v^*;\ e'^*$,
then $S;\ C \vdash e'^* : ti_1^*;\ l_1;\ \phi_1 \to ti_2^*;\ l_2;\ \phi_2$

Proof. By case analysis on the reduction rules.

Most proof cases are omitted, the complete proof can be found in the appendix (B.2).

- $S;\ C \vdash L^0[\mathsf{trap}] : ti_1^*;\ l_1;\ \phi_1 \to ti_2^*;\ l_2;\ \phi_2$
  $\wedge\ L^0[\mathsf{trap}] \hookrightarrow \mathsf{trap}$

  This case is trivial since trap accepts any precondition and postcondition. Thus, $S;\ C \vdash \mathsf{trap} : ti_1^*;\ l_1;\ \phi_1 \to ti_2^*;\ l_2;\ \phi_2$ by Rule Trap.

- $S;\ C \vdash (t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ t.binop : ti_1^*;\ l_1;\ \phi_1 \to ti_2^*;\ l_2;\ \phi_2$
  $\wedge\ (t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ t.binop \hookrightarrow t.\mathsf{const}\ c$ where $c = binop(c_1, c_2)$

  We want to show that $S;\ C \vdash t.\mathsf{const}\ c : ti_1^*;\ l_1;\ \phi_1 \to ti_2^*;\ l_2;\ \phi_2$.

  We begin by reasoning about the type of the original instructions $(t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ t.binop$

  By Lemma Inversion on Rule Composition, Rule Const, and Rule Binop, we know that $ti_2^* = ti_1^*(t\ a_3)$, $l_2 = l_1$, and that

  $$\begin{array}{ll} \phi_1, (t\ a_1), (= a_1\ (t\ c_1)), & \implies \phi_2 \\ \quad (t\ a_2), (= a_2\ (t\ c_2)), & \\ \quad (t\ a_3), (= a_3\ (binop\ a_1\ a_2)) & \end{array}$$

  Now we will show that $t.\mathsf{const}\ c$ has the appropriate type.

  By *const*, $S;\ C \vdash t.\mathsf{const}\ c : \epsilon;\ l_1;\ \phi_1 \to (t\ a_3);\ l_1;\ \phi_1, (t\ a_3), (= a_3\ (t\ c))$.

Because $c = binop_t(c_1, c_2)$, then by $\Longrightarrow$,

$$\begin{aligned}\phi_1, (t\ a), (=\ a\ (t\ c)) \implies\ & \phi_1, (t\ a_1), (=\ a_1\ (t\ c_1)),\\ & (t\ a_2), (=\ a_2\ (t\ c_2)),\\ & (t\ a_3), (=\ a_3\ (binop\ a_1 a_2))\end{aligned}$$

Therefore, $S; C \vdash (t.\mathsf{const}\ c) : ti_1^*; l_1; \phi_1 \to ti_1^*\ (t\ a_3); l_1; \phi_2$, by Rule Stack-Poly and Rule Subtyping.

- $C \vdash (t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ t.binop : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$
  $\wedge\ (t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ t.binop \hookrightarrow \mathsf{trap}$

  This case is trivial since trap accepts any precondition and postcondition. Thus, $S; C \vdash \mathsf{trap} : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$ by Rule Trap.

- Case: $S; C \vdash (t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ (\mathsf{i32.const}\ 0)\ \mathsf{select}$
  $: ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$
  $\wedge\ (t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ (\mathsf{i32.const}\ 0)\ \mathsf{select} \hookrightarrow (t.\mathsf{const}\ c_2)$

  We want to show that $S; C \vdash (t.\mathsf{const}\ c_2) : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$.

  First, we reason about what the original type must look like.

  By Lemma Inversion on Rule Composition, Rule Const, and Rule Select, we know that $ti_2^* = ti_1^*\ (a_3\ )$, $l_2 = l_1$, and

  $$\begin{aligned}&\phi_1, (t\ a_1), (=\ a_1\ (t\ c_1)),\\ &\quad (t\ a_2), (=\ a_2\ (t\ c_2)),\\ &\quad (\mathsf{i32}\ a), (=\ a\ (\mathsf{i32}\ 0)),\\ &\quad (t\ a_3), (if\ (=\ a\ (\mathsf{i32}\ 0))\ (=\ a_3\ a_2)\ (=\ a_3\ a_1))\\ &\implies \phi_2\end{aligned}$$

  Now we show that $(t.\mathsf{const}\ c_2)$ has the appropriate type.

  By Rule Const,
  $C \vdash (t.\mathsf{const}\ c_2) : \epsilon; l_1; \phi_1$
  $\to (t\ a_3); l_1; \phi_1, (t\ a_3), (=\ a_3\ (t\ c_2))$

Then, $S; C \vdash (t.\mathsf{const}\ c_2) : ti_1^*; l_1; \phi_1 \to ti_1^*\ (t\ a_3); l_1; \phi_1, (t\ a_3), (= a_3\ (t\ c_2))$ by Rule Stack-Poly.

By $\Longrightarrow$, we have

$$\begin{aligned}\phi_1, (t\ a_3), (= a_3\ (t\ c_2)) \Longrightarrow\ & \phi_1, (t\ a_1), (= a_1\ (t\ c_1)),\\ & (t\ a_2), (= a_2\ (t\ c_2)),\\ & (\mathsf{i32}\ a), (= a\ (\mathsf{i32}\ 0)),\\ & (t\ a_3), (\mathsf{if}\ (= a\ (\mathsf{i32}\ 0))\\ & \qquad (= a_3\ a_2)\\ & \qquad (= a_3\ a_1))\end{aligned}$$

Therefore, $S; C \vdash (t.\mathsf{const}\ c_2) : ti_1^*; l_1; \phi_1 \to ti_2^*\ (t\ a_3); l_1; \phi_2$ by $sub - typing$

- Case: $S; C \vdash (t.\mathsf{const}\ c)^n\ \mathsf{block}\ (ti_3^n; l_3; \phi_3 \to ti_4^m; l_4; \phi_4)\ e^*\ \mathsf{end} : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$
  $\wedge\ (t.\mathsf{const}\ c)^n\ \mathsf{block}\ (ti_3^n; l_3; \phi_3 \to ti_4^m; l_4; \phi_4)\ e^*\ \mathsf{end}$
  $\hookrightarrow \mathsf{label}_m\{\epsilon\}\ (t.\mathsf{const}\ c)^n\ e^*\ \mathsf{end}$

  We want to show that $\mathsf{label}_m\{\epsilon\}\ (t.\mathsf{const}\ c)^n\ e^*\ \mathsf{end} : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$.

  First, we reason about $ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$.

  We know $S; C \vdash \mathsf{block}\ (ti_3^n; l_3; \phi_3 \to ti_4^m; l_4; \phi_4)\ e^*\ \mathsf{end}$
  $: ti_1^*\ (t\ a)^n; l_1; \phi_1, (t\ a)^n, (= a\ (t\ c))^n \to ti_2^*; l_2; \phi_2$
  by Lemma Inversion on Rule Composition and Rule Const.

  By Lemma Inversion on Rule Block, $l_1 = l_3$ and $l_2 = l_4$. We will use l_1, l_2 in place of l_3, l_4, respectively, for the remainder of the case.

  Then, $S; C, \mathrm{label}(t_4^m; l_2; \phi_4) \vdash e^* : (t\ a)^n; l_1; \phi_3 \to ti_4^m; l_2; \phi_4$ because it is a premise of Rule Block which we have already assumed to hold.

  Also, $(t\ a)^n = ti_3^n$, $ti_2^* = ti_1^*\ ti_4^m$, $\phi_1, (t\ a)^n, (= a\ (t\ c))^n \Longrightarrow \phi_3$, and $\phi_4 \Longrightarrow \phi_2$ by Lemma Inversion on Rule Block.

  Now we have all the information we need to show that
  $\mathsf{label}_m\{\epsilon\}\ (t.\mathsf{const}\ c)^n\ e^*\ \mathsf{end} : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$.

Remember that Rule Label uses the types of both the body $(t.\mathsf{const}\ c)^n\ e^*$ and the stored instructions ϵ.

First, we show the type of the body.

We have

$$S;\ C, \mathrm{label}(t_4^m;\ l_2;\ \phi_4) \vdash (t.\mathsf{const}\ c)^n : \epsilon;\ l_1;\ \phi_1 \\ \to (t\ a)^n;\ l_1;\ \phi_1, (t\ a)^n, (=\ a\ (t\ c))^n$$

by Rule Const.

Then, since $\phi_1, (t\ a)^n, (=\ a\ (t\ c))^n \implies \phi_3$, we have

$$S;\ C, \mathrm{label}(t_3^n;\ l_1;\ \phi_3) \vdash (t.\mathsf{const}\ c)^n : \epsilon;\ l_1;\ \phi_1 \to (t\ a)^n;\ l_1;\ \phi_3$$

by Rule Subtyping.

Recall we have $S;\ C, \mathrm{label}(t_4^m;\ l_2;\ \phi_4) \vdash e^* : (t\ a)^n;\ l_1;\ \phi_3 \to ti_4^m;\ l_2;\ \phi_4$.

Then $S;\ C, \mathrm{label}(t_4^m;\ l_2;\ \phi_4) \vdash (t.\mathsf{const}\ c)^n\ e^* : \epsilon;\ l_1;\ \phi_1 \to ti_4^m;\ l_2;\ \phi_4$ by Rule Composition.

We have the type we want from the body. Now we get the type we want of the stored instructions. We already have the postcondition we want, $t_4^m;\ l_2;\ \phi_4$, in the label stack, so we want the stored instruction to just pass the information through. Since the stored instructions is ϵ, this is simple to show: we have $S;\ C \vdash \epsilon : ti_2^m;\ l_2;\ \phi_4 \to ti_2^m;\ l_2;\ \phi_4$ by Rule Empty and Rule Stack-Poly.

Therefore, $C \vdash \mathsf{label}_m\{\epsilon\}\ (t.\mathsf{const}\ c)^n\ e^*\ \mathsf{end} : \epsilon;\ l_1;\ \phi_1 \to ti_2^m;\ l_2;\ \phi_4$ by *label*.

Finally, since $\phi_4 \implies \phi_2$, $S;\ C \vdash \mathsf{label}_m\{\epsilon\}\ (t.\mathsf{const}\ c)^n\ e^*\ \mathsf{end} : ti_1^*;\ l_1;\ \phi_1 \to ti_1^*\ ti_4^m;\ l_2;\ \phi_2$ by Rule Stack-Poly and Rule Subtyping.

- Case: $S;\ C \vdash \mathsf{label}_n\{e^*\}\ L^j[(t.\mathsf{const}\ c)^n\ (\mathsf{br}\ j)]\ \mathsf{end} : ti_1^*;\ l_1;\ \phi_1 \to ti_2^*;\ l_2;\ \phi_2$
  $\wedge\ \mathsf{label}_n\{e^*\}\ L^j[(t.\mathsf{const}\ c)^n\ (\mathsf{br}\ j)] \hookrightarrow (t.\mathsf{const}\ c)^n\ e^*$

  We want to show that $v^n\ e^* : ti_1^*;\ l_1;\ \phi_1 \to ti_2^*;\ l_2;\ \phi_2$.

Intuitively, this proof works because the premise of Rule Br assumes that $C_{\text{label}}(i)$ is the precondition ($ti_1^n;\ l_3;\ \phi_5$, as we will soon see) of the stored instructions e^* in the $i+1$th label, and the postcondition of the label block is immediately reachable from the postcondition of e^*. Meanwhile, that assumptions is ensured by Rule Label, which ensures that e^* has the same precondition as the $i+1$th branch postcondition on the label stack and the same postcondition as the label block instruction.

By Lemma Inversion on Rule Label, $ti_2^* = ti_1^*\ ti_4^*$ for some ti_4^*.

Also, $S;\ C, \text{label}(ti_1^n;\ l_3;\ \phi_5)^j \vdash (t.\text{const}\ c)^n\ (\text{br}\ j) : \epsilon;\ l_3;\ \phi_3 \to ti_\emptyset^*;\ l_\emptyset;\ \phi_\emptyset$ for some l_3 and ϕ_3, where $\phi_5 = \phi_3, (t\ a)^n, (=\ a\ (t\ c))^n$, by Lemma Inversion on Rule Label and Rule Br.

Then, $S;\ C, \text{label}(ti_1^n;\ l_3;\ \phi_5)^j \vdash (\text{br}\ j) : ti_1^n;\ l_3;\ \phi_5 \to ti_\emptyset^*;\ l_\emptyset;\ \phi_\emptyset$, by Lemma Inversion on Rule Composition and Rule Const.

Then, $S;\ C, \text{label}(ti_1^n;\ l_3;\ \phi_5)^j \vdash (t.\text{const}\ c)^n : \epsilon;\ l_3;\ \phi_3 \to ti_1^n;\ l_3;\ \phi_5$ since it is a premise of *composition* which we have assumed to hold.

Further, $S;\ C \vdash e^* : ti_1^n;\ l_3;\ \phi_5 \to ti_2^*;\ l_2;\ \phi_4$ since it is a premise of Rule Label which we have assumed to hold, and $\phi_4 \implies \phi_2$ by Lemma Inversion on Rule Label.

Then, $S;\ C \vdash (t.\text{const}\ c)^n\ e^* : \epsilon;\ l_1;\ \phi_1 \to ti_2^*;\ l_2;\ \phi_4$ by Lemma Lift-Consts and Rule Composition.

Finally, $C \vdash (t.\text{const}\ c)^n\ e^* : ti_1^*;\ l_1;\ \phi_1 \to ti_1^*\ ti_4^*;\ l_2;\ \phi_2$ by Rule Stack-Poly and Rule Subtyping.

- Case: $S;\ C \vdash (\text{i32.const}\ j)\ \text{call\_indirect}\ ti_3^*;\ l_3;\ \phi_3 \to ti_4^*;\ l_4;\ \phi_4$
  $: ti_1^*;\ l_1;\ \phi_1 \to ti_2^*;\ l_2;\ \phi_2$
  $\wedge\ s;\ (\text{i32.const}\ j)\ \text{call\_indirect}\ ti_3^*;\ l_3;\ \phi_3 \to ti_4^*;\ l_4;\ \phi_4 \hookrightarrow_i \text{call}\ s_{\text{tab}}(i,j)$
  where $s_{\text{tab}}(i,j)_{\text{code}} = (\text{func}\ tfi_0\ \text{local}\ t^*\ e^*)$ and $tfi_0 <: ti_3^*;\ l_3;\ \phi_3 \to ti_4^*;\ l_4;\ \phi_4$

  We want to show that $\text{call}\ s_{\text{tab}}(i,j) : ti_1^*;\ l_1;\ \phi_1 \to ti_2^*;\ l_2;\ \phi_2$.

  By Lemma Inversion on Rule Composition, Rule Const, and Rule Call-Indirect, we know that $ti_1^* = ti_0^*\ ti_3^*$ and $ti_2^* = ti_0^*\ ti_4^*$ for some

ti_0^*, $l_1 = l$)2, $\phi_1 \implies \phi_3$, and $\phi_4 \implies \phi_2$.

We know $S \vdash s_{\text{tab}}(i,j) : tfi_0$ since it is a premise of $\vdash s : S$ which we have assumed to hold.

Then, $S; C \vdash \mathsf{call}\ s_{\text{tab}}(i,j) : tfi_0$ by Rule CALL-CL.

$S; C \vdash \mathsf{call}\ s_{\text{tab}}(i,j) : ti_3^*; l_1; \phi_3 \to ti_4^*; l_2; \phi_4$ by Rule SUBTYPING.

Therefore, $S; C \vdash \mathsf{call}\ s_{\text{tab}}(i,j) : ti_0^*\ ti_1^*; l_1; \phi_1 \to ti_0^*\ ti_1^*; l_2; \phi_2$ by Rule STACK-POLY.

□

4.3.2 Progress

Progress ensures that if a program is well typed then it either: entirely consists of values, traps, or is reducible (*i.e.,* there exists another program that it reduces to). Proving progress for Wasm-prechk is the key metatheoretic property that ensures that our claim that Wasm-prechk is as safe as Wasm is valid. This is because it connects the static guarantees of the type system to the dynamic assumptions of *prechk*-tagged instructions. By proving that well-typed *prechk*-tagged instructions will always be reducible, we prove that the static guarantees are sufficient to ensure that they will not trap and therefore the dynamic checks are unnecessary.

Since most Wasm-prechk instructions have the same semantics as in Wasm, and every Wasm-prechk type includes all the information of a Wasm type, we can reuse the Wasm proof for those instructions by using the erasure function from Section 4.2.2. The intuition for this is that the Wasm-prechk indexed type system provides strictly more information than the Wasm type system. However, for Wasm-prechk instructions that do not have the same semantics as in Wasm, specifically *prechk*-tagged instructions, we still must prove those cases.

Theorem 4. Progress If $\vdash_i s; v^*; e^* : ti^*; l; \phi$ then either $e^* = v'^*$, $e^* = \mathsf{trap}$, or $s; v^*; e^* \hookrightarrow_i s'; v'^*; e'^*$.

Proof. We proceed by induction on $\vdash_i s; v^*; e^* : ti^*; l; \phi$.

Because $\vdash_i s; v^*; e^* : ti^*; l; \phi$, we know that $\vdash s : S$ for some S, and that $S; \epsilon \vdash_i v^*; e^* : ti^*; l; \phi$ because they are premises of Rule PROGRAM which we have assumed to hold.

Then we know that $(\vdash v : (t_v\ a_v); \phi_v)^*$ and $S; S_{\text{inst}}(i), \text{local}\ t_v^* \vdash e^* : \epsilon; (t_v\ a_v)^*; \phi_v^* \to ti^*; l; \phi$ because they are premises of Rule CODE which we have assumed to hold.

- Case: $\vdash_i s; v^*; (t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ t.\mathsf{div}_{\mathsf{prechk}}$

  We must show that $(t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ t.\mathsf{div}_{\mathsf{prechk}} \hookrightarrow e'^*$ for some e'^*.

  We have $S; ()\ \vdash_i\ v^*; (t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ t.\mathsf{div}_{\mathsf{prechk}}\ :\ ti^*; l; \phi$ for some ti^*, l, and ϕ because it is a premise of Rule PROGRAM which we have assumed to hold.

  Then, $(\vdash v : (t_v\ a_v); \phi_v)^*$ for some $(t_v\ a_v)^*$ and ϕ_v^*, since it is a premise of Rule CODE which we have assumed to hold.

  It is important to note that ϕ_v^* cannot contain a contradiction because it contains a single equality constraint per fresh index variable (see Rule ADMIN-CONST).

  Further,

$$S; S_{\text{inst}}(i), \text{local}\ t_v^* \vdash (t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ t.\mathsf{div}_{\mathsf{prechk}} : \epsilon; (t_v\ a_v)^*); \phi_v^* \to ti^*; l; \phi$$

  because it too is a premise of Rule CODE.

  Then,

$$\begin{aligned} S_{\text{inst}}(i) \vdash\ & (t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2) \\ & : \epsilon; (t_v\ a_v)^*); \phi_v^* \\ & \to (t\ a_1)\ (t\ a_2); (t_v\ a_v)^*); \phi_v^*, (t\ a_1), (= a_1\ (t\ c_1)), \\ & \qquad\qquad (t\ a_2), (= a_2\ (t\ c_2)) \end{aligned}$$

  where $\phi_v^*, (t\ a_1), (= a_1\ (t\ c_1)), (t\ a_2), (= a_2\ (t\ c_2)) \implies \neg(= a_2\ (t\ 0))$ by Lemma INVERSION on Rule COMPOSITION and Rule DIV-PRECHK.

Therefore, it must be the case that $c_2 \neq 0$, and therefore there must exist some c_3 such that $c_3 = div(c_1, c_2)$ since $div(c_1, c_2)$ is well-defined when c_2 is non-zero. Then, $s;\ (t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ t.\mathsf{div}_{\mathsf{prechk}} \hookrightarrow_i (t.\mathsf{const}\ c_3)$.

- Case: $\vdash_i s;\ v^*;\ (\mathsf{i32.const}\ k)\ (t.\mathsf{load}_{\mathsf{prechk}}\ (tp\_sx)\ align\ o)$

  We must show that $s;\ (\mathsf{i32.const}\ k)\ (t.\mathsf{load}_{\mathsf{prechk}}\ (tp\_sx)\ align\ o) \hookrightarrow e'^*$ for some e'^*.

  We have $S;\ \epsilon \vdash_i v^*;\ (\mathsf{i32.const}\ k)\ (t.\mathsf{load}_{\mathsf{prechk}}\ (tp\_sx)\ align\ o) : ti^*;\ l;\ \phi$ for some ti^*, l, and ϕ because it is a premise of Rule Program which we have assumed to hold.

  We also have that $\vdash s : S$, and therefore $(n \leq |b^*|)^*$ where $S_{\text{tab}} = n^*$ and $s_{\text{mem}} = (b^*)^*$.

  Then, $(\vdash v : (t_v\ a_v);\ \phi_v)^*$ for some $(t_v\ a_v)^*$ and ϕ_v^*, since it is a premise of Rule Code which we have assumed to hold.

  It is important to note that ϕ_v^* cannot contain a contradiction because it contains a single equality constraint per fresh index variable (see Rule Admin-Const).

  Further, we have that

  $$S;\ S_{\text{inst}}(i), \text{local}\ t_v^* \vdash (\mathsf{i32.const}\ k)\ (t.\mathsf{load}_{\mathsf{prechk}}\ (tp\_sx)\ align\ o) : \epsilon;\ (t_v\ a_v)^*;\ \phi_v^* \to ti^*;\ l;\ \phi$$

  because it too is a premise of Rule Code.

  Then,

  $$S_{\text{inst}}(i) \vdash (\mathsf{i32.const}\ k) : \epsilon;\ (t_v\ a_v)^*;\ \phi_v^* \to (\mathsf{i32}\ a);\ (t_v\ a_v)^*;\ \phi_v^*, (\mathsf{i32}\ a), (=\ a\ (\mathsf{i32}\ k))$$

where

$$\phi_v^*, (\mathsf{i32}\ a), (=\ a\ (\mathsf{i32}\ k)) \implies (\mathsf{ge}\ (\mathsf{add}\ a\ (\mathsf{i32}\ o))(\mathsf{i32}\ 0)), \\ (\mathsf{le}\ (\mathsf{add}\ a\ (\mathsf{add}\ (\mathsf{i32}\ o + width))) \\ (\mathsf{i32}\ n_2 * 64\mathrm{Ki}))$$

and $n_2 * 64\mathrm{Ki} = S_{\mathrm{mem}}(i, j)$ by Lemma Inversion on Rule Composition and Rule Store-Prechk.

Because we have

$$\phi_v^*, (\mathsf{i32}\ a), (=\ a\ (\mathsf{i32}\ k)) \implies (\mathsf{ge}\ (\mathsf{add}\ a\ (\mathsf{i32}\ o))(\mathsf{i32}\ 0)), \\ (\mathsf{le}\ (\mathsf{add}\ a\ (\mathsf{add}\ (\mathsf{i32}\ o + width))) \\ (\mathsf{i32}\ n_2 * 64\mathrm{Ki}))$$

, then we must have $k + o \geq 0$ and $k + o + |tp| \leq n_2 * 64\mathrm{Ki}$.

Recall $\vdash s : S$. Then, since $n_2 * 64\mathrm{Ki} = S_{\mathrm{mem}}(i, j)$, we have $s_{\mathrm{mem}}(i, j) = b_2^*$ where $n_2 * 64\mathrm{Ki} \leq |b_2^*|$.

Therefore, it must be the case that $k + o \geq 0$ and $k + o + |tp| < |b_2^*|$, and therefore $s_{\mathrm{mem}}(i, k + o, |tp|) = b_3^*$ for some b_3^* that is a subsequence of b_2^*. Then, $s;\ (\mathsf{i32.const}\ k)\ (t.\mathsf{load}_{\mathsf{prechk}}\ (tp\_sx)\ align\ o) \hookrightarrow_i t.\mathsf{const}\ \mathrm{const}_t^{sx}(b_3^*)$.

- Case: $\vdash_i s;\ v^*;\ (\mathsf{i32.const}\ k)\ t.\mathsf{load}_{\mathsf{prechk}}\ align\ o$

  Same as above, except with $|t|$ replacing $|tp|$ and $\mathrm{const}_t(b_3^*)$ instead of $\mathrm{const}_t^{sx}(b_3^*)$.

- Case: $\vdash_i s;\ v^*;\ (\mathsf{i32.const}\ k)\ (t.\mathsf{const}\ c)\ (t.\mathsf{store}_{\mathsf{prechk}}\ tp\ align\ o)$

  We must show that $s;\ (\mathsf{i32.const}\ k)\ (t.\mathsf{store}_{\mathsf{prechk}}\ tp\ align\ o) \hookrightarrow e'^*$ for some e'^*.

  We have

$$S;\ ()\vdash_i v^*;\ (\mathsf{i32.const}\ k)\ (t.\mathsf{store}_{\mathsf{prechk}}\ (tp\_sx)\ align\ o) : ti^*;\ l;\ \phi$$

for some ti^*, l, and ϕ because it is a premise of Rule PROGRAM which we have assumed to hold.

We also have that $\vdash s : S$, and therefore $(n \leq |b^*|)^*$ where $S_{\text{tab}} = n^*$ and $s_{\text{mem}} = (b^*)^*$.

Then, $(\vdash v : (t_v\ a_v);\ \phi_v)^*$ for some $(t_v\ a_v)^*$ and ϕ_v^*, since it is a premise of Rule CODE which we have assumed to hold.

It is important to note that ϕ_v^* cannot contain a contradiction because it contains a single equality constraint per fresh index variable (see Rule ADMIN-CONST).

Further, we have that

$$S;\ S_{\text{inst}}(i), \text{local}\ t_v^* \vdash (\text{i32.const}\ k)\ (t.\text{store}_{\text{prechk}}\ tp\ align\ o) : \epsilon;\ (t_v\ a_v)^*;\ \phi_v^* \to ti^*;\ l;\ \phi$$

because it too is a premise of Rule CODE.

Then,

$$\begin{aligned} &S_{\text{inst}}(i) \vdash (\text{i32.const}\ k)\ (t.\text{const}\ c) : \\ &\quad \epsilon;\ (t_v\ a_v)^*;\ \phi_v^* \\ &\quad \to (\text{i32}\ a)\ (t\ a_2);\ (t_v\ a_v)^*;\ \phi_v^*, (\text{i32}\ a), (= a\ (\text{i32}\ k)), \\ &\qquad\qquad\qquad\qquad\qquad\qquad (t\ a_2), (= a_2\ (t\ c)) \end{aligned}$$

where

$$\begin{aligned} &\phi_v^*, (\text{i32}\ a), (= a\ (\text{i32}\ k)), \\ &\quad (t\ a_2), (= a_2\ (t\ c)) \end{aligned} \implies \begin{aligned} &(\text{ge}\ (\text{add}\ a\ (\text{i32}\ o))(\text{i32}\ 0)), \\ &(\text{le}\ (\text{add}\ a\ (\text{add}\ (\text{i32}\ o + width))) \\ &\quad (\text{i32}\ n_2 * 64\text{Ki})) \end{aligned}$$

and $n_2 * 64\text{Ki} = S_{\text{mem}}(i, j)$ by Lemma INVERSION on Rule COMPOSITION and Rule LOAD-PRECHK.

Because we have

$$\begin{array}{ll} (\mathsf{i32}\ a), (=\ a\ (\mathsf{i32}\ k)), & \implies (\mathsf{ge}\ (\mathsf{add}\ a\ (\mathsf{i32}\ o))(\mathsf{i32}\ 0)), \\ (t\ a_2), (=\ a_2\ (t\ c)) & \quad (\mathsf{le}\ (\mathsf{add}\ a\ (\mathsf{add}\ (\mathsf{i32}\ o + width))) \\ & \qquad (\mathsf{i32}\ n_2 * 64\mathrm{Ki})) \end{array}$$

, then we must have $k + o \geq 0$ and $k + o + |tp| \leq n_2 * 64\mathrm{Ki}$.

Recall $\vdash s : S$. Then, since $n_2 * 64\mathrm{Ki} = S_{\mathrm{mem}}(i, j)$, we have $s_{\mathrm{mem}}(i, j) = b_2^*$ where $n_2 * 64\mathrm{Ki} \leq |b_2^*|$.

It must be the case that $k + o \geq 0$ and $k + o + |tp| < |b_2^*|$, and therefore $s_{\mathrm{mem}}(i, k + 0, |tp|) = b_3^*$ for some b_3^* that is a subsequence of b_2^* Then, we can construct $s' = s$ with $s'_{\mathrm{mem}}(i, k + o, |tp|) = bits_t^{|tp|}(c)$ because $|bits_t^{|tp|}(c)| \models |b_3^*|$. Then,

$$s;\ (\mathsf{i32.const}\ k)\ (\mathsf{i32.const}\ c)\ (t.\mathsf{store}_{\mathsf{prechk}}\ tp\ align\ o) \hookrightarrow_i s';\ \epsilon$$

- Case: $\vdash_i s;\ v^*;\ (\mathsf{i32.const}\ c)\ (t.\mathsf{store}_{\mathsf{prechk}}\ align\ o)$

  Same as above, except with $|t|$ replacing $|tp|$.

- Case: $\vdash_i (\mathsf{i32.const}\ c)\ \mathsf{call\_indirect}\ ti_1^*;\ l_1;\ \phi_1 \to ti_2^*;\ l_2;\ \phi_2$

  We must show that $(\mathsf{i32.const}\ c)\ \mathsf{call\_indirect}\ ti_1^*;\ l_1;\ \phi_1 \to ti_2^*;\ l_2;\ \phi_2 \hookrightarrow e'^*$ for some e'^*.

  We have $S;\ () \vdash_i v^*;\ (\mathsf{i32.const}\ c)\ \mathsf{call\_indirect}\ ti_1^*;\ l_1;\ \phi_1 \to ti_2^*;\ l_2;\ \phi_2 : ti^*;\ l;\ \phi$ for some ti^*, l, and ϕ because it is a premise of Rule Program which we have assumed to hold.

  We also have that $\vdash s : S$, and therefore $S_{\mathrm{tab}}(i) = (n, tfi^n)$ and $(S \vdash cl : tfi)^*$ where $s_{\mathrm{tab}}(i) = cl^*$ and $n \leq |cl^*|$.

  Then, $(\vdash v : (t_v\ a_v);\ \phi_v)^*$ for some $(t_v\ a_v)^*$ and ϕ_v^*, since it is a premise of Rule Code which we have assumed to hold.

  It is important to note that ϕ_v^* cannot contain a contradiction because it contains a single equality constraint per fresh index variable (see Rule Admin-Const).

Then,

$$
\begin{aligned}
&S_{\text{inst}}(i) \vdash (\mathsf{i32.const}\ c) : \\
&\qquad \epsilon;\ (t_v\ a_v)^*;\ \phi_v^* \\
&\qquad \to (\mathsf{i32}\ a);\ (t_v\ a_v)^*;\ \phi_v^*, (\mathsf{i32}\ a), (=\ a\ (\mathsf{i32}\ c))
\end{aligned}
$$

where $\phi_v^*, (\mathsf{i32}\ a), (=\ a\ (\mathsf{i32}\ c)) \implies (\mathsf{gt}\ n\ a) \wedge (\mathsf{le}\ (\mathsf{i32}\ 0)\ a)$ by Lemma Inversion on Rule Composition and Rule Call-Indirect-Prechk.

We have

$$\forall i.\ (\phi \implies \neg(=\ (\mathsf{i32}\ i)\ a)) \vee\ tfis(i) <: ti_1^*;\ l_1;\ \phi_1 \to ti_2^*;\ l_2;\ \phi_2$$

where $tfis = tfi^n$, because it is a premise of Rule Call-Indirect-Prechk which we have assumed to hold by Lemma Inversion. Since $(\mathsf{i32}\ a);\ (t_v\ a_v)^*;\ \phi_v^*, (\mathsf{i32}\ a), (=\ a\ (\mathsf{i32}\ c)) \implies (=\ (\mathsf{i32}\ c)\ a)$, then it has to be the case that $tfis(c) <: ti_1^*;\ l_1;\ \phi_1 \to ti_2^*;\ l_2;\ \phi_2$.

Let, $\{\text{inst}\ j,\ \text{func}\ f\} = s_{\text{tab}}(i, c)$. Recall from before that $(S \vdash cl : tfi)^*$. Then, $S \vdash \{\text{inst}\ j,\ \text{func}\ f\} : tfi_2$ for some tfi_2.

$S_{\text{inst}}(j) \vdash f : tfi_2$, as it is a premise of $S \vdash \{\text{inst}\ j,\ \text{func}\ f\} : tfi_2$.

Then, we know that $f = \mathsf{func}\ tfi_2\ \mathsf{local} \ldots$ because it is a premise of $S_{\text{inst}}(j) \vdash f : tfi_2$, and we know that $tfi_2 <: ti_1^*;\ l_1;\ \phi_1 \to ti_2^*;\ l_2;\ \phi_2$.

Thus, $s;\ (\mathsf{i32.const}\ c)\ \mathsf{call\_indirect}\ ti_1^*;\ l_1;\ \phi_1 \to ti_2^*;\ l_2;\ \phi_2$.
$\hookrightarrow \mathsf{call}\ \{\text{inst}\ j,\ \text{func}\ f\}$

- Otherwise, we reuse the Wasm proof, which we can do thanks to Theorem 2.

□

Chapter 5

Implementation

To ensure the feasibility of implementing the Wasm-prechk language we implemented a reference implementation [1] in Redex [2]. We were able to handle the entirety of the type system and use the reference implementation to typecheck several test programs, including test programs for each *prechk*-tagged instruction, and negative results that ensure illegal *prechk*-tagged instructions are not well-typed. Further, our reference implementation is able to implement constraint solving and implication.

5.1 Reference Implementation

We developed the reference implementation of Wasm-prechk in Redex by creating a reference implementation of Wasm and extending it with the Wasm-prechk syntax and type system. This ensured that our model works on silicon and not just in set theory, and made it possible to quickly test out various approaches. The syntactic representation we used in the Redex model differs a bit from the syntactic representation provided here, as it is s-expression based.

However, we have only implemented a typechecker in the Redex model. Thus, we must manually construct derivations and ask if they are valid. This may be solvable using a bi-directional type system approach, or by otherwise

[1] `https://zenodo.org/record/3995114`

coming up with a type inference algorithm.

We implemented the static typing judgment `(⊢ C (e ...) tfi)`, where `e ...` is Redex for e^*. Here is Rule Const in Redex. Note that it is essentially the same, minus a few syntactic differences.

```
[(⊢ C ((t const c))
      ((() l ϕ)
       -> (((t a)) l ((ϕ (t a)) (= a (t c))))))]
```

There is an extra set of parentheses around `(t const c)` because the judgment works over sequences of instructions `(e ...)`, where `e ... = (t const c)` here. Also, there's no dot between the type t and the constant instruction keyword *const* because that is not necessary in Redex's s-expressions.

In the above example, there is no requirement about the freshness of a. Redex does not support freshness requirements in judgments. However, because we use the typing judgment to check manually constructed derivations, we are simply careful in writing the derivations to ensure that this property is true.

5.2 Constraint Solving in Practice

In our implementation, we reason about constraints using the Z3 theorem prover [1].

5.2.1 Translation of Constraints to Z3

Constraint solving and implication involves translating various parts of the Wasm-prechk index language into Z3 constraints. We use Z3 bitvectors to represent index variables. Z3 bitvectors are integers with a fixed-width bit string representation, like Wasm-prechk's i32 and i64, that Z3 can reason about using standard operations (addition, multiplication, shift left, etc...), and handle overflow (which can occur in Wasm). Further, fixed-width bitvectors are a finite domain that has more efficient reasoning and decidability compared to the natural numbers. Since bitvectors have similar operations to the Wasm-

$\boxed{compile_{z3}(x) \doteq SMT}$

$$
\begin{array}{rcl}
a & \doteq & a \\
(\text{i32 } c) & \doteq & (\_\ \texttt{bv}c\ 32) \\
(\text{i64 } c) & \doteq & (\_\ \texttt{bv}c\ 64) \\
(\text{binop } x\ y) & \doteq & (\texttt{bv}binop\ compile_{z3}(x)\ compile_{z3}(y)) \\
(\text{relop } x\ y) & \doteq & (\texttt{bv}relop\ compile_{z3}(x)\ compile_{z3}(y))
\end{array}
$$

$\boxed{compile_{z3}(P) \doteq SMT}$

$$
\begin{array}{rcl}
(=\ x\ y) & \doteq & (=\ compile_{z3}(x)\ compile_{z3}(y)) \\
(\text{if} P_1\ P_2\ P_3) & \doteq & (\texttt{ite}\ \ compile_{z3}(P_1)\ compile_{z3}(P_2)\ compile_{z3}(P_3)) \\
\neg P & \doteq & (\texttt{not}\ \ compile_{z3}(P)) \\
P_1 \wedge P_2 & \doteq & (\texttt{and}\ \ compile_{z3}(P_1)\ compile_{z3}(P_2)) \\
P_1 \vee P_2 & \doteq & (\texttt{or}\ \ compile_{z3}(P_1)\ compile_{z3}(P_2))
\end{array}
$$

$\boxed{compile_{z3}((t\ a)) \doteq SMT}$

$$
\begin{array}{rcl}
(\text{i32 } a) & \doteq & (\texttt{declare-const}\ a\ (\texttt{\_BitVector32})) \\
(\text{i64 } a) & \doteq & (\texttt{declare-const}\ a\ (\texttt{\_BitVector64}))
\end{array}
$$

Figure 5.1: Translation of Wasm-prechk Index Language to Z3

prechk index language, translating an index x to Z3 only requires changing the name of operations (for example, *add* becomes `bv_add`). Translating propositions to Z3 is also straightforward because Z3 has support for all of the first-order logic constructs we use to build and combine propositions.

To test whether the satisfiability of one index type context ϕ_1 implies that some other index type context ϕ_2 is satisfiable, we first generate bitvectors to represent all of the index variables. Z3 constraints are generated based on the propositions in both contexts. We assert that the constraints generated for the first context must hold. Finally, we ask Z3 to find an assignment to the variables declared in the type index contexts where the constraints from the second context do not hold (a counterexample). If a counterexample cannot be found, then the implication must hold, otherwise, it does not hold.

```
(declare-const a (_ BitVec 32))
(declare-const b (_ BitVec 32))
(declare-const c (_ BitVec 32))
(declare-const d (_ BitVec 32))
(define-fun satisfies () Bool
    (=> (and (= c (_ bv1 32))
             (ite (= b (_ bv0 32))
                  (= d c)
                  (= d b))
        (and (not (= d (_ bv0 32))))))
(assert (not satisfies))
(check-sat)
(get-model)
```

Figure 5.2: Example of a Z3 query for constraint satisfaction

We define the translations for certain parts of the index language to SMT (the Z3 constraint language) used in the reference interpreter in Figure 5.1. One key thing to note in the translation is that we use a slightly different representation of *relop* and *testop* within the reference implementation. With *relop*, the Z3 versions of the Wasm-prechk index language relation operators returns booleans, not bitvectors, so we convert every use of *relop* from $(= a\ (relop\ x\ y))$ to

$$(\text{if}\ (= (t\ 1)\ (relop\ x\ y))\ (= a\ (t\ 1))\ (= a\ (t\ 0)))$$

, where t is the primitive Wasm type of a from the type declaration of a in ϕ. We do the same for *testop*, except we first convert $(\mathsf{eqz}\ \ x)$ (the only test operator) to $(= (t\ 0)\ x)$, so then $(= a\ (\mathsf{eqz}\ \ x))$ becomes

$$(\text{if}\ (= (t\ 0)\ x)\ (= a\ (t\ 1))\ (= a\ (t\ 0)))$$

We also include an example of a complete Z3 query in Figure 5.2. It is the Z3 query for testing the implication in the $\mathsf{div}_{\mathsf{prechk}}$ rule in the example in section 3.3.2.

How we implement the table in Z3 Recall from subsection 3.3.2 that for $\mathsf{call\_indirect}_{\mathsf{prechk}}\ tfi$, we have to reason about which functions in a table can be called. In practice, we construct a Z3 array (intuitively similar to a normal array) that is the same size as the table (we chose Z3 arrays because they have a similar abstraction to tables). We fill the array with boolean values, which are *true* if the function at the table index is a suitable function type (a subtype of the expected type tfi), and $false$ otherwise. Finally, we assert all of the translated constraints from the index type context about the table index, and then make sure that reading from the array using the table index returns *true*. We do not show here how we generate Z3 constraints for tables, but it can be found in the Redex model.

5.2.2 Impact of Using Z3

Our choice of using Z3 has impacted Wasm-prechk in several ways.

Floating Point Values and Integer Bit Operations We currently do not support floating point values because Z3 is unable to reason about them. In addition, the Wasm unary operators (*ctz*, *clz*, and *popcnt*, which provide bit-level information) and certain binary operators (*rotr* and *rotl*) would be difficult and inefficient to reason about using Z3 due to their non-linearity, so we do not currently support them in Wasm-prechk. However, we treat these operators (and floating point values), like memory operations and simply assume nothing about them.

Type Annotations in Constraints The requirement of adding explicit type annotations for index variables and constants that appear in type indices comes from needing to know what width the variable will be when we convert it to a Z3 bitvector. Without these type annotations, we would not know whether a constant c should be represented using a 32-bit or 64-bit vector (depending on if it is an $\mathsf{i32}$ or $\mathsf{i64}$).

Performance Concerns There are performance concerns when using an SMT solver. Our small examples programs have been near instantaneous to

typecheck (usually involving only one Z3 invocation with a few constraints). For example, the query in Figure 5.2, is very small and takes milliseconds to run. However, it is likely that we will see a slowdown due to significantly larger constraint sets when typechecking large Wasm programs. This would require a clever approach to reducing the size of the constraint set (perhaps using a form of type-level garbage collection on constraints on index variables that cannot be referred to). Also, we would want to make sure when writing a typechecker/synthesizer that we invoke Z3 as little as possible, by minimizing the amount of reasoning we do about the constraints.

Chapter 6

Discussion

By creating Wasm-prechk, we have taken the first step towards creating a practical system in which an expressive type system is used with an existing and widely low-level language for safety and performance. Wasm-prechk provides concrete ways to use type information for compiler optimization at the assembly language level. This is a first step in the sense that it provides the scaffolding to build such a system that could be part of the infrastructure of the internet: unlike prior work, it is backwards compatible with a commonly used language, Wasm, supported by many major browsers. However, there are still a number of unanswered questions. We have a number of future ideas for this work some based on what we think is necessary to realize our eventual goal of making Wasm-prechk practical in the real-world, and others based on problems identified during the course of the project so far.

Support for Streaming Compilation The format of Wasm code allows compilation and execution to begin with only part of the program downloaded. Similar streaming compilation is theoretically possible with Wasm-prechk, but there are unanswered questions about how to perform typechecking in such a compilation pipeline. Here are two examples of problems that we expect to face implementing such a system. First of all, we must make sure such a system is safe, which is complicated by the fact that we may begin executing code before we have finished typechecking. This should not be

too much of an issue as long as we ensure code is typechecked before we can execute it, so we only execute well-typed code and if we come across code that is not well typed we halt execution and throw a type error. Second of all, this will require highly efficient typechecking, preferably performed in parallel to typecheck many functions at one. We could also try to be clever and prioritize typechecking on functions that we expect to be executed sooner.

More Optimizations There is the potential to find other optimizations we can perform with the additional type information. For example, remember from section 3.3 that an if may have a contradiction in the index type context in one of its branches. In this case, that branch will never be executed, and therefore the other branch must always be taken, so we can safely replace the if instruction with the other branch. We can do similar optimizations with br_if and select.

6.1 Future Work

Empirical Evaluation The first step would involve testing the actual performance costs of dynamic checks. We could then implement a type-checker and compiler for Wasm-prechk so we are able to perform experiments and measure the real performance benefit provided by Wasm-prechk. This would allow us to empirically test whether Wasm-prechk actually improves performance. Our plan is to implement Wasm-prechk in Rust building on the CraneLift compilier.

Better Type Annotations in Embedding Recall that the embedding of Wasm into Wasm-prechk from subsection 4.2.1 does not take advantage of the possibility of using type annotations on functions and blocks to check stronger guarantees about programs. This means that we could potentially miss opportunities to remove dynamic checks by using *prechk*-tagged instructions because we have less type information to work with. Type annotations can be added by the developer, who will then get stronger guarantees of correctness along with the potential for more optimizations. However, we would prefer

for the developer not to have to hand-annotate compiled Wasm. Instead, we could use static analysis to try to find the weakest preconditions that guarantee the safety of *prechk*-tagging instructions. We could also attempt to have have a compiler from a higher-level language to Wasm add annotations as a form of type preserving compilation similar to System F to Typed Assembly Language [5].

6.2 Limitations

Reasoning About Global Variables Reasoning about global variables is made difficult because static typechecking is restricted to within the module we are checking, this restricts the reasoning ability of Wasm-prechk and again could cause us to miss opportunities for optimizations. Thus, it is difficult to reason about global variables imported from another module. Concretely, imagine, in the jth module calling a function f that was imported from the ith module. The call instruction will be reduced to call $\{\text{inst } i,\ \text{func } f\}$ where i is the module index for the module instance where f is defined. Theoretically, f should not change the global variables in the jth module. However, it may call a function in the jth module which could change the globals in the jth module, and since we do not know what the behavior of f is statically within j, we have to assume the worst and can make no assumptions about the global variables after f returns.

Handling the Dynamic Resizing of Memory While linear memory chunks are initialized with a static size, Wasm allows dynamically growing memory using the grow_memory instruction. Currently, Wasm-prechk only supports typechecking *prechk*-tagged loads and stores based on the static size. It should be possible to reason about the size of memory being increased by inserting a dependency on the result of the grow_memory instruction. If the result is -1, we know that the memory remains the same size. Otherwise, the result will be equal to the new memory size. This would require more dependency in the type system then we currently have with indexed types, since static type values would depend on dynamic control flow.

Chapter 7

Conclusion

We have introduced Wasm-prechk, a low-level language that uses an expressive type system to potentially improve performance via the elimination of unnecessary run-time checks. Wasm-prechk is based on Wasm, an existing real-world language commonly used in performance-critical and untrusted contexts, where both safety and performance are critical. To ensure the safety of Wasm-prechk, we have proven the type safety of the Wasm-prechk language as well as showing a sound type erasure to Wasm, demonstrating that Wasm-prechk is at least as safe as Wasm. We have shown that an indexed type system can be used in a low-level language to reduce the number of dynamic checks required, without sacrificing safety and security guarantees or increasing the programmer's proof burden. We built a reference interpreter for Wasm-prechk to demonstrate the practicality of implementing a typechecker for Wasm-prechk. This demonstrates the usefulness of using expressive type systems as a practical tool to improve performance and ensure safety for low-level languages in real use cases.

Bibliography

[1] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS/ETAPS)*, 2008. URL `https://dl.acm.org/doi/10.5555/1792734.1792766`.

[2] M. Felleisen, R. B. Findler, and M. Flatt. *Semantics engineering with PLT Redex*. 2009. URL `https://redex.racket-lang.org/`.

[3] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications confining the Wily Hacker. In *Proceedings of the Conference on USENIX Security Symposium, Focusing on Applications of Cryptography (SSYM)*, 1996. URL `https://doi.org/10.5555/1267569.1267570`.

[4] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. F. Bastien. Bringing the web up to speed with WebAssembly. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017. URL `https://doi.org/10.1145/3062341.3062363`.

[5] J. G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems*, 21(3), 1999. URL `https://doi.org/10.1145/319301.319345`.

[6] G. C. Necula. Proof-carrying code. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, POPL '97, 1997. URL `https://doi.org/10.1145/263699.263712`.

[7] C. Reis, A. Moshchuk, and N. Oskov. Site isolation: Process separation for web sites within the browser. In *USENIX Security Symposium*

(USENIX Security), 2019. URL `https://www.usenix.org/conference/usenixsecurity19/presentation/reis`.

[8] D. Tarditi, J. G. Morrisett, P. Cheng, C. A. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1996. URL `https://doi.org/10.1145/231379.231414`.

[9] H. Xi and R. Harper. A dependently typed assembly language. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2001. URL `https://doi.org/10.1145/507635.507657`.

Appendix A

Complete Wasm-prechk Typing Judgment Definition

$$\boxed{tfi <: tfi}$$

$$\frac{\phi_0 \implies \phi_1 \qquad \phi_2 \implies \phi_3}{ti_1^*;\ l_1;\ \phi_1 \to ti_2^*;\ l_2;\ \phi_2 <: ti_1^*;\ l_1;\ \phi_0 \to ti_2^*;\ l_2;\ \phi_3}\ \textsc{Implies}$$

$$\boxed{C \vdash e^* : tfi}$$

$$\frac{}{C \vdash \mathsf{unreachable} : ti_1^*;\ l_1;\ \phi_1 \to ti_2^*;\ l_2;\ \phi_2}\ \textsc{Unreachable}$$

$$\frac{}{C \vdash \mathsf{nop} : \epsilon;\ l;\ \phi \to \epsilon;\ l;\ \phi}\ \textsc{Nop} \qquad \frac{}{C \vdash \mathsf{drop} : (t\ a);\ l;\ \phi \to \epsilon;\ l;\ \phi}\ \textsc{Drop}$$

$$\frac{a \notin \phi}{C \vdash t.\mathsf{const}\ c : \epsilon;\ l;\ \phi \to (t\ a);\ l;\ \phi, (t\ a), (= a (t\ c))}\ \textsc{Const}$$

$$\frac{a_3 \notin \phi}{\begin{array}{l} C \vdash t.binop : (t\ a_1)\ (t\ a_2);\ l;\ \phi \\ \qquad \to (t\ a_3);\ l;\ \phi, (t\ a_3), (= a_3\ (\|binop\|\ a_1\ a_2)) \end{array}}\ \textsc{Binop}$$

$$\frac{a_3 \notin \phi}{\begin{array}{l} C \vdash t.testop : (t\ a_1)\ l;\ \phi \\ \qquad \to (\mathsf{i32}\ a_2);\ l;\ \phi, (t\ a_2), (= a_2\ (\|testop\|\ a_1)) \end{array}}\ \textsc{Testop}$$

$$\frac{a_3 \notin \phi}{\begin{array}{l} C \vdash t.relop : (t\ a_1)\ (t\ a_2);\ l;\ \phi \\ \qquad \to (t\ a_3);\ l;\ \phi, (t\ a_3), (= a_3\ (\|relop\|\ a_1\ a_2)) \end{array}}\ \textsc{Relop}$$

$$\frac{a_3 \notin \phi}{\begin{array}{l} C \vdash \mathsf{select} : (t\ a_1)\ (t\ a_2)\ (i32\ a);\ l;\ \phi \\ \qquad \to (t\ a_3);\ l;\ \phi, (t\ a_3), \\ \qquad\qquad (\text{if}\ (= a\ (\mathsf{i32}\ 0))\ (= a_3\ a_2)\ (= a_3\ a_1)) \end{array}}\ \textsc{Select}$$

$$\frac{C_2, \text{label}\ (ti_2^*;\ l_2;\ \phi_2) \vdash e^* : ti_1^*;\ l_1;\ \phi_1 \to ti_2^*;\ l_2;\ \phi_2}{C \vdash \mathsf{block}\ (ti_1^*;\ l_1;\ \phi_1 \to ti_2^*;\ l_2;\ \phi_2)\ e^*\ \mathsf{end} : ti_1^*;\ l_1;\ \phi_1 \to ti_2^*;\ l_2;\ \phi_2}\ \textsc{Block}$$

$$\frac{C_2, \text{label } (ti_1^*; l_1; \phi_1)^* \vdash e^* : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2}{C \vdash \mathsf{loop}\ (ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2)\ e^*\ \mathsf{end} : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2}\ \textsc{Loop}$$

$$\frac{\begin{array}{c} C_2, \text{label } (ti_2^*; l_2; \phi_2) \vdash e_1^* : ti_1^*; l_1; \phi_1, \neg(= a\ (\mathsf{i32}\ 0)) \to ti_2^*; l_2; \phi_2 \\ C_2, \text{label } (ti_2^*; l_2; \phi_2) \vdash e_2^* : ti_1^*; l_1; \phi_1, (= a\ (\mathsf{i32}\ 0))) \to ti_2^*; l_2; \phi_2 \end{array}}{C \vdash \mathsf{if}\ (ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2)\ e_1^*\ \mathsf{else}\ e_2^*\ \mathsf{end} : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2}\ \textsc{If}$$

$$\frac{C_{\text{label}}(i) = ti^*; l_1; \phi_1}{C \vdash \mathsf{br}\ i : ti_1^*\ ti^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2}\ \textsc{Br}$$

$$\frac{C_{\text{return}} = ti^*; l_1; \phi_1}{C \vdash \mathsf{return} : ti_1^*\ ti^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2}\ \textsc{Return}$$

$$\frac{C_{\text{label}}(i) = ti^*; l_1; \phi_1, \neg(= a\ (\mathsf{i32}\ 0))}{C \vdash \mathsf{br\_if}\ i : ti^*\ a; l_1; \phi_1 \to ti^*; l_1; \phi_1, (= a\ (\mathsf{i32}\ 0))}\ \textsc{Br-If}$$

$$\frac{(C_{\text{label}}(i) = ti^*; l_1; \phi_i)^+ \qquad (\phi_1 \implies \phi_i)^n}{C \vdash \mathsf{br\_table}\ i^+ : ti_1^*\ ti^*\ a; l_1; \phi_1 \to ti_2^*; l_2; \phi_2}\ \textsc{Br-Table}$$

$$\frac{C_{\text{func}}(i) = ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2}{C \vdash \mathsf{call}\ i : ti_1^*; l; \phi_1 \to ti_2^*; l; \phi_1, \phi_2}\ \textsc{Call}$$

$$\frac{C_{\text{table}}(i) = (j, tfi_2^*)}{\begin{array}{l} C \vdash \mathsf{call\_indirect}\ ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2 \\ \quad : ti_1^*\ (i32\ a); l; \phi_1 \to ti_2^*; l; \phi_1, \phi_2 \end{array}}\ \textsc{Call-Indirect}$$

$$\frac{C_{\text{local}}(i) = t \qquad l(i) = (t\ a) \qquad a_2 \notin \phi}{C \vdash \mathsf{get\_local}\ i : \epsilon; l; \phi \to (t\ a_2); l; \phi, (t\ a_2), (= a_2\ a)}\ \textsc{Get-Local}$$

$$\frac{C_{\text{local}}(i) = t \qquad l_2 = l_1[i := (t\ a)]}{C \vdash \mathsf{set\_local}\ i : (t\ a); l_1; \phi \to \epsilon; l_2; \phi}\ \textsc{Set-Local}$$

$$\frac{C_{\text{local}}(i) = t \qquad l_2 = l_1[i := (t\ a)] \qquad a_2 \notin \phi}{C \vdash \mathsf{tee\_local}\ i : (t\ a); l_1; \phi \to (t\ a_2); l_2; \phi, (t\ a_2), (= a_2\ a)}\ \textsc{Tee-Local}$$

$$\frac{C_{\text{global}}(i) = \text{mut}^?\ t \qquad a \notin \phi}{C \vdash \mathsf{get\_global}\ i : \epsilon; l; \phi \to (t\ a); l; \phi, (t\ a)}\ \textsc{Get-Global}$$

$$\frac{C_{\text{global}}(i) = \text{mut}\ t}{C \vdash \mathsf{set\_global}\ i : (t\ a); l; \phi \to \epsilon; l; \phi}\ \textsc{Set-Global}$$

$$\frac{C_{\text{memory}} = n \qquad 2^{align} \leq (|tp| <)^?|t| \qquad a_2 \notin \phi}{C \vdash t.\mathsf{load}\ (tp\_sx)^?\ align\ o : (\mathsf{i32}\ a_1); l; \phi \to (t\ a_2); l; \phi, (t\ a_2)}\ \textsc{Mem-Load}$$

$$\frac{C_{\text{memory}} = n \qquad 2^{align} \leq (|tp| <)^?|t|}{C \vdash t.\mathsf{store}\ tp^?\ align\ o : (\mathsf{i32}\ a_1)\ (t\ a_2); l; \phi \to \epsilon; l; \phi}\ \textsc{Mem-Store}$$

$$\frac{C_{\text{memory}} = n \qquad a \notin \phi}{C \vdash \mathsf{current\_memory}\ : \epsilon; l; \phi \to (\mathsf{i32}\ a); l; \phi, (\mathsf{i32}\ a)}\ \textsc{Current-Memory}$$

$$\frac{C_{\text{memory}} = n \qquad a_2 \notin \phi}{C \vdash \mathsf{grow\_memory}\ : (\mathsf{i32}\ a_1); l; \phi \to (\mathsf{i32}\ a_2); l; \phi, (\mathsf{i32}\ a_2)}\ \textsc{Grow-Memory}$$

$$\frac{}{C \vdash \epsilon : \epsilon; l; \phi \to \epsilon; l; \phi}\ \textsc{Empty}$$

$$\frac{C \vdash e^* : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2}{C \vdash e^* : ti^*\ ti_1^*; l_1; \phi_1 \to ti^*\ ti_2^*; l_2; \phi_2}\ \textsc{Stack-Poly}$$

$$\frac{C \vdash e_1^* : ti_1^*;\ l_1;\ \phi_1 \to ti_2^*;\ l_2;\ \phi_2 \qquad C \vdash e_2 : ti_2^*;\ l_2;\ \phi_2 \to ti_3^*;\ l_3;\ \phi_3}{C \vdash e_1^*\ e_2 : ti_1^*;\ l_1;\ \phi_1 \to ti_3^*;\ l_3;\ \phi_3}\ \textsc{Composition}$$

$$\frac{ti_1^*;\ l_1;\ \phi_1 \to ti_2^*;\ l_2;\ \phi_2 <: ti_1^*;\ l_1;\ \phi_0 \to ti_2^*;\ l_2;\ \phi_3 \qquad C \vdash e^* \to ti_1^*;\ l_1;\ \phi_1 \to ti_2^*;\ l_2;\ \phi_2}{C \vdash e^* \to ti_1^*;\ l_1;\ \phi_0 \to ti_2^*;\ l_2;\ \phi_3}\ \textsc{Subtyping}$$

$$\frac{\phi \implies \neg(=\ a_2\ 0) \qquad a_3 \notin \phi}{\begin{array}{l} C \vdash t.\mathsf{div}_{\mathsf{prechk}} : (t\ a_1)\ (t\ a_2);\ l;\ \phi \\ \qquad\qquad \to (t\ a_3);\ l;\ \phi, (t\ a_3), (=\ a_3\ (\|div\|\ a_1\ a_2)) \end{array}}\ \textsc{Div-Prechk}$$

$$\frac{\begin{array}{c} C_{\text{memory}} = n \qquad 2^{align} \leq (|tp| <)^?|t| \qquad a_3 \notin \phi \\ \phi \implies (\mathsf{ge}\ (\mathsf{add}\ a_1\ (\mathsf{i32}\ o))(\mathsf{i32}\ 0)), \\ \qquad (\mathsf{le}\ (\mathsf{add}\ a_1\ (\mathsf{add}\ (\mathsf{i32}\ o + width)))\ (\mathsf{i32}\ n * 64\text{Ki})) \end{array}}{\begin{array}{l} C \vdash t.\mathsf{load}_{\mathsf{prechk}}\ (tp\_sx)^?\ align\ o : (\mathsf{i32}\ a_1);\ l;\ \phi \\ \qquad\qquad \to (t\ a_2);\ l;\ \phi, (t\ a_2) \end{array}}\ \textsc{Load-Prechk}$$

$$\frac{\begin{array}{c} C_{\text{memory}} = n \qquad 2^{align} \leq (|tp| <)^?|t| \\ \phi \implies (\mathsf{ge}\ (\mathsf{add}\ a_1\ (\mathsf{i32}\ o))\ (\mathsf{i32}\ 0)), \\ \qquad (\mathsf{le}\ (\mathsf{add}\ a_1\ (\mathsf{add}\ (\mathsf{i32}\ o + width)))\ (\mathsf{i32}\ n * 64\text{Ki})) \end{array}}{C \vdash t.\mathsf{store}_{\mathsf{prechk}}\ tp^?\ align\ o : (\mathsf{i32}\ a_1)\ (t\ a_2);\ l;\ \phi \to \epsilon;\ l;\ \phi}\ \textsc{Store-Prechk}$$

$$\frac{\begin{array}{c} C_{table}(i) = (n, (tfi_2 ...)) \\ \phi \implies (\mathsf{gt}\ n\ a) \wedge (\mathsf{le}\ (\mathsf{i32}\ 0)\ a) \\ tfis = (tfi_2 ...) \qquad tfi = ti_1^*;\ l_1;\ \phi_1 \to ti_2^*;\ l_2;\ \phi_2 \\ \forall 0 < i \leq n.\ (\phi \implies \neg(=\ (\mathsf{i32}\ i)\ a)) \vee\ tfis(i) <: tfi \end{array}}{\begin{array}{l} C \vdash \mathsf{call\_indirect}_{\mathsf{prechk}}\ tfi : ti_1^*\ (i32\ a);\ l;\ \phi_1 \\ \qquad\qquad \to ti_2^*;\ l;\ \phi_1, \phi_2 \end{array}}\ \textsc{Call-Indirect-Prechk}$$

Appendix B

Metatheory Proofs

B.1 Sound Erasure Proofs

Lemma 13. Sound Static Typing Erasure

If $C \vdash e^* : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$,
then $erase_C(C) \vdash erase_{e^*}(e^*) : erase_{tfi}(ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2)$

Proof. We proceed by induction over typing derivations. Most proof cases are omitted as they are simple, but we provide a few to give an idea of what the proofs look like. Intuitively, we want to show that erasing the typing derivation produces a valid Wasm typing derivation.

For most of the cases, the sequence of instructions e^* contains only a single instruction e_2, so we elide the step of turning $erase_{e^*}(e^*)$ into $erase_{e^*}(e_2)$.

- Case: $C \vdash t.binop : (t\ a_1)\ (t\ a_2); l_1; \phi_1 \to (t\ a_3); l_1; \phi_1, (t\ a_3), (= a_3\ (binop\ a_1\ a_2))$

  We want to show that

  $$erase_C(C) \vdash erase_{e^*}(t.binop) : erase_{tfi}((t\ a_1)\ (t\ a_2); l_1; \phi_1 \to (t\ a_3); l_1; \phi_1, (t\ a_3), (= a_3\ (binop\ a_1\ a_2)))$$

  By the definition of $erase_e$, we want to show that $erase_C(C) \vdash t.binop : t\ t \to t$ is valid in Wasm.

Trivially, we have $erase_C(C) \vdash t.binop : t\ t \to t$ by Rule WASM-BINOP, since Rule WASM-BINOP works under any module type context.

- Case: $C \vdash \mathsf{block}\ (t_1\ a_1)^*; l_1; \phi_1 \to (t_2\ a_2)^*; l_2; \phi_2\ e^*\ \mathsf{end} : (t_1\ a_1)^*; l_1; \phi_1 \to (t_2\ a_2)^*; l_2; \phi_2$

  We want to show that

$$erase_C(C) \vdash erase_e(\mathsf{block}\ (t_1\ a_1)^*; l_1; \phi_1 \to (t_2\ a_2)^*; l_2; \phi_2\ e^*\ \mathsf{end} : (t_1\ a_1)^*; l_1; \phi_1 \to (t_2\ a_2)^*; l_2; \phi_2)$$

  By the definition of $erase_e$ and $erase_{tfi}$ (we also perform the step of erasing indexed function types here to avoid an extra step), we want to show that $erase_C(C) \vdash \mathbf{block}\ \ t_1^* \to t_2^* e^*\ \mathbf{end} : t_1^* \to t_2^*$ is a valid Wasm judgment.

  This proof is slightly more involved, since the derivation for this rule includes a premise that

$$C, \mathrm{label}((t_2\ a_2)^*; l_2; \phi_2) \vdash e^* : (t_1\ a_1)^*; l_1; \phi_1 \to (t_2\ a_2)^*; l_2; \phi_2$$

  By the inductive hypothesis for the well-typedness of e^*, we have that

$$erase_C(C, \mathrm{label}((t_2\ a_2)^*; l_2; \phi_2)) \vdash erase_{e^*}(e^*) : erase_{tfi}((t_1\ a_1)^*; l_1; \phi_1 \to (t_2\ a_2)^*; l_2; \phi_2)$$

  Then we have $erase_C(C), \mathrm{label}(t_2^*) \vdash erase_{e^*}(e^*) : t_1^* \to t_2^*$ by definition of $erase_C$ and $erase_{tfi}$.

  Now that we have satisfied the premise,

$$erase_C(C) \vdash \mathbf{block}\ \ t_1^* \to t_2^* e^*\ \mathbf{end} : t_1^* \to t_2^*$$

by Rule WASM-BLOCK.

- Case: $C \vdash \mathsf{br}\ \ i : (t_1\ a_1)^*; l_1; \phi_1 \to (t_2\ a_2)^*; l_2; \phi_2$

  We want to show that

  $$\begin{aligned} &erase_C(C) \vdash erase_e(\mathsf{br}\ i) \\ &\qquad : erase_{tfi}((t_1\ a_1)^*; l_1; \phi_1 \to (t_2\ a_2)^*; l_2; \phi_2) \end{aligned}$$

  We have to reason about $erase_C(C)$ because the typing judgment relies on the label stack from C.

  From $C \vdash \mathsf{br}\ \ i : (t_1\ a_1)^*; l_1; \phi_1 \to (t_2\ a_2)^*; l_2; \phi_2$, we have that $C_{\text{label}}(i) = (t_1\ a_1)^*; l_1; \phi_1$, since it is a premise.

  Then $erase_C(C)_{\text{label}}(i) = t_1^*$, by the definition of $erase_C$.

  $$\begin{aligned} &erase_C(C) \vdash erase_e(\mathsf{br}\ i) \\ &\qquad : erase_{tfi}((t_1\ a_1)^*; l_1; \phi_1 \\ &\qquad\ \to (t_2\ a_2)^*; l_2; \phi_2) \end{aligned} \quad = erase_C(C) \vdash \mathsf{br}\ i : t_1^* \to t_2^*$$

  Recall that $erase_C(C)_{\text{label}}(i) = t_1^* \to t_2^*$, then $erase_C(C) \vdash \mathsf{br}\ i : t_1^* \to t_2^*$ by Rule WASM-BR.

- Case: $C \vdash \mathsf{call}\ \ i : (t_1\ a_1)^*; l_1; \phi_1 \to (t_2\ a_2)^*; l_2; \phi_2$

  We want to show that

  $$\begin{aligned} &erase_C(C) \vdash erase_e(\mathsf{call}\ i) \\ &\qquad : erase_{tfi}((t_1\ a_1)^*; l_1; \phi_1 \to (t_2\ a_2)^*; l_2; \phi_2) \end{aligned}$$

  We again have to reason about $erase_C(C)$ because the typing judgment relies on the function type from C.

  From $C \vdash \mathsf{call}\ \ i : (t_1\ a_1)^*; l_1; \phi_1 \to (t_2\ a_2)^*; l_2; \phi_2$, we have that $C_{\text{func}}(i) = (t_1\ a_1)^*; l_1; \phi_1 \to (t_2\ a_2)^*; l_2; \phi_2$, since it is a premise.

Then, $erase_C(C)_{\text{func}}(i) = erase_{tfi}((t_1\ a_1)^*;\ l_1;\ \phi_1 \to (t_2\ a_2)^*;\ l_2;\ \phi_2) = t_1^* \to t_2^*$, by the definition of $erase_C$.

$$\begin{aligned} erase_C(C) \vdash erase_e(\mathsf{call}\ i) \\ : erase_{tfi}((t_1\ a_1)^*;\ l_1;\ \phi_1 \\ \to (t_2\ a_2)^*;\ l_2;\ \phi_2) \end{aligned} = erase_C(C) \vdash \mathsf{call}\ i : t_1^* \to t_2^*$$

Recall that $erase_C(C)_{\text{func}}(i) = t_1^* \to t_2^*$.

Then $erase_C(C) \vdash \mathsf{call}\ i : t_1^* \to t_2^*$ by Rule Wasm-Call.

- Case: $C \vdash \mathsf{set\_local}\ \ i : (t\ a);\ l_1;\ \phi_1 \to \epsilon;\ l_1[i := a];\ \phi_1$

  We want to show that

$$\begin{aligned} erase_C(C) \vdash erase_e(\mathsf{set\_local}\ i) \\ : erase_{tfi}((t\ a);\ l_1;\ \phi_1 \to \epsilon;\ l_1[i := a];\ \phi_1) \end{aligned}$$

  We again have to reason about $erase_C(C)$ because the typing judgment relies on the local variable types from C.

  From $C \vdash \mathsf{set\_local}\ \ i : (t\ a);\ l_1;\ \phi_1 \to \epsilon;\ l_1[i := a];\ \phi_1$, we have that $C_{\text{local}}(i) = t$, since it is a premise.

  Then, we have that $erase_C(C)_{\text{local}}(i) = t$, by the definition of $erase_C$.

$$\begin{aligned} erase_C(C) \vdash erase_e(\mathsf{set\_local}\ i) \\ : erase_{tfi}((t\ a)^*;\ l_1;\ \phi_1 \\ \to \epsilon;\ l_1[i := a];\ \phi_1) \end{aligned} = erase_C(C) \vdash \mathsf{set\_local}\ i : t \to \epsilon$$

  Recall that $erase_C(C)_{\text{local}}(i) = t \to \epsilon$, then $erase_C(C) \vdash \mathsf{set\_local}\ i : t \to \epsilon$ by Rule Wasm-Set-Local.

- Case: $C \vdash e_1^*\ e_2 : (t_1\ a_1)^*;\ l_1;\ \phi_1 \to (t_2\ a_2);\ l_2;\ \phi_2$

We want to show that

$$\begin{aligned} &erase_C(C) \vdash erase_e(e_1^*\ e_2) \\ &\qquad : erase_{tfi}((t_1\ a_1)^*;\ l_1;\ \phi_1 \to (t_2\ a_2);\ l_2;\ \phi_2) \end{aligned}$$

For this typing rule, we must invoke the inductive hypothesis twice: one on the sequence e_1^* and once on the instruction e_2. Then we must show that we can compose the erased subsequence together to get a well-typed sequence.

We know that $C \vdash e_1^* : (t_1\ a_1)^*;\ l_1;\ \phi_1 \to (t_3\ a_3);\ l_3;\ \phi_3$ and that $C \vdash e_2 : (t_3\ a_3)^*;\ l_3;\ \phi_3 \to (t_2\ a_2);\ l_2;\ \phi_2$ because they are premieses of Rule Composition which we have assumed to hold.

$erase_C(C) \vdash erase_{e^*}(e_1^*) : erase_{tfi}((t_1\ a_1)^*;\ l_1;\ \phi_1 \to (t_3\ a_3);\ l_3;\ \phi_3)$ by the inductive hypothesis on e_1^*.

$erase_C(C) \vdash erase_{e^*}(e_1^*) : t_1^* \to t_3^*$ by definition of $erase_{tfi}$.

$erase_C(C) \vdash erase_{e^*}(e_2) : erase_{tfi}((t_3\ a_3)^*;\ l_3;\ \phi_3 \to (t_2\ a_2);\ l_2;\ \phi_2)$, by the inductive hypothesis on e_2.

$erase_C(C) \vdash erase_{e^*}(e_2^*) : t_3^* \to t_2^*$ by definition of $erase_{tfi}$.

$$\begin{aligned} &erase_C(C) \vdash erase_{e^*}(e_1^*\ e_2) \\ &\qquad : erase_{tfi}((t_1\ a_1)^*;\ l_1;\ \phi_1 \\ &\qquad \to (t_2\ a_2);\ l_2;\ \phi_2) \end{aligned} \quad = \begin{aligned} &erase_C(C) \vdash\ erase_{e^*}(e_1^*) \\ &\qquad\qquad\ erase_{e^*}(e_2) \\ &\qquad\qquad : t_1^* \to t_2^* \end{aligned}$$

Recall that we have $erase_C(C) \vdash erase_{e^*}(e_1^*) : t_1^* \to t_3^*$ and that $erase_C(C) \vdash erase_{e^*}(e_2^*) : t_3^* \to t_2^*$ by definition of $erase_{tfi}$.

Then, $erase_C(C) \vdash erase_{e^*}(e_1^*)\ erase_{e^*}(e_2) : t_1^* \to t_2^*$ by Rule Wasm-Composition.

□

Lemma 14. Sound Admin Typing Erasure

If $S; C \vdash e^* : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$,
then $erase_S(S); erase_C(C) \vdash erase_{e^*}(e^*) : erase_{tfi}(ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2)$

Proof. We proceed by induction over typing rules. In addition to the prior cases from Lemma Sound Static Typing Erasure, which trivially still hold since the value of S does not matter to those rules, we add proves for a few administrative typing rules, which may refer to S. Again, several proof cases are omitted as they are simple.

- $S; C \vdash \mathsf{label}_n\{e_0^*\}\ e^*\ \mathsf{end} : \epsilon; l_1; \phi_1 \to (t_2\ a_2)^n; l_2; \phi_2$

  We must show that

  $$erase_S(S); erase_C(C) \vdash erase_{e^*}(e_0^*) : erase_{tfi}((t_3\ a_3)^*; l_3; \phi_3 \to (t_2\ a_2)^n; l_2; \phi_2)$$

  and

  $$erase_S(S); erase_C(C, \text{label}((t_3\ a_3)^*; l_3; \phi_3)) \vdash erase_{e^*}(e^*) : erase_{tfi}(\epsilon; l_1; \phi_1 \to (t_3\ a_3)^*; l_3; \phi_3)$$

  as they are the premises of typechecking a label block in Wasm.

  We have that $S; C \vdash e_0^* : (t_3\ a_3)^*; l_3; \phi_3 \to (t_2\ a_2)^n; l_2; \phi_2$ since it is a premise of Rule Label which we have assumed to hold.

  Then, by the inductive hypothesis for the stored instructions e_0^* being well typed, we have that

  $$erase_S(S); erase_C(C) \vdash erase_{e^*}(e_0^*) : erase_{tfi}((t_3\ a_3)^*; l_3; \phi_3 \to (t_2\ a_2)^*; l_2; \phi_2)$$

  $S; C, \text{label}((t_3\ a_3)^*; l_3; \phi_3)) \vdash e^* : \epsilon; l_1; \phi_1 \to (t_3\ a_3)^*; l_3; \phi_3$, because it is a premise of Rule Label which we have assumed to hold.

By the inductive hypothesis for the body e^* being well typed, we have that

$$\begin{aligned} erase_S(S);\, erase_C(C, \text{label}((t_3\ a_3)^*;\, l_3;\, \phi_3)) \\ \vdash erase_{e^*}(e^*) \\ : erase_{tfi}(\epsilon;\, l_1;\, \phi_1 \to (t_3\ a_3)^*;\, l_3;\, \phi_3) \end{aligned}$$

- $S;\, C \vdash \mathsf{local}_n\{i;\, v^*\}\ e^*\ \mathsf{end} : \epsilon;\, l_1;\, \phi_1 \to (t_2\ a_2)^n;\, l_2;\, \phi_2$

  The premise of this rule relies on Rule Code with a non-empty return postcondition, which we have not yet proved sound erasure for, so instead we must derive Rule Code for the erased program.

  Thus, we must show that $(\vdash v : t_v)^*$ and

  $$\begin{aligned} erase_S(S);\, erase_C(S_{\text{inst}}(i), \text{local } t_v^*,\ \text{return}((t_2\ a_2)^n;\, l_2;\, \phi_2)) \\ \vdash erase_{e^*}(e^*) \\ : erase_{tfi}(\epsilon;\, l_1;\, \phi_1 \to (t_2\ a_2)^n;\, l_2;\, \phi_2) \end{aligned}$$

  We have Rule Code as a premise of Rule Local, which we have assumed to hold.

  Therefore, $(\vdash v : t_v)^*$ trivially, since neither values nor primitive types are affected by erasure.

  We also know that

  $$\begin{aligned} S;\, S_{\text{inst}}(i), \text{local } t_v^*,\ \text{return}((t_2\ a_2)^n;\, l_2;\, \phi_2) \vdash e^*\ \mathsf{end} \\ : \epsilon;\, l_1;\, \phi_1 \to (t_2\ a_2)^n;\, l_2;\, \phi_2 \end{aligned}$$

  Therefore, by the inductive hypothesis of the well-typedness of e^*, we have that

  $$\begin{aligned} erase_S(S);\, erase_C(S_{\text{inst}}(i), \text{local } t_v^*,\ \text{return}((t_2\ a_2)^n;\, l_2;\, \phi_2)) \\ \vdash erase_{e^*}(e^*) \\ : erase_{tfi}(\epsilon;\, l_1;\, \phi_1 \to (t_2\ a_2)^n;\, l_2;\, \phi_2) \end{aligned}$$

□

B.2 Subject Reduction Lemmas and Proofs

Lemma 15. INVERSION

1. If $S; C \vdash t.\mathsf{const}\ c : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$, then $ti_2^* = ti_1^*\ (t\ a)$, $l_1 = l_2$, and $\phi_1, (t\ a), (= a\ (t\ c)) \implies \phi_2$.

2. If $S; C \vdash t.binop : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$, then $ti_1^* = ti^*\ (t\ a_1)\ (t\ a_2)$, $ti_2^* = ti^*\ (t\ a_3)$, $l_1 = l_2$, and $\phi_1, (t\ a_3), (= a_3\ (binop\ a_1\ a_2)) \implies \phi_2$.

3. If $S; C \vdash t.testop : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$, then $ti_1^* = ti^*\ (t\ a_1)$, $ti_2^* = ti^*\ (\mathsf{i32}\ a_2)$, $l_1 = l_2$, and $\phi_1, (\mathsf{i32}\ a_2), (= a_2\ (testop\ a_1)) \implies \phi_2$.

4. If $S; C \vdash t.relop : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$, then $ti_1^* = ti^*\ (t\ a_1)\ (t\ a_2)$, $ti_2^* = ti^*\ (\mathsf{i32}\ a_3)$, $l_1 = l_2$, and $\phi_1, (\mathsf{i32}\ a_3), (= a_3\ (relop\ a_1\ a_2)) \implies \phi_2$.

5. If $S; C \vdash \mathsf{nop} : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$, then $ti_1^* = ti_2^*$, $l_1 = l_2$, and $\phi_1 \implies \phi_2$.

6. If $S; C \vdash \mathsf{drop} : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$, then $ti_1^* = ti_0^*\ (t\ a)$, $ti_2^* = ti_0^*$, $l_1 = l_2$, and $\phi_1 \implies \phi_2$.

7. If $S; C \vdash \mathsf{select} : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$,
then $ti_1^* = ti_0^*\ (t\ a_1)\ (t\ a_2)\ (\mathsf{i32}\ a_3)$, $ti_2^* = ti_0^*; (t\ a)$, $l_1 = l_2$, and $\phi_1, (t\ a), (\mathrm{if}(= a_3\ (\mathsf{i32}\ 0))\ (= a_1\ a)\ (= a_2\ a)) \implies \phi_2$.

8. If $S; C \vdash \mathsf{block}\ (ti_3^*; l_3; \phi_3 \to ti_4^m; l_4; \phi_4)$
$e^*\ \mathsf{end} : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$
then $ti_1^* = ti_0^*\ ti_3^*$, $ti_2^* = ti_0^*\ ti_4^m$, $l_1 = l_3$, $l_2 = l_4$, $\phi_1 \implies \phi_3, \phi_4 \implies \phi_2$, and $S; C, \mathrm{label}(ti_4^m; l_4; \phi_4) \vdash e^* : ti_3^*; l_3; \phi_3 \to ti_4^m; l_4; \phi_4$.

9. If $S; C \vdash \mathsf{loop}\ (ti_3^n; l_3; \phi_3 \to ti_4^m; l_4; \phi_4)\ e^*\ \mathsf{end} : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$, then $ti_1^* = ti_0^*\ ti_3^n$, $ti_2^* = ti_0^*\ ti_4^m$, $l_1 = l_3$, $l_2 = l_4$, $\phi_1 \implies \phi_3, \phi_4 \implies \phi_2$, and $S; C, \mathrm{label}(ti_3^n; l_3; \phi_3) \vdash e^* : ti_3^n; l_3; \phi_3 \to ti_4^m; l_4; \phi_4$.

10. If $S; C \vdash \mathsf{if}\ (ti_3^n; l_3; \phi_3 \to ti_4^m; l_4; \phi_4)\ e_1^*\ \mathsf{else}\ e_2^*\ \mathsf{end} : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$, then $ti_1^* = ti_0^*\ ti_3^n\ (\mathsf{i32}\ a)$, $ti_2^* = ti_0^*\ ti_4^m$, $l_1 = l_3$, $l_2 = l_4$, $\phi_1 \implies \phi_3$, $\phi_4 \implies \phi_2$, $S; C, \mathrm{label}(ti_4^m; l_4; \phi_4) \vdash e_1^* : ti_3^n; l_3; \phi_3, \neg(= a\ (\mathsf{i32}\ 0)) \to ti_4^m; l_4; \phi_4$, and $S; C, \mathrm{label}(ti_4^m; l_4; \phi_4) \vdash e_2^* : ti_3^n; l_3; \phi_3, (= a\ (\mathsf{i32}\ 0)) \to ti_4^m; l_4; \phi_4$.

11. If $S; C \vdash \mathsf{label}_n\{e_0^*\}e^*\ \mathsf{end} : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$, then $ti_2^* = ti_1^*\ ti^*$, $S; C \vdash e_0^* : ti_3^*; l_3; \phi_3 \to ti^*; l_2; \phi_2$, and $S; C, \mathrm{label}\ (ti_3^*; l_3; \phi_3) \vdash e^* : \epsilon; l_1; \phi_1 \to ti^*; l_2; \phi_2$

12. If $S; C \vdash \mathsf{br}\ i : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$, then $ti_1^* = ti_3^*\ ti^*$, $C_{\mathrm{label}}(i) = ti^*; l_1; \phi_3$, and $\phi_1 \implies \phi_3$.

13. If $S; C \vdash \mathsf{br\_if}\ i : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$, then $ti_1^* = ti_2^*\ (\mathsf{i32}\ a)$, $l_1 = l_2$, $C_{\mathrm{label}}(i) = ti_2^*; l_1; \phi_3, \neg(= a\ (\mathsf{i32}\ 0))$, $\phi_1 \implies \phi_3$, and $\phi_1, (= a\ (\mathsf{i32}\ 0)) \implies \phi_2$.

14. If $S; C \vdash \mathsf{br\_table}\ i^+ : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$, then $ti_1^* = ti_2^*\ (\mathsf{i32}\ a)$, $l_1 = l_2$, $C_{\mathrm{label}}(i) = ti_2^*; l_1; \phi_3$, $\phi_1 \implies \phi_3$, and $\phi_1 \implies \phi_2$.

15. If $S; C \vdash \mathsf{call}\ i : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$, then $ti_1^* = ti_0^*\ ti_3^*$, $ti_2^* = ti_0^*\ ti_4^*$, $C_{\mathrm{func}}(j) = ti_3^*; l_3; \phi_3 \to ti_4^*; l_4; \phi_4$, $l_2 = l_1$, $\phi_1 \implies \phi_3$, and $\phi_3, \phi_4 \implies \phi_2$.

16. If $S; C \vdash \mathsf{call\_indirect}\ ti_3^*; l_3; \phi_3 \to ti_4^*; l_4; \phi_4 : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$, then $ti_1^* = ti_0^*\ ti_3^*$, $ti_2^* = ti_0^*\ ti_4^*$, $l_2 = l_1$, $\phi_1 \implies \phi_3$, and $\phi_3, \phi_4 \implies \phi_2$.

17. If $S; C \vdash \mathsf{call}\ cl : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$, then $ti_2^* = ti_1^*\ ti_4^m$, $l_2 = l_1$, $\phi_1 \implies \phi_3$, $\phi_4 \implies \phi_2$, and $S \vdash cl : \epsilon; ti_3^n\ (t\ a)^n; \phi_3, (t\ a)^k, (= a\ (t\ 0))^k \to ti_4^m; l_4; \phi_4$

18. If $S; C \vdash \mathsf{local}_n\{i; v_l^*\}\ e^*\ \mathsf{end} : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$, then $ti_2^* = ti_1^*\ ti^n$, $l_1 = l_2$, $S; (ti^n; l_3; \phi_3) \vdash_i v_l^*; e^* : ti^n; l_3; \phi_3$, and $\phi_1, \phi_3 \implies \phi_2$.

19. If $S; C \vdash \mathsf{return} : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$, then $ti_1^* = ti_3^*\ ti^*$, $l_1 = l_3$, $C_{\mathrm{return}} = ti^*; l_3; \phi_3$, and $\phi_1 \implies \phi_3$.

20. If $S; C \vdash \mathsf{get\_local}\ i : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$, then $ti_2^* = ti_1^*\ (t\ a)$, $l_1 = l_2$, $l_1(i) = (t\ a)$, and $\phi_1, (t\ a_2), (= a_2\ a) \implies \phi_2$.

21. If $S; C \vdash \mathsf{set\_local}\ i : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$, then $ti_1^* = ti_2^*\ (t\ a)$, $l_2 = l_1[i := (t\ a)]$, $C_{\text{local}}(i) = t$, and $\phi_1 \implies \phi_2$.

22. If $S; C \vdash \mathsf{tee\_local}\ i : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$, then $ti_1^* = ti^*\ (t\ a)$, $ti_2^* = ti^*\ (t\ a_2)$, $l_2 = l_1[i := (t\ a)]$, $C_{\text{local}}(i) = t$, and $\phi_1, (t\ a_2), (= a_2\ a) \implies \phi_2$.

23. If $S; C \vdash \mathsf{get\_global}\ i : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$, then $ti_2^* = ti_1^*\ (t\ a)$, $l_1 = l_2$, $C_{\text{global}}(i) = \text{mut}^?\ t$, and $\phi_1, (t\ a) \implies \phi_2$.

24. If $S; C \vdash \mathsf{set\_global}\ i : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$, then $ti_1^* = ti_2^*\ (t\ a)$, $l_1 = l_2$, $C_{\text{global}}(i) = \text{mut}\ t$, and $\phi_1 \implies \phi_2$.

25. If $S; C \vdash t.\mathsf{load}\ (tp\_sx)^?\ align\ o : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$, then $ti_1^* = ti^*\ (\mathsf{i32}\ a_1)$, $ti_2^* = ti^*\ (t\ a_2)$, $l_1 = l_2$, $C_{\text{memory}} = n$, $2^{align} \leq (|tp| <)^?|t|$, and $\phi_1, (t\ a_2) \implies \phi_2$.

26. If $S; C \vdash t.\mathsf{store}\ tp^?\ align\ o : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$, then $ti_1^* = ti_2^*\ (\mathsf{i32}\ a_1)\ (t\ a_2)$, $l_1 = l_2$, $C_{\text{memory}} = n$, $2^{align} \leq (|tp| <)^?|t|$, and $\phi_1 \implies \phi_2$.

27. If $S; C \vdash \mathsf{current\_memory} : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$, then $ti_2^* = ti_1^*\ (\mathsf{i32}\ a)$, $l_1 = l_2$, $C_{\text{memory}} = n$, and $\phi_1, (\mathsf{i32}\ a) \implies \phi_2$.

28. If $S; C \vdash \mathsf{grow\_memory} : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$, then $ti_1^* = ti^*\ (\mathsf{i32}\ a_1)$, $ti_2^* = ti^*\ (\mathsf{i32}\ a_2)$, $l_1 = l_2$, $C_{\text{memory}} = n$, and $\phi_1, (\mathsf{i32}\ a_2) \implies \phi_2$.

29. If $S; C \vdash e_1^*\ e_2 : ti_1^*; l_1; \phi_1 \to ti_3^*; l_3; \phi_3$, then $S; C \vdash e_1^* : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$, and $S; C \vdash e_2 : ti_2^*; l_2; \phi_2 \to ti_3^*; l_3; \phi_3$.

Proof. Proof omitted, but follows from induction over typing derivations. □

Lemma 16. Subject Reduction for Code

If $S; (ti; l; \phi)^? \vdash v^*; e^* : ti; l; \phi$, $\vdash s : S$, (we omit this on rules that do not use the store) and $s; v^*; e^* \hookrightarrow s'; v'^*; e'^*$, then $S; (ti; l; \phi)^? \vdash v'^*; e'^* : ti; l; \phi$, and $\vdash s' : S$ (we omit this on rules that do not change the store)

Proof. By induction on reduction.

- Case: $S;\ (ti^*;\ l;\ \phi)^?\ \vdash_i\ v_1^j\ (t.\mathsf{const}\ c)\ v_2^k;\ \mathsf{get\_local}\ j : ti^*;\ l;\ \phi$
  $\wedge\ v_1^j\ (t.\mathsf{const}\ c)\ v_2^k;\ \mathsf{get\_local}\ j \hookrightarrow (t.\mathsf{const}\ c)$

  We want to show that $S;\ (ti;\ l;\ \phi)^?\ \vdash_i\ v_1^j\ (t.\mathsf{const}\ c)\ v_2^k;\ (t.\mathsf{const}\ c) : ti^*;\ l;\ \phi$.

  We know $\vdash\ (t.\mathsf{const}\ c) : (t\ a);\ \circ, (t\ a), (=\ a\ (t\ c))$ and $S;\ C \vdash\ \mathsf{get\_local}\ j : \epsilon;\ l_1;\ \phi_1^j, ((t\ a), (=\ a\ (t\ c))), \phi_2^k \to ti^*;\ l;\ \phi$, because they are premises of Rule Code that we have assumed to hold.

  By Lemma Inversion on Rule Get-Local, $ti^* = (t\ a_2)$, $l_1 = l$, and $\phi_1^j, ((t\ a), (=\ a\ (t\ c))), \phi_2^k, (t\ a_2), (=\ a_2\ a) \implies \phi$.

  Now we will reconstruct the same type for $v_1^j\ (t.\mathsf{const}\ c)\ v_2^k;\ (t.\mathsf{const}\ c)$.

  We have $S;\ S_{\mathrm{inst}}(i) \vdash\ (t.\mathsf{const}\ c) : \epsilon;\ l;\ \phi_1^j, ((t\ a), (=\ a\ (t\ c))), \phi_2^k \to (t\ a_2);\ l;\ \phi_1^j, ((t\ a), (=\ a\ (t\ c))), \phi_2^k, (t\ a_2), (=\ a_2\ (t\ c))$ by Rule Const.

  Further, $((t\ a), (=\ a\ (t\ c))), (t\ a_2), (=\ a_2\ (t\ c)) \implies ((t\ a), (=\ a\ (t\ c))), (t\ a_2), (=\ a_2\ a)$ by $\implies$.

  Then, $S;\ S_{\mathrm{inst}}(i) \vdash\ (t.\mathsf{const}\ c) : \epsilon;\ l;\ \phi_1^j, ((t\ a), (=\ a\ (t\ c))), \phi_2^k \to (t\ a_2);\ l;\ \phi$ by Rule Subtyping.

  Therefore, $S;\ (ti^*;\ l;\ \phi)^?\ \vdash_i\ v_1^j\ (t.\mathsf{const}\ c)\ v_2^k;\ (t.\mathsf{const}\ c) : ti^*;\ l;\ \phi$ by Rule Code.

- Case: $S;\ (ti^*;\ l;\ \phi)^? \vdash_i\ v_1^j\ (t.\mathsf{const}\ c)\ v_2^k;\ (t.\mathsf{const}\ c')\ (\mathsf{set\_local}\ j) : ti^*;\ l;\ \phi$
  $\wedge\ v_1^j\ (t.\mathsf{const}\ c)\ v_2^k;\ (t.\mathsf{const}\ c')\ \mathsf{set\_local}\ j \hookrightarrow v_1^j\ (t.\mathsf{const}\ c')\ v_2^k;\ \epsilon$

  We want to show that

  $$S;\ (ti^*;\ l;\ \phi)^?\ \vdash_i\ v_1^j\ (t.\mathsf{const}\ c')\ v_2^k;\ (t.\mathsf{const}\ c')\ (\mathsf{set\_local}\ j) : ti^*;\ l;\ \phi$$

  We know $\vdash\ (t.\mathsf{const}\ c) : (t\ a);\ (\circ, (t\ a), (=\ a\ (t\ c))),\ S;\ S_{\mathrm{inst}}(i) \vdash\ (t.\mathsf{const}\ c')\ (\mathsf{set\_local}\ j) : \epsilon : l_1;\ \phi_1^j, (\circ, (t\ a), (=\ a\ (t\ c))), \phi_2^k \to ti^*;\ l;\ \phi$, and $l_1(j) = (t\ a)$, and $C_{\mathrm{local}}(j) = t$ because they are premises of Rule Code that we have assumed to hold.

By Lemma Inversion on Rule Composition, $S;\ S_{\text{inst}}(i) \vdash (t.\text{const}\ c') : \epsilon;\ l_1;\ \phi_1^j, (\circ, (t\ a), (=\ a\ (t\ c))), \phi_2^k \to ti_3^*;\ l_3;\ \phi_3$, $S;\ S_{\text{inst}}(i) \vdash \text{set\_local}\ j : ti_3^*;\ l_3;\ \phi_3 \to ti^*;\ l;\ \phi$.

Recall that $t = C_{\text{local}}(j)$, then by Lemma Inversion on Rule Set-Local we have $ti_3^* = ti^*\ (t\ a')$, $l = l_3[j := (t\ a')]$, and $\phi_3 \implies \phi$.

Then, by Lemma Inversion on Rule Const, $ti^* = \epsilon$, $l_1 = l_3$, and $\phi_1^j, (\circ, (t\ a), (=\ a\ (t\ c))), \phi_2^k, (t\ a'), (=\ a'\ (t\ c')) \implies \phi_3$.

Now we have all the information we need to derive the same type for $(t.\text{const}\ c')$

We have $S;\ S_{\text{inst}}(i) \vdash \epsilon : \epsilon;\ l;\ \phi \to \epsilon;\ l;\ \phi$ by Rule Empty.

Then, $S;\ S_{\text{inst}}(i) \vdash \epsilon : \epsilon;\ l;\ \phi_1^j, (\circ, (t\ a), (=\ a\ (t\ c))), \phi_2^k, (t\ a'), (=\ a'\ (t\ c')) \to \epsilon;\ l;\ \phi$ by Rule Subtyping.

Since a is fresh, $\phi_1^j, \phi_2^k, (t\ a'), (=\ a'\ (t\ c')) \implies \phi_1^j, (\circ, (t\ a), (=\ a\ (t\ c))), \phi_2^k, (t\ a'), (=\ a'\ (t\ c'))$.

Then, $S;\ S_{\text{inst}}(i) \vdash \epsilon : \epsilon;\ l;\ \phi_1^j, \phi_2^k, (t\ a'), (=\ a'\ (t\ c')) \to \epsilon;\ l;\ \phi$ by Rule Subtyping.

Further, $\vdash (t.\text{const}\ c') : (t\ a');\ \circ, (t\ a'), (=\ a'\ (t\ c'))$ by Rule Admin-Const.

Therefore, $S;\ (ti^*;\ l;\ \phi)^? \vdash_i v_1^j\ (t.\text{const}\ c')\ v_2^k;\ \epsilon : ti^*;\ l;\ \phi$ by Rule Code.

- Case: $S;\ (ti^*;\ l;\ \phi)^? \vdash_i v^*;\ \text{get\_global}\ j : ti^*;\ l;\ \phi$
  $\wedge\ s;\ \text{get\_global}\ j \hookrightarrow_i s_{\text{glob}}(i, j)$

  We want to show that $S;\ (ti^*;\ l;\ \phi)^? \vdash_i v^*;\ s_{\text{glob}}(i, j) : ti^*;\ l;\ \phi$

  We have $\vdash v : ti_v;\ \phi_v)^*$ and $S;\ S_{\text{inst}}(i) \vdash \text{get\_global}\ j : \epsilon;\ ti_v^*;\ \phi_v^* \to ti^*;\ l;\ \phi$ because they are premises of Rule Code that we have assumed to hold.

  Then, by Lemma Inversion on Rule Get-Global, $ti^* = (t\ a)$, $l = l_1$, $C_{\text{global}}(j) = \text{mut}^? t$, and $\phi_v^*, (t\ a) \implies \phi$.

  Recall that we assume $\vdash s : S$, then we know $S \vdash s_{\text{inst}}(i) : C$ because it is a premise of Rule Store.

Recall that $S_{\text{global}}(i,j) = \text{mut}^? t$, then $\vdash s_{\text{glob}}(i,j) : (t\ a_1);\ \phi_1$ because it is a premise of Rule INSTANCE that we have assumed to hold.

Now, we can show that $s_{\text{glob}}(i,j)$ has the appropriate type.

We have $S;\ S_{\text{inst}}(i) \vdash t.\text{const}\ c : \epsilon;\ l;\ \phi_v^* \to (t\ a);\ l;\ \phi_v^*, (t\ a), (= a\ (t\ c))$, where $t.\text{const}\ c = s_{\text{glob}}(i,j)$, by Rule CONST.

We also know $(t\ a), (= a\ (t\ c)) \implies (t\ a)$.

Thus, $S;\ S_{\text{inst}}(i) \vdash s_{\text{glob}}(i,j) : \epsilon;\ l;\ \phi_v^* \to (t\ a);\ l;\ \phi$ by Rule SUBTYPING.

Recall $\vdash v : ti_v;\ \phi_v)^*$, then $S;\ (ti^*;\ l;\ \phi)^? \vdash_i v^*;\ s_{\text{glob}}(i,j) : (t\ a);\ l;\ \phi$ by Rule CODE, having assumed that the other premises hold.

- Case: $S;\ (ti^*;\ l;\ \phi)^? \vdash v_l^*;\ (t.\text{const}\ c)\ (\text{set\_global}\ j) : ti^*;\ l;\ \phi$
  $\wedge\ s;\ (t.\text{const}\ c)\ (\text{set\_global}\ j) \hookrightarrow_i s';\ \epsilon$, where $s' = s$ with $\text{glob}(i,j) = (t.\text{const}\ c)$

  We want to show that $S;\ (ti^*;\ l;\ \phi)^? \vdash v_l^*;\ \epsilon : ti^*;\ l;\ \phi$.

  We have $\vdash (v_l^* : ti_v;\ \phi_v)^*$ and $S;\ S_{\text{inst}}(i) \vdash v\ (\text{set\_global}\ j) : \epsilon;\ ti_v^*;\ \phi_v^* \to ti^*;\ l;\ \phi$ because they are premises of Rule CODE that we have assumed to hold.

  Further, by Lemma INVERSION on Rule COMPOSITION, Rule SET-GLOBAL, and Rule CONST, $ti^* = \epsilon$, $l_1 = l$, $C_{\text{global}}(j) = \text{mut}\ t$, and $\phi_v^*, (t\ a), (= a\ (t\ c)) \implies \phi$.

  We have $S;\ S_{\text{inst}}(i) \vdash \epsilon : \epsilon;\ l;\ \phi \to \epsilon;\ l;\ \phi$ by Rule EMPTY.

  Since a is fresh, $\phi_v^* \implies \phi_v^*, (t\ a), (= a\ (t\ c))$.

  Thus $S;\ S_{\text{inst}}(i) \vdash \epsilon : \epsilon;\ l;\ \phi_v^* \to \epsilon;\ l;\ \phi$ by Rule SUBTYPING.

  Recall that $\vdash (v_l^* : ti_v;\ \phi_v)^*$, then $S;\ (ti^*;\ l;\ \phi)^? \vdash_i v_l^*;\ \epsilon : ti^*;\ l;\ \phi$ by Rule CODE.

  Now we must ensure that the new store s' is well typed: $\vdash s' : S$.

  Recall $\vdash s : S$, then $S_{\text{glob}}(i,j) = \text{mut}^?\ t$ and $s_{\text{glob}}(i,j) = (t.\text{const}\ c')$ where $\vdash (t.\text{const}\ c') : (t\ a_g);\ \circ, (t\ a_g), (= a_g\ (t\ c'))$ because it is a premise of $\vdash s : S$.

We know $\vdash (t.\mathsf{const}\ c) : (t.\mathsf{const}\ c) : (t\ a_{g2}); \circ, (t\ a_{g2}), (= a_{g2}\ (t\ c))$, and therefore $\vdash s' : S$ by Rule STORE.

- Case: $S; (ti^*; l; \phi)^? \vdash_i v^*; (\mathsf{i32.const}\ k)\ (t.\mathsf{load}\ align\ o) : ti^*; l; \phi$
  $\wedge\ s; (\mathsf{i32.const}\ k)\ (t.\mathsf{load}\ align\ o) \hookrightarrow_i t.\mathsf{const}\ \mathrm{const}_t(b^*)$,
  where $s_{\mathrm{mem}}(i, k+o, |t|) = b^*$

  We want to show that $S; (ti^*; l; \phi)^? \vdash_i v^*; (t.\mathsf{const}\ \mathrm{const}_t(b^*)) : ti^*; l; \phi$

  We know $(\vdash v : ti_v; \phi_v)^*$ and $S; C \vdash (\mathsf{i32.const}\ k)\ (t.\mathsf{load}\ align\ o) : \epsilon; ti_v^*; \phi_v^* \to ti^*; l; \phi$ because they are premises of Rule CODE which we have assumed to hold.

  Then, by Lemma INVERSION on Rule COMPOSITION, Rule CONST, Rule MEM-LOAD, we know $ti^* = (t\ a)$, $ti_v^* = l$, and $\phi_v^*, (t\ a) \implies \phi$.

  We have

  $$S; S_{\mathrm{inst}}(i) \vdash t.\mathsf{const}\ \mathrm{const}_t(b^*) : \epsilon; ti_v^*; \phi_v^* \to (t\ a); l; \phi_v^*, (t\ a), (= a\ (t\ c))$$

  by Rule CONST.

  Then, $S; S_{\mathrm{inst}}(i) \vdash (t.\mathsf{const}\ \mathrm{const}_t(b^*)) : \epsilon; ti_v^*; \phi_v^* \to (t\ a); l; \phi$ by Rule SUBTYPING.

  Recall $(\vdash v : ti_v; \phi_v)^*$, then $S; (ti^*; l; \phi)^? \vdash_i v^*; t.\mathsf{const}\ \mathrm{const}_t(b^*) : ti^*; l; \phi$ by Rule CODE.

- Case: $S; (ti^*; l; \phi)^? \vdash_i v^*; (\mathsf{i32.const}\ k)\ (t.\mathsf{load}\ tp_s x\ align\ o) : ti^*; l; \phi$
  $\wedge\ s; (\mathsf{i32.const}\ k)\ (t.\mathsf{load}\ tp_s x\ align\ o) \hookrightarrow_i s; t.\mathsf{const}\ \mathrm{const}_t^{sx}(b^*)$, where $s_{\mathrm{mem}}(i, k+o, |tp|) = b^*$

  Similar to above case, except with $|tp|$ replacing $|t|$ and $\mathrm{const}_t^{sx}(b^*)$ instead of $\mathrm{const}_t(b^*)$.

- Case: $S; (ti^*; l; \phi)^? \vdash_i v^*; (\mathsf{i32.const}\ k)\ (t.\mathsf{load}\ tp\_sx^?\ align\ o) : ti^*; l; \phi$
  $\wedge\ s; (\mathsf{i32.const}\ k)\ (t.\mathsf{load}\ tp\_sx^?\ align\ o) \hookrightarrow_i \mathsf{trap}$

We know $(\vdash v : ti_v;\ \phi_v)^*$ because it is a premise of Rule Code which we have assumed to hold.

We have $S;\ S_{\text{inst}}(i) \vdash \mathsf{trap} : \epsilon;\ ti_v^*;\ \phi_v^* \to ti^*;\ l;\ \phi$ by Rule Trap.

Then, $S;\ (ti^*;\ l;\ \phi)^? \vdash_i v^*;\ \mathsf{trap}$ by Rule Code.

- Case: $S;\ (ti^*;\ l;\ \phi)^? \vdash_i v^*;\ (\mathsf{i32.const}\ k)\ (t.\mathsf{const}\ c)\ (t.\mathsf{store}\ align\ o) : ti^*;\ l;\ \phi$
  $\wedge\ s;\ (\mathsf{i32.const}\ k)\ (t.\mathsf{const}\ c)\ (t.\mathsf{store}\ align\ o) \hookrightarrow_i s';\ \epsilon$, where $s' = s$ with $\text{mem}(i, k+o, |t|) = \text{bits}_t^{|t|}(c)$

  We know $(\vdash v : ti_v;\ \phi_v)^*$ and

$$S;\ S_{\text{inst}}(i) \vdash (\mathsf{i32.const}\ k)\ (t.\mathsf{const}\ c)\ (t.\mathsf{store}\ align\ o) : \epsilon;\ ti_v^*;\ \phi_v^* \to ti^*;\ l;\ \phi$$

  because they are premises of Rule Code which we have assumed to hold.

  Then, by Lemma Inversion on Rule Composition, Rule Const, and Rule Mem-Store, we have $ti^* = \epsilon$, $ti_v = l$, and $\phi_v^*, (\mathsf{i32}\ a_1), (= a_1\ (\mathsf{i32}\ k)), (t\ a_2), (= a_2\ (t\ c)) \implies \phi$.

  Since a_1 and a_2 are fresh, $\phi_v^* \implies \phi$.

  We have $S;\ S_{\text{inst}}(i) \vdash \epsilon : \epsilon;\ l;\ \phi_v^* \to \epsilon;\ l;\ \phi_v^*$ by Rule Empty.

  Then, $S;\ S_{\text{inst}}(i) \vdash \epsilon : \epsilon;\ ti_v;\ \phi_v^* \to \epsilon;\ l;\ \phi$ by Rule Subtyping.

  Recall that $(\vdash v : ti_v;\ \phi_v)^*$. Therefore, $S;\ (ti^*;\ l;\ \phi)^? \vdash_i v^*;\ \epsilon : ti^*;\ l;\ \phi$ by Rule Code.

  Now we must ensure that the new store s' is well typed: $\vdash s' : S$.

  Recall $\vdash s : S$, then $S_{\text{mem}}(i) = n$ and $s_{\text{mem}}(i) = b^*$ where $n \leq |b^*|$ because it's a premise of Rule Store.

  Since $s' = s$ with $\text{mem}(i, k+o, |t|) = \text{bits}_t^{|t|}(c)$, then $|s'_t extmem(i) \models |s_t extmem(i)|$, and therefore $n \leq |s'_t extmem(i)|$, so $s' : S$ by Rule Store.

- Case: $S;\ (ti^*;\ l;\ \phi)^?\ \vdash_i\ v^*;\ (\mathsf{i32.const}\ k)\ (t.\mathsf{const}\ c)\ (t.\mathsf{store}\ tp\ align\ o) : ti_1^*;\ l_1;\ \phi_1 \to ti_2^*;\ l_2;\ \phi_2 \wedge \vdash s : S$
  $\wedge\ s;\ (\mathsf{i32.const}\ k)\ (t.\mathsf{const}\ c)\ (t.\mathsf{store}\ tp\ align\ o) \hookrightarrow_i\ s';\ \epsilon$, where $s' = s$ with $\mathrm{mem}(i, k+o, |tp|) = \mathrm{bits}_t^{|tp|}(c)$

  Similar to above case, except with $|tp|$ replacing $|t|$ and $\mathrm{const}_t^{sx}(b^*)$ instead of $\mathrm{const}_t(b^*)$.

- Case: $S;\ (ti^*;\ l;\ \phi)^?\ \vdash_i\ v^*;\ (\mathsf{i32.const}\ k)\ (t.\mathsf{const}\ c)\ (t.\mathsf{store}\ tp^?\ align\ o) : ti^*;\ l;\ \phi$
  $\wedge\ s;\ (\mathsf{i32.const}\ k)\ (t.\mathsf{const}\ c)\ (t.\mathsf{store}\ tp^?\ align\ o) \hookrightarrow_i \mathsf{trap}$

  We know $(\vdash v : ti_v;\ \phi_v)^*$ because it is a premise of Rule Code which we have assumed to hold.

  We have $S;\ S_{\mathrm{inst}}(i) \vdash \mathsf{trap} : \epsilon;\ ti_v^*;\ \phi_v^* \to ti^*;\ l;\ \phi$ by Rule Trap.

  Then, $S;\ (ti^*;\ l;\ \phi)^?\ \vdash_i\ v^*;\ \mathsf{trap}$ by Rule Code.

- Case: $S;\ (ti^*;\ l;\ \phi)^?\ \vdash_i\ v^*;\ \mathsf{current\_memory}\ : ti^*;\ l;\ \phi$
  $\wedge\ s;\ \mathsf{current\_memory}\ \hookrightarrow_i\ \mathsf{i32.const}\ |s_{\mathrm{mem}}(i, *)|/64\mathrm{Ki}$

  We know $(\vdash v : ti_v;\ \phi_v)^*$ and $S;\ S_{\mathrm{inst}}(i) \vdash \mathsf{current\_memory}\ : \epsilon;\ ti_v^*;\ \phi_v^* \to ti^*;\ l;\ \phi$ because they are premises of Rule Code which we have assumed to hold.

  Then, by Lemma Inversion on Rule Current-Memory, $ti^* = (\mathsf{i32}\ a)$, $ti_v^* = l$, and $\phi_v^*, (\mathsf{i32}\ a) \implies \phi$.

  Let $c = |s_{\mathrm{mem}}(i, *)|/64\mathrm{Ki}$. Although note that the actual value of c is irrelevant to the rest of the proof case.

  $S;\ S_{\mathrm{inst}}(i) \vdash \mathsf{i32.const}\ c : \epsilon;\ l;\ \phi_v^* \to (\mathsf{i32}\ a);\ l;\ \phi_v^*, (\mathsf{i32}\ a), (=\ a\ (\mathsf{i32}\ c))$ by Rule Const.

  $S;\ S_{\mathrm{inst}}(i) \vdash \mathsf{i32.const}\ c : \epsilon;\ ti^*;\ \phi_v^* \to (\mathsf{i32}\ a);\ l;\ \phi$ by Rule Subtyping.

  Recall that $(\vdash\ v\ :\ ti_v;\ \phi_v)^*$. Thus, $S;\ (ti^*;\ l;\ \phi)^?\ \vdash_i\ v^*;\ \mathsf{i32.const}\ c\ : ti^*;\ l;\ \phi$ by Rule Code.

- Case: $S;\ (ti^*;\ l;\ \phi)^?\ \vdash_i\ v^*;\ (\mathsf{i32.const}\ k)\ \mathsf{grow\_memory}\ : ti^*;\ l;\ \phi$
  $\wedge\ s;\ (\mathsf{i32.const}\ k)\ \mathsf{grow\_memory}\ \hookrightarrow_i\ s';\ \mathsf{i32.const}\ |s_{\mathrm{mem}}(i, *)|/64\mathrm{Ki}$,

where $s' = s$ with $\text{mem}(i, *) = s_{\text{mem}}(i, *)(0)^{k \cdot 64\text{Ki}}$

We have $(\vdash v : ti_v; \phi_v)^*$ and $S; S_{\text{inst}}(i) \vdash (\text{i32.const } k)$ grow_memory $: \epsilon; ti_v^*; \phi_v^* \to ti^*; l; \phi$ because they are premises of Rule Code which we have assumed to hold.

By Lemma Inversion on Rule Composition, Rule Const, and Rule Grow-Memory, we also have $ti^* = (\text{i32 } a_1)$, $ti_v^* = l$, and $\phi_v^{\phi_v}, (\text{i32 } a_2), (= a_2\ (\text{i32 } k)), (\text{i32 } a_1) \implies \phi$.

Further, $S_{\text{mem}}(i) \leq |s_{\text{mem}}(i, *)|$ because it is a premise of Rule Store on $\vdash s : S$, which we have assumed to hold.

Let $c = \text{i32.const } |s_{\text{mem}}(i, *)|/64\text{Ki}$. Although note that the actual value of c is irrelevant to the rest of the proof case.

$$S; S_{\text{inst}}(i) \vdash \text{i32.const } c : \epsilon; l; \phi_v^* \to (\text{i32 } a_1); l; \phi_v^*, (\text{i32 } a_1), (= a_1\ (\text{i32 } c))$$

by Rule Const.

Since a_2 is fresh, $\phi_v^*, (\text{i32 } a_1) \implies \phi$.

$S; S_{\text{inst}}(i) \vdash \text{i32.const } c : \epsilon; ti_v^*; \phi_v^* \to (\text{i32 } a_1); l; \phi$ by Rule Subtyping.

$S; (ti^*; l; \phi)^? \vdash_i \text{i32.const } c : ti^*; l; \phi$ by Rule Code.

Now we must ensure that the new store s' is well typed: $\vdash s' : S$.

Recall $\vdash s : S$, then $S_{\text{mem}}(i) = n$ and $s_{\text{mem}}(i) = b^*$ where $n \leq |b^*|$ because it's a premise of Rule Store.

Since $s' = s$ with $\text{mem}(i, *) = s_{\text{mem}}(i, *)(0)^{k \cdot 64\text{Ki}}$, then $|s'_{\text{mem}}(i)| > |s_{\text{mem}}(i)|$, and therefore $n \leq |s'_{\text{mem}}(i)|$, so $s' : S$ by Rule Store.

- Case: $S; (ti^*; l; \phi)^? \vdash_i v^*; (\text{i32.const } k)$ grow_memory $: ti^*; l; \phi$
  $\wedge\ s; (\text{i32.const } k)$ grow_memory $\hookrightarrow_i \text{i32.const } (-1)$

  Same as above case since the value of c is irrelevant (and can therefore be -1).

- Case: $S;\ (ti^*;\ l;\ \phi)^?\ \vdash_j\ v_0^*;\ \mathsf{local}\{i;\ v^*\}\ e^*\ :\ ti^*;\ l;\ \phi$
  $\wedge\ s;\ v_0^*;\ \mathsf{local}\{i;\ v^*\}\ e^*\ \hookrightarrow_j\ s';\ v_0^*;\ \mathsf{local}\{i;\ v'^*\}\ e'^*$
  where $s;\ v^*;\ e^*\ \hookrightarrow_i\ s';\ v'^*;\ e'^*$

  We want to show that $\vdash_j\ s';\ v_0^*;\ \mathsf{local}\{i;\ v'^*\}\ e'^*\ :\ ti^*;\ l;\ \phi$.

  First, we will derive the type of the body of the local block.

  We have $S;\ S_{\text{inst}}(j)\ \vdash\ \mathsf{local}\{i;\ v^*\}\ e^*\ :\ \epsilon;\ ti_v^*;\ \phi_{v0}^*\ \rightarrow\ ti^*;\ l;\ \phi$ where $(\vdash\ v_0\ :\ ti_v;\ \phi_{v0})^*$ because they are premises of Rule Code.

  Then, $S;\ (ti^*;\ l;\ \phi)\ \vdash_i\ v^*;\ e^*\ :\ ti^*;\ l;\ \phi_0$, where $\phi_0\ \implies\ \phi$ by Lemma Inversion on Rule Local.

  Now, we invoke the inductive hypothesis and use it to rebuild the original type.

  Since $S;\ (ti^*;\ l;\ \phi)\ \vdash_i\ v^*;\ e^*\ :\ ti^*;\ l;\ \phi_0$, $s\ \vdash\ S$ and $s;\ v^*;\ e^*\ \hookrightarrow_i\ s';\ v'^*;\ e'^*$, then by the inductive hypothesis we know that $\vdash\ s'\ :\ S$ and

  $$S;\ (ti^*;\ l;\ \phi)\ \vdash_i\ v'^*;\ e'^*\ :\ ti^*;\ l;\ \phi_0$$

  Thus, $S;\ S_{\text{inst}}(j)\ \vdash\ \mathsf{local}\{i;\ v'^*\}\ e'^*\ :\ \epsilon;\ ti_v^*;\ \phi_{v0}^*\ \rightarrow\ ti^*;\ l;\ \phi$ by Rule Local.

  Finally, $S;\ (ti^*;\ l;\ \phi)^?\ \vdash_i\ v_0^*;\ \mathsf{local}\{i;\ v'^*\}\ e'^*\ :\ ti^*;\ l;\ \phi$ and $\vdash\ s'\ :\ S$.

- Case: $S;\ (ti^*;\ l;\ \phi)^?\ \vdash_i\ v^*;\ L^k[e^*]\ :\ ti^*;\ l;\ \phi$
  $\wedge\ s;\ v^*;\ L^k[e^*]\ \hookrightarrow_i\ s';\ v'^*;\ L^k[e'^*]$
  where $s;\ v^*;\ e^*\ \hookrightarrow_i\ s';\ v'^*;\ e'^*$

  We want to show that $S;\ (ti^*;\ l;\ \phi)^*\ \vdash_i\ L^k[e'^*]\ :\ ti^*;\ l;\ \phi$.

  First, we will derive the type of the body of the local context.

  We have $S;\ C\ \vdash\ L^k[e^*]\ :\ \epsilon;\ ti_v^*;\ \phi_v^*\ \rightarrow\ ti^*;\ l;\ \phi$,
  where $C = S_{\text{inst}}(i), \text{local}\ t_v^*, \text{return}\ (ti^*;\ l;\ \phi)^?$ because it is a premise of Rule Code.

  The intuition for the proof is that there is no requirement on what the label stack is of the module type context C under which $L^k[e^*]$ is

typed. Thus, we can reduce e^* outside of L^k, but with the module type context C as if it were inside of L^k.

The proof continues via induction on k.

- Case: $k = 0$

  $L^k[e^*] = v_0^*\ e^*\ e_0^*$.

  By Lemma Inversion on Rule Composition and Rule Const
  $S; C \vdash e^* : ti_1^*; ti_v^*; \phi_v^*, \phi_{v0}^* \to ti_2^*; l_2; \phi_2$ and
  $S; C \vdash e_0^* : ti_2^*; l_2; \phi_2 \to ti^*; l; \phi$.

  Since $s; v^*; e^*$ reduces, e^* can be typed with an empty stack precondition. Therefore $ti_2^* = ti_1^*\ ti_3^*$ and $S; C \vdash e^* : \epsilon; ti_v^*; \phi_v^*, \phi_{v0}^* \to ti_3^*; l_2; \phi_2$.

  Since ϕ_{v0}^* only contains fresh index variables, $\phi_v^* \implies \phi_v^*, \phi_{v0}^*$.

  $S; C \vdash e^* : \epsilon; ti_v^*; \phi_v^* \to ti_3^*; l_2; \phi_2$ by Rule Implies.

  Then, $S; (ti^*; l; \phi)^? \vdash_i v^*; e^* : ti_3^*; l_2; \phi_2$ by Rule Code.

  Now, we invoke the Lemma Subject Reduction for Code inductive hypothesis and rebuild the type using the reduced expression.

  Since $S; (ti^*; l; \phi)^* \vdash_i v^*; e^* : ti_3^*; l_2; \phi_2$, $s \vdash S$, and $s; v^*; e^* \hookrightarrow_i s'; v'; e'^*$, then by the inductive hypothesis we know that $\vdash s' : S$ and $S; (ti^*; l; \phi) \vdash_i v'^*; e'^* : ti_3^*; l_2; \phi_2$.

  $S; C \vdash e'^* : \epsilon; ti_{v'}^*; \phi_{v'}^* \to ti_3^*; l_2; \phi_2$ because it is a premise of Rule Code.

  $S; C \vdash v_0^* : \epsilon; ti_{v'}^*; \phi_{v'}^* \to ti_1^*; ti_{v'}^*; \phi_{v'}^*, \phi_{v0}^*$ by Rule Const.

  $S; C \vdash e'^* : ti_1^*; ti_{v'}^*; \phi_{v'}^*, \phi_{v0}^* \to ti_2^*; l_2; \phi_2$ by Rule Implies and Rule Stack-Poly.

  $S; C \vdash v_0^*\ e'^*\ e_0^* : \epsilon; ti_{v'}^*; \phi_{v'}^*, \phi_{v0}^* \to ti^*; l; \phi$ by Rule Composition.

  Therefore, $S; (ti^*; l; \phi)^* \vdash_i L^k[e'^*] : ti^*; l; \phi$ and $\vdash s' : S$.

- Case: $k > 0$

  $L^k[e^*] = v_k^*$ label$_n\{e_0^*\}\ L^{k-1}[e^*]$ end e_k^*.

By Lemma Inversion on Rule Const and Rule Composition, $S; C \vdash \mathsf{label}_n\{e_0^*\}\ L^{k-1}[e^*]\ \mathsf{end} : ti_1^*; ti_v^*; \phi_v^*, \phi_{vk}^* \to ti_2^*; l_2; \phi_2$ and $S; C \vdash e_k^* : ti_2^*; l_2; \phi_2 \to ti^*; l; \phi$.

By Lemma Inversion on Rule Label, $ti_2^* = ti_1^*\ ti_3^*$, $S; C \vdash e_0^* : ti_4^*; l_4; \phi_4 \to ti_3^*; l_2; \phi_2$, and $S; C, \text{label}\ (ti_4^*; l_4; \phi_4) \vdash L^{k-1}[e^*] : \epsilon; ti_v^*; \phi_v^*, \phi_{vk}^* \to ti_3^*; l_2; \phi_2$.

Since ϕ_{vk}^* only contains fresh index variables, $\phi_v^* \implies \phi_v^*, \phi_{vk}^*$.

$S; C, \text{label}\ (ti_4^*; l_4; \phi_4) \vdash L^{k-1}[e^*] : \epsilon; ti_v^*; \phi_v^* \to ti_3^*; l_2; \phi_2$ by Rule Implies.

$S; (ti^*; l; \phi)^? \vdash v^*; L^{k-1}[e^*] : ti_3^*; l_2; \phi_2$ by Rule Code.

Now, we invoke the inductive hypothesis on $L^{k-1}[e^*]$ and rebuild the type using the reduced expression.

Since $S; (ti^*; l; \phi)^* \vdash_i v^*; L^{k-1}[e^*] : ti_3^*; l_2; \phi_2$, $s \vdash S$, and $s; v^*; e^* \hookrightarrow_i s'; v'; e'^*$, then by the inductive hypothesis on $L^{k-1}[e^*]$ we know that $\vdash s' : S$ and $S; (ti^*; l; \phi) \vdash_i v'^*; L^{k-1}[e'^*] : ti_3^*; l_2; \phi_2$

$S; C, \text{label}\ \vdash L^{k-1}[e'^*] : \epsilon; ti_{v'}^*; \phi_{v'}^* \to ti_3^*; l_2; \phi_2$ because it is a premise of Rule Code.

$S; C \vdash v_k^* : \epsilon; ti_{v'}^*; \phi_{v'}^* \to ti_1^*; ti_{v'}^*; \phi_{v'}^*, \phi_{vk}^*$ by Rule Const.

$S; C \vdash \mathsf{label}_n\{e_0^*\}\ L^{k-1}[e'^*]\ \mathsf{end} : \epsilon; ti_{v'}^*; \phi_{v'}^* \to ti_3^*; l_2; \phi_2$ by Rule Label.

$S; C \vdash \mathsf{label}_n\{e_0^*\}\ L^{k-1}[e'^*]\ \mathsf{end} : ti_1^*; ti_{v'}^*; \phi_{v'}^*, \phi_{vk}^* \to ti_2^*; l_2; \phi_2$ by Rule Implies and Rule Stack-Poly.

$S; C \vdash v_k^*\ \mathsf{label}_n\{e_0^*\}\ L^{k-1}[e^*]\ \mathsf{end}\ e_k^* : \epsilon; ti_{v'}^*; \phi_{v'}^* \to ti^*; l; \phi$ by Rule Composition.

Therefore, $S; (ti^*; l; \phi)^* \vdash_i L^k[e'^*] : ti^*; l; \phi$ and $\vdash s' : S$.

- Otherwise: we have $(\vdash (t.\mathsf{const}\ c) : (t\ a); (\circ, (t\ a), (= a\ (t\ c))))^*$, and $S, S_{\text{inst}}(i) \vdash e^* : \epsilon; (t\ a)^*; (\circ, (t\ a), (= a\ (t\ c)))^* \to ti; l; \phi$

  By Lemma Subject Reduction Without Effects, we have $S, S_{\text{inst}}(i) \vdash e'^* : \epsilon; (t\ a)^*; (\circ, (t\ a), (= a\ (t\ c)))^* \to ti; l; \phi$.

  Then, $S; (ti^*; l; \phi)^? \vdash v^*; e'^* : ti; l; \phi$

□

Lemma 17. Subject Reduction Without Effects

If $S; C \vdash e^* : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$,
$\vdash s : S$ (note: we omit this for cases which do not use s),
and $s; v^*; e^* \hookrightarrow s; v^*; e'^*$,
then $S; C \vdash e'^* : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$

Proof. By case analysis on the reduction rules.

- $S; C \vdash L^0[\mathsf{trap}] : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$
  $\wedge\ L^0[\mathsf{trap}] \hookrightarrow \mathsf{trap}$

  This case is trivial since trap accepts any precondition and postcondition.
  Thus, $S; C \vdash \mathsf{trap} : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$ by Rule Trap.

- $S; C \vdash (t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ t.binop : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$
  $\wedge\ (t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ t.binop \hookrightarrow t.\mathsf{const}\ c$ where $c = binop(c_1, c_2)$

  We want to show that $S; C \vdash t.\mathsf{const}\ c : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$.

  We begin by reasoning about the type of the original instructions $(t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ t.binop$

  By Lemma Inversion on Rule Composition, Rule Const, and Rule Binop, we know that $ti_2^* = ti_1^*(t\ a_3)$, $l_2 = l_1$, and that

  $$\begin{array}{ll} \phi_1, (t\ a_1), (= a_1\ (t\ c_1)), & \implies \phi_2 \\ \quad (t\ a_2), (= a_2\ (t\ c_2)), & \\ \quad (t\ a_3), (= a_3\ (binop\ a_1\ a_2)) & \end{array}$$

  Now we will show that $t.\mathsf{const}\ c$ has the appropriate type.

  By *const*, $S; C \vdash t.\mathsf{const}\ c : \epsilon; l_1; \phi_1 \to (t\ a_3); l_1; \phi_1, (t\ a_3), (= a_3\ (t\ c))$.

  Because $c = binop_t(c_1, c_2)$, then by $\implies$,

  $$\begin{array}{ll} \phi_1, (t\ a), (= a\ (t\ c)) \implies & \phi_1, (t\ a_1), (= a_1\ (t\ c_1)), \\ & \quad (t\ a_2), (= a_2\ (t\ c_2)), \\ & \quad (t\ a_3), (= a_3\ (binop\ a_1 a_2)) \end{array}$$

Therefore, $S; C \vdash (t.\mathsf{const}\ c) : ti_1^*; l_1; \phi_1 \to ti_1^*\ (t\ a_3); l_1; \phi_2$, by Rule Stack-Poly and Rule Subtyping.

- $C \vdash (t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ t.binop : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$
  $\wedge\ (t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ t.binop \hookrightarrow \mathsf{trap}$

  This case is trivial since trap accepts any precondition and postcondition. Thus, $S; C \vdash \mathsf{trap} : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$ by Rule Trap.

- $C \vdash (t.\mathsf{const}\ c)\ t.testop : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$
  $\wedge\ (t.\mathsf{const}\ c)\ t.testop \hookrightarrow \mathsf{i32.const}\ c_2$ where $c_2 = testop(c)$

  We want to show that $S; C \vdash \mathsf{i32.const}\ c_2 : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$.

  First, we use our reasoning principles to get more information about the original type.

  By Lemma Inversion on Rule Composition, Rule Const, and Rule Testop, we know that $ti_2^* = ti_1^*\ (t\ a_2)$, $l_2 = l_1$, and that

  $$\begin{array}{l}\phi_1, (t\ a_1), (= a_1\ (t\ c)),\\ \quad (\mathsf{i32}\ a_2), (= a_2\ (testop\ a_1))\\ \quad \Longrightarrow\ \phi_2\end{array}$$

  Now we show that $\mathsf{i32.const}\ c_2$ has the appropriate type.

  By *const*, $C \vdash \mathsf{i32.const}\ c_2 : \epsilon; l_1; \phi_1 \to (\mathsf{i32}\ a_2); l_1; \phi_1, (\mathsf{i32}\ a_2), (= a_2\ (t\ c_2))$.

  Because $c_2 = testop(c)$, then by $\Longrightarrow$,

  $$\phi_1, (t\ a), (= a\ (t\ c_2)) \Longrightarrow \phi_1, (t\ a_1), (= a_1\ (t\ c)), (\mathsf{i32}\ a_2), (= a_2\ (testop\ a_1))$$

  Therefore, $S; C \vdash t.\mathsf{const}\ c_2 : ti_1^*; l_1; \phi_1 \to ti_1^*\ (t\ a_2); l_1; \phi_2$, by Rule Stack-Poly and Rule Subtyping

- $S; C \vdash (t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ t.relop : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$
  $\wedge\ (t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ t.relop \hookrightarrow t.\mathsf{const}\ c$ where $c = relop(c_1, c_2)$

This case is identical to the $(t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ t.binop \hookrightarrow t.\mathsf{const}\ c$ case, except that *binop* is replaced with *relop*.

- $S; C \vdash \mathsf{unreachable} : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$
  $\wedge\ \mathsf{unreachable} \hookrightarrow \mathsf{trap}$

  This case is once again trivial since trap accepts any precondition and postcondition. Thus, $S; C \vdash \mathsf{trap} : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$ by *trap*.

- $S; C \vdash \mathsf{nop} : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$
  $\wedge\ \mathsf{nop} \hookrightarrow \epsilon$

  We want to show that $S; C \vdash \epsilon : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$.

  This case follows from the fact that the postcondition $ti_2^*; l_2; \phi_2$ for nop must be immediately reachable from the precondition $ti_1^*; l_1; \phi_1$.

  By Lemma Inversion on Rule Nop, we know that $ti_2^* = ti_1^*$, $l_2 = l_1$, and $\phi_1 \implies \phi_2$.

  Then, $S; C \vdash \epsilon : \epsilon; l; g; \phi_1 \to \epsilon; l; g; \phi_1$ by Rule Empty.

  Thus, $S; C \vdash \epsilon ti_1^*; l; g; \phi_1 \to ti_1^*; l; g; \phi_2$ by Rule Stack-Poly and Rule Subtyping.

- $S; C \vdash (t.\mathsf{const}\ c)\ \mathsf{drop} : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$
  $\wedge\ (t.\mathsf{const}\ c)\ \mathsf{drop} \hookrightarrow \epsilon$

  We want to show that $S; C \vdash \epsilon : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$.

  Like the above case, this follow from the fact that the postcondition $ti_2^*; l_2; \phi_2$ must be immediately reachable from the precondition $ti_1^*; l_1; \phi_1$. However, there are a few extra steps since we now have to reason about two different instructions (and therefore Rule Composition).

  By Lemma Inversion on Rule Composition, Rule Const, and Rule Drop, we know that $ti_2^* = ti_1^*$, $l_2 = l_1$, and $\phi_1 \implies \phi_2$.

  By *empty*, $S; C \vdash \epsilon : \epsilon; l_1; \phi_1 \to \epsilon; l_1; \phi_1$.

  Thus, $S; C \vdash \epsilon ti_1^*; l; g; \phi_1 \to ti_1^*; l; g; \phi_2$ by Rule Stack-Poly and Rule Subtyping.

- Case: $S; C \vdash (t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ (\mathsf{i32.const}\ 0)\ \mathsf{select}$
  $: ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$
  $\wedge\ (t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ (\mathsf{i32.const}\ 0)\ \mathsf{select} \hookrightarrow (t.\mathsf{const}\ c_2)$

  We want to show that $S; C \vdash (t.\mathsf{const}\ c_2) : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$.

  First, we reason about what the original type must look like.

  By Lemma Inversion on Rule Composition, Rule Const, and Rule Select, we know that $ti_2^* = ti_1^*\ (a_3\ )$, $l_2 = l_1$, and

  $$\begin{array}{l}\phi_1, (t\ a_1), (= a_1\ (t\ c_1)),\\ \quad (t\ a_2), (= a_2\ (t\ c_2)),\\ \quad (\mathsf{i32}\ a), (= a\ (\mathsf{i32}\ 0)),\\ \quad (t\ a_3), (if\ (= a\ (\mathsf{i32}\ 0))\ (= a_3\ a_2)\ (= a_3\ a_1))\\ \implies \phi_2\end{array}$$

  Now we show that $(t.\mathsf{const}\ c_2)$ has the appropriate type.

  By Rule Const,
  $C \vdash (t.\mathsf{const}\ c_2) : \epsilon; l_1; \phi_1$
  $\to (t\ a_3); l_1; \phi_1, (t\ a_3), (= a_3\ (t\ c_2))$
  Then, $S; C \vdash (t.\mathsf{const}\ c_2) : ti_1^*; l_1; \phi_1 \to ti_1^*\ (t\ a_3); l_1; \phi_1, (t\ a_3), (= a_3\ (t\ c_2))$ by Rule Stack-Poly.

  By $\implies$, we have

  $$\begin{array}{rl}\phi_1, (t\ a_3), (= a_3\ (t\ c_2)) \implies & \phi_1, (t\ a_1), (= a_1\ (t\ c_1)),\\ & (t\ a_2), (= a_2\ (t\ c_2)),\\ & (\mathsf{i32}\ a), (= a\ (\mathsf{i32}\ 0)),\\ & (t\ a_3), (\mathrm{if}\ (= a\ (\mathsf{i32}\ 0))\\ & \qquad (= a_3\ a_2)\\ & \qquad (= a_3\ a_1))\end{array}$$

  Therefore, $S; C \vdash (t.\mathsf{const}\ c_2) : ti_1^*; l_1; \phi_1 \to ti_2^*\ (t\ a_3); l_1; \phi_2$ by $sub - typing$

- Case: $S; C \vdash (t.\mathsf{const}\ c)^n\ \mathsf{block}\ (ti_3^n; l_3; \phi_3 \to ti_4^m; l_4; \phi_4)\ e^*\ \mathsf{end}$:

$ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$
$\wedge\ (t.\mathsf{const}\ c)^n\ \mathsf{block}\ (ti_3^n; l_3; \phi_3 \to ti_4^m; l_4; \phi_4)\ e^*\ \mathsf{end}$
$\quad \hookrightarrow \mathsf{label}_m\{\epsilon\}\ (t.\mathsf{const}\ c)^n\ e^*\ \mathsf{end}$

We want to show that $\mathsf{label}_m\{\epsilon\}\ (t.\mathsf{const}\ c)^n\ e^*\ \mathsf{end} : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$.

First, we reason about $ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$.

We know $S; C \vdash \mathsf{block}\ (ti_3^n; l_3; \phi_3 \to ti_4^m; l_4; \phi_4)\ e^*\ \mathsf{end}$
$: ti_1^*\ (t\ a)^n; l_1; \phi_1, (t\ a)^n, (= a\ (t\ c))^n \to ti_2^*; l_2; \phi_2$
by Lemma Inversion on Rule Composition and Rule Const.

By Lemma Inversion on Rule Block, $l_1 = l_3$ and $l_2 = l_4$. We will use l_1, l_2 in place of l_3, l_4, respectively, for the remainder of the case.

Then, $S; C, \mathrm{label}(t_4^m; l_2; \phi_4) \vdash e^* : (t\ a)^n; l_1; \phi_3 \to ti_4^m; l_2; \phi_4$ because it is a premise of Rule Block which we have already assumed to hold.

Also, $(t\ a)^n = ti_3^n$, $ti_2^* = ti_1^*\ ti_4^m$, $\phi_1, (t\ a)^n, (= a\ (t\ c))^n \implies \phi_3$, and $\phi_4 \implies \phi_2$ by Lemma Inversion on Rule Block.

Now we have all the information we need to show that
$\mathsf{label}_m\{\epsilon\}\ (t.\mathsf{const}\ c)^n\ e^*\ \mathsf{end} : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$.

Remember that Rule Label uses the types of both the body $(t.\mathsf{const}\ c)^n\ e^*$ and the stored instructions ϵ.

First, we show the type of the body.

We have

$$S; C, \mathrm{label}(t_4^m; l_2; \phi_4) \vdash (t.\mathsf{const}\ c)^n : \epsilon; l_1; \phi_1 \to (t\ a)^n; l_1; \phi_1, (t\ a)^n, (= a\ (t\ c))^n$$

by Rule Const.

Then, since $\phi_1, (t\ a)^n, (= a\ (t\ c))^n \implies \phi_3$, we have

$$S; C, \mathrm{label}(t_3^n; l_1; \phi_3) \vdash (t.\mathsf{const}\ c)^n : \epsilon; l_1; \phi_1 \to (t\ a)^n; l_1; \phi_3$$

by Rule Subtyping.

Recall we have $S;\ C, \text{label}(t_4^m;\ l_2;\ \phi_4) \vdash e^* : (t\ a)^n;\ l_1;\ \phi_3 \to ti_4^m;\ l_2;\ \phi_4$.

Then $S;\ C, \text{label}(t_4^m;\ l_2;\ \phi_4) \vdash (t.\mathsf{const}\ c)^n\ e^* : \epsilon;\ l_1;\ \phi_1 \to ti_4^m;\ l_2;\ \phi_4$ by Rule Composition.

We have the type we want from the body. Now we get the type we want of the stored instructions. We already have the postcondition we want, $t_4^m;\ l_2;\ \phi_4$, in the label stack, so we want the stored instruction to just pass the information through. Since the stored instructions is ϵ, this is simple to show: we have $S;\ C \vdash \epsilon : ti_2^m;\ l_2;\ \phi_4 \to ti_2^m;\ l_2;\ \phi_4$ by Rule Empty and Rule Stack-Poly.

Therefore, $C \vdash \mathsf{label}_m\{\epsilon\}\ (t.\mathsf{const}\ c)^n\ e^*\ \mathsf{end} : \epsilon;\ l_1;\ \phi_1 \to ti_2^m;\ l_2;\ \phi_4$ by *label*.

Finally, since $\phi_4 \implies \phi_2$, $S;\ C \vdash \mathsf{label}_m\{\epsilon\}\ (t.\mathsf{const}\ c)^n\ e^*\ \mathsf{end} : ti_1^*;\ l_1;\ \phi_1 \to ti_1^*\ ti_4^m;\ l_2;\ \phi_2$ by Rule Stack-Poly and Rule Subtyping.

- Case: $S;\ C \vdash (t.\mathsf{const}\ c)^n\ \mathsf{loop}\ (ti_3^n;\ l_3;\ \phi_3 \to ti_4^m;\ l_4;\ \phi_4)\ e^*\ \mathsf{end} : ti_1^*;\ l_1;\ \phi_1 \to ti_2^*;\ l_2;\ \phi_2$
  $\wedge\ (t.\mathsf{const}\ c)^n\ \mathsf{loop}\ ti_3^n;\ l_3;\ \phi_3 \to ti_4^m;\ l_4;\ \phi_4\ e^*\ \mathsf{end}$
  $\hookrightarrow \mathsf{label}_n\{\mathsf{loop}\ ti_3^n;\ l_3;\ \phi_3 \to ti_4^m;\ l_4;\ \phi_4\ e^*\ \mathsf{end}\}\ (t.\mathsf{const}\ c)^n\ e^*\ \mathsf{end}$

  We want to show that $\mathsf{label}_n\ \{\mathsf{loop}\ ti_3^n;\ l_3;\ \phi_3 \to ti_4^m;\ l_4;\ \phi_4\ e^*\ \mathsf{end}\}\ (t.\mathsf{const}\ c)^n\ e^*\ \mathsf{end} : ti_1^*;\ l_1;\ \phi_1 \to ti_2^*;\ l_2;\ \phi_2$

  This rule is similar to the above one, except that we must reason a little more about the stored instructions since we are storing the loop.

  We start by figuring out what $ti_1^*;\ l_1;\ \phi_1 \to ti_2^*;\ l_2;\ \phi_2$ looks like.

  We know

  $$S;\ C \vdash \mathsf{loop}\ ti_3^n;\ l_3;\ \phi_3 \to ti_4^m;\ l_4;\ \phi_4\ e^*\ \mathsf{end} : ti_1^*\ (t\ a)^n;\ l_1;\ \phi_1, (t\ a)^n, (=\ a\ (t\ c))^n \to ti_2^*;\ l_2;\ \phi_2$$

  by Lemma Inversion on Rule Composition and Rule Const.

  By Lemma Inversion on Rule Loop, $l_1 = l_3$ and $l_2 = l_4$. We will use l_1, l_2 in place of l_3, l_4, respectively, for the remainder of the case.

Then, $S; C, \text{label}(t_4^m; l_2; \phi_4) \vdash e^* : (t\ a)^n; l_1; \phi_3 \to ti_4^m; l_2; \phi_4$ because it is a premise of Rule Loop which we have already assumed to hold.

Also, $(t\ a)^n = ti_3^n$, $ti_2^* = ti_1^*\ ti_4^m$, $\phi_1, (t\ a)^n, (= a\ (t\ c))^n \implies \phi_3$, and $\phi_4 \implies \phi_2$ by Lemma Inversion on Rule Loop.

Now we have all the information we need to show that

$$S; C \vdash \text{label}_n\{\text{loop } ti_3^n; l_3; \phi_3 \to ti_4^m; l_4; \phi_4\ e^*\ \text{end}\}\ (t.\text{const } c)^n\ e^*\ \text{end} : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$$

We have $S; C, \text{label}(t_3^n; l_1; \phi_3) \vdash (t.\text{const } c)^n : \epsilon; l_1; \phi_1 \to (t\ a)^n; l_1; \phi_1, (t\ a)^n, (= a\ (t\ c))^n$ by Rule Const.

Then, since $\phi_1, (t\ a)^n, (= a\ (t\ c))^n \implies \phi_3$, we have

$$S; C, \text{label}(t_3^n; l_1; \phi_3) \vdash (t.\text{const } c)^n : \epsilon; l_1; \phi_1 \to (t\ a)^n; l_1; \phi_3$$

by Rule Subtyping.

Recall that $S; C, \text{label}(t_3^n; l_1; \phi_3) \vdash e^* : ti_1^n; l_1; \phi_3 \to ti_2^m; l_1; \phi_4$.

Then $S; C, \text{label}(t_3^n; l_1; \phi_3) \vdash (t.\text{const } c)^n\ e^* : \epsilon; l_1; \phi_1 \to ti_4^m; l_2; \phi_4$ by *composition*.

We have the type we want from the body. Now we get the type we want of the stored instructions, which in this case is the loop. Since we already have the necessary type information of the body, we then have that $S; C \vdash \text{loop } tfi\ e^*\ \text{end} : (t\ a)^n; l_1; \phi_1, (t\ a)^n, (= a\ (t\ c))^n \to ti_4^m; l_2; \phi_4$ by Rule Loop.

Therefore, $S; C \vdash \text{label}_m\{\text{loop } tfi\ e^*\ \text{end}\}\ v^n\ e^*\ \text{end} : \epsilon; l_1; \phi_1 \to ti_4^m; l_2; \phi_4$ by Rule Label.

Finally, since $\phi_4 \implies \phi_2$, $S; C \vdash \text{label}_m\{\epsilon\}\ (t.\text{const } c)^n\ e^*\ \text{end} : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$ by Rule Stack-Poly and Rule Subtyping.

- Case: $S; C \vdash (\text{i32.const } 0)\ \text{if}\ (ti_3^n; l_3; \phi_3 \to ti_4^m; l_4; \phi_4)\ e_1^*\ \text{else}\ e_2^*\ \text{end} : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$

$\wedge$ (i32.const 0) if ti_3^n; l_3; $\phi_3 \to ti_4^m$; l_4; ϕ_4 e_1^* else e_2^* end
$\hookrightarrow$ block ti_3^n; l_3; $\phi_3 \to ti_4^m$; l_4; ϕ_4 e_2^* end

We want to show that

$$\text{block } ti_3^n;\ l_3;\ \phi_3 \to ti_4^m;\ l_4;\ \phi_4\ e_1^*\ \text{end} : ti_1^*;\ l_1;\ \phi_1 \to ti_2^*;\ l_2;\ \phi_2$$

First, we reason about ti_1^*; l_1; $\phi_1 \to ti_2^*$; l_2; ϕ_2.

We know

$$\begin{aligned} S;\ C \vdash\ & \text{if } ti_3^n;\ l_3;\ \phi_3 \to ti_4^m;\ l_4;\ \phi_4\ e_1^*\ \text{else}\ e_2^*\ \text{end} \\ & : ti_1^*\ (t\ a);\ l_1;\ \phi_1, (t\ a)^n, (=\ a\ (t\ 0))^n \to ti_2^*;\ l_2;\ \phi_2 \end{aligned}$$

by Lemma Inversion on Rule Composition and Rule Const.

Then, we have S; C, label(ti_4^m; l_4; ϕ_4) $\vdash e_2^* : ti_3^n$; l_3; $\phi_3, (=\ a\ (\text{i32}\ 0)) \to ti_4^m$; l_4; ϕ_4 because it is a premise of Rule If which we have assumed to hold.

By Lemma Inversion on Rule If, $ti_1^* = ti_0^*\ ti_3^n$ and $ti_2^* = ti_0^*\ ti_4^m$ for some ti_0^*, $l_1 = l_3$, $l_2 = l_4$, $\phi_1, (\text{i32}\ a), (=\ a\ (\text{i32}\ 0)) \implies \phi_3$, and $\phi_4 \implies \phi_2$.

Now, we show that block ti_3^n; l_3; $\phi_3 \to ti_4^m$; l_4; ϕ_4 e_1^* end : ti_1^*; l_1; $\phi_1 \to ti_2^*$; l_2; ϕ_2

S; $C \vdash$ block ti_3^n; l_3; $\phi_3, (=\ a\ (\text{i32}\ 0)) \to ti_4^m$; l_4; ϕ_4 e_2^* end by Rule Block.

Since a is fresh after reduction, $\phi_1 \implies \phi_1, (\text{i32}\ a), (=\ a\ (\text{i32}\ 0))$ by $\implies$.

Therefore, S; $C \vdash$ block ti_3^n; l_3; $\phi_3, (=\ a\ (\text{i32}\ 0)) \to ti_4^m$; l_4; ϕ_4 e_2^* end : $ti_0^*\ ti_3^n$; l_1; $\phi_1, (t\ a), (=\ a\ (\text{i32}\ 0)) \to s\ ti_0^*\ ti_4^m$; l_2; ϕ_2 by Rule Stack-Poly and Rule Subtyping.

- Case: $S; C \vdash (\mathsf{i32.const}\ k+1)\ \mathsf{if}\ (ti_3^n; l_3; \phi_3 \to ti_4^m; l_4; \phi_4)\ e_1^*$
  $\mathsf{else}\ e_2^*$
  end
  $: ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$
  $\wedge\ (\mathsf{i32.const}\ k+1)\ \mathsf{if}\ ti_3^n; l_3; \phi_3 \to ti_4^m; l_4; \phi_4\ e_1^*\ \mathsf{else}\ e_2^*\ \mathsf{end}$
  $\hookrightarrow \mathsf{block}\ ti_3^n; l_3; \phi_3 \to ti_4^m; l_4; \phi_4\ e_1^*\ \mathsf{end}$

  This case is the same as above, except with e_2 instead of e_1 and $k+1$ instead of 0.

- Case: $S; C \vdash \mathsf{label}_n\{e^*\}\ v^n\ \mathsf{end} : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$
  $\wedge\ \mathsf{label}_n\{e^*\}\ v^n\ \mathsf{end} \hookrightarrow v^n$

  We want to show that $v^n : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$

  We first figure out what $ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$ looks like.

  By Lemma Inversion on Rule Label, we know $ti_2^* = ti_1^*\ ti_4^n$.

  $S; C \vdash v^n : \epsilon; l_1; \phi_1 \to ti_4^n; l_2; \phi_2$ because it is a premise of Rule Label which we have assumed to hold.

  Now we can show that v^n has the same type.

  Therefore, $S; C \vdash v^n : ti_1^*; l_1; \phi_1 \to ti_1^*\ ti_4^n; l_1; \phi_2$ by Rule Stack-Poly.

- Case: $S; C \vdash \mathsf{label}_n\{e^*\}\ \mathsf{trap}\ \mathsf{end} : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$
  $\wedge\ \mathsf{label}_n\{e^*\}\ \mathsf{trap}\ \mathsf{end} \hookrightarrow \mathsf{trap}$

  Trivially, $C \vdash \mathsf{trap} : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$ by Rule Trap since trap accepts any precondition and postcondition.

- Case: $S; C \vdash \mathsf{label}_n\{e^*\}\ L^j[(t.\mathsf{const}\ c)^n\ (\mathsf{br}\ j)]\ \mathsf{end} : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$
  $\wedge\ \mathsf{label}_n\{e^*\}\ L^j[(t.\mathsf{const}\ c)^n\ (\mathsf{br}\ j)] \hookrightarrow (t.\mathsf{const}\ c)^n\ e^*$

  We want to show that $v^n\ e^* : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$.

  Intuitively, this proof works because the premise of Rule Br assumes that $C_{\text{label}}(i)$ is the precondition ($ti_1^n; l_3; \phi_5$, as we will soon see) of the stored instructions e^* in the $i+1$th label, and the postcondition of the label block is immediately reachable from the postcondition of e^*.

Meanwhile, that assumptions is ensured by Rule Label, which ensures that e^* has the same precondition as the $i+1$th branch postcondition on the label stack and the same postcondition as the label block instruction.

By Lemma Inversion on Rule Label, $ti_2^* = ti_1^*\ ti_4^*$ for some ti_4^*.

Also, $S; C, \text{label}(ti_1^n; l_3; \phi_5)^j \vdash (t.\text{const}\ c)^n\ (\text{br}\ j) : \epsilon; l_3; \phi_3 \to ti_\emptyset^*; l_\emptyset; \phi_\emptyset$ for some l_3 and ϕ_3, where $\phi_5 = \phi_3, (t\ a)^n, (=\ a\ (t\ c))^n$, by Lemma Inversion on Rule Label and Rule Br.

Then, $S; C, \text{label}(ti_1^n; l_3; \phi_5)^j \vdash (\text{br}\ j) : ti_1^n; l_3; \phi_5 \to ti_\emptyset^*; l_\emptyset; \phi_\emptyset$, by Lemma Inversion on Rule Composition and Rule Const.

Then, $S; C, \text{label}(ti_1^n; l_3; \phi_5)^j \vdash (t.\text{const}\ c)^n : \epsilon; l_3; \phi_3 \to ti_1^n; l_3; \phi_5$ since it is a premise of *composition* which we have assumed to hold.

Further, $S; C \vdash e^* : ti_1^n; l_3; \phi_5 \to ti_2^*; l_2; \phi_4$ since it is a premise of Rule Label which we have assumed to hold, and $\phi_4 \implies \phi_2$ by Lemma Inversion on Rule Label.

Then, $S; C \vdash (t.\text{const}\ c)^n\ e^* : \epsilon; l_1; \phi_1 \to ti_2^*; l_2; \phi_4$ by Lemma Lift-Consts and Rule Composition.

Finally, $C \vdash (t.\text{const}\ c)^n\ e^* : ti_1^*; l_1; \phi_1 \to ti_1^*\ ti_4^*; l_2; \phi_2$ by Rule Stack-Poly and Rule Subtyping.

- Case: $S; C \vdash (\text{i32.const}\ 0)\ (\text{br\_if}\ j) : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$
  $\wedge\ (\text{i32.const}\ 0)\ (\text{br\_if}\ j) \hookrightarrow \epsilon$

  In the case that br_if does not branch, it acts exactly like drop (consumes (i32.const 0) and reduces to the empty sequence). Thus, this case is the same as the drop case.

- Case: $S; C \vdash (\text{i32.const}\ k+1)\ (\text{br\_if}\ j) : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$
  $\wedge\ (\text{i32.const}\ k+1)\ (\text{br\_if}\ j) \hookrightarrow \text{br}\ j$

  We want to show that $S; C \vdash \text{br}\ j : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$

  We know $S; C \vdash \text{br\_table}\ j : ti_1^*\ (\text{i32}\ a); l_1; \phi_1, (t\ a), (=\ a\ (\text{i32}\ k)) \to ti_2^*; l_2; \phi_2$ by Lemma Inversion on Rule Composition and Rule Const.

Then we know $C_{\text{label}}(j) = (ti_3^*;\ l_1;\ \phi_3)$, where $ti_1^* = ti_0^*\ ti_3^*$, $ti_2^* = ti_0^*\ ti_3^*$, and $\phi_1, (t\ a), \neg(=\ a\ (\text{i32}\ 0)) \implies \phi_3$ by Lemma Inversion on Rule Br-Table.

Now we can show that $S;\ C \vdash \text{br}\ j : ti_0^*\ ti_3^*;\ l_1;\ \phi_1 \to ti_0^*\ ti_3^*;\ l_2;\ \phi_2$.

We have $S;\ C \vdash \text{br}\ j : ti_3^*;\ l_1;\ \phi_3 \to ti_3^*;\ l_2;\ \phi_2$ by Rule Br.

Then, $S;\ C \vdash \text{br}\ j : ti_3^*;\ l_1;\ \phi_1, (t\ a), \neg(=\ a\ (\text{i32}\ 0)) \to ti_3^*;\ l_2;\ \phi_2$, and therefore $S;\ C \vdash \text{br}\ j : ti_3^*;\ l_1;\ \phi_1, (t\ a), (=\ a\ (\text{i32}\ k)) \to ti_3^*;\ l_2;\ \phi_2$ by Rule Subtyping

Because a is fresh after reduction, $\phi_1 \implies \phi_1, (\text{i32}\ a), (=\ a\ (\text{i32}\ k))$.

Therefore, $C \vdash \text{br}\ j : ti_0^*\ ti_3^*;\ l_1;\ \phi_1 \to ti_0^*\ ti_3^*;\ l_2;\ \phi_2$ by Rule Stack-Poly and Rule Subtyping.

- Case: $S;\ C \vdash (\text{i32.const}\ k)\ (\text{br\_table}\ j_1^k\ j\ j_2^*) : ti_1^*;\ l_1;\ \phi_1 \to ti_2^*;\ l_2;\ \phi_2 \wedge (\text{i32.const}\ k)\ (\text{br\_table}\ j_1^k\ j\ j_2^*) \hookrightarrow \text{br}\ j$

  We want to show that $S;\ C \vdash \text{br}\ j : ti_1^*;\ l_1;\ \phi_1 \to ti_2^*;\ l_2;\ \phi_2$

  This case is similar in structure to the $(\text{i32.const}\ k+1)\ (\text{br\_if}\ j)$ case.

  We know

  $$S;\ C \vdash \text{br\_table}\ j_1^k\ j\ j_2^* : ti_1^*\ (\text{i32}\ a);\ l_1;\ \phi_1, (t\ a), (=\ a\ (\text{i32}\ k)) \\ rightarrowti_2^*;\ l_2;\ \phi_2$$

  by Lemma Inversion on Rule Composition and Rule Const.

  Then we know $C_{\text{label}}(j) = (ti_3^*;\ l_1;\ \phi_3)$, where $ti_1^* = ti_0^*\ ti_3^*$, $ti_2^* = ti_0^*\ ti_3^*$, and $\phi_1, (t\ a), \neg(=\ a\ (\text{i32}\ 0)) \implies \phi_3$ by Lemma Inversion on Rule Br-Table.

  Now we can show that $S;\ C \vdash \text{br}\ j : ti_0^*\ ti_3^*;\ l_1;\ \phi_1 \to ti_0^*\ ti_3^*;\ l_2;\ \phi_2$.

  We have $S;\ C \vdash \text{br}\ j : ti_3^*;\ l_1;\ \phi_3 \to ti_3^*;\ l_2;\ \phi_2$ by Rule Br.

  Then, $S;\ C \vdash \text{br}\ j : ti_3^*;\ l_1;\ \phi_1, (t\ a), (=\ a\ (\text{i32}\ k)) \to ti_3^*;\ l_2;\ \phi_2$ by Rule Subtyping.

  Because a is fresh after reduction, $\phi_1 \implies \phi_1, (\text{i32}\ a), (=\ a\ (\text{i32}\ k))$.

Therefore, $C \vdash \mathsf{br}\ j : ti_0^*\ ti_3^*; l_1; \phi_1 \to ti_0^*\ ti_3^*; l_2; \phi_2$ by Rule Stack-Poly and Rule Subtyping.

- Case: $C \vdash (\mathsf{i32.const}\ k + n)\ (\mathsf{br\_table}\ j_1^k\ j) : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$
  $\wedge\ (\mathsf{i32.const}\ k + n)\ (\mathsf{br\_table}\ j_1^k\ j) \hookrightarrow \mathsf{br}\ j$

  Same as above.

- Case: $S; C \vdash \mathsf{call}\ j : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$
  $\wedge\ s; \mathsf{call}\ j \hookrightarrow_i \mathsf{call}\ s_{\text{func}}(i, j)$

  We want to show that $\mathsf{call}\ s_{\text{func}}(i, j) : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$.

  By Lemma Inversion on Rule Call, we know that $l_2 = l_1$, $ti_1^* = ti_0^*\ ti_3^*$, $ti_2^* = ti_0^*\ ti_4^*$, $\phi_1 \implies \phi_3$, and $\phi_3, \phi_4 \implies \phi_2$, where $ti_3^*; l_3; \phi_3 \to ti_4^*; l_4; \phi_4 = C_{\text{func}}(j)$.

  We also know $S \vdash s_{\text{inst}}(i) : C$ since it is a premise of $\vdash s : S$ which we have assumed to hold.

  Then we know $S \vdash s_{\text{func}}(i, j) : ti_3^*; l_3; \phi_3 \to ti_4^*; l_4; \phi_4$ because it is a premise of $S \vdash s_{\text{inst}}(i) : C$.

  Therefore, $S; C \vdash \mathsf{call}\ s_{\text{func}}(i, j) : ti_3^*; l_3; \phi_3 \to ti_4^*; l_4; \phi_4$ by Rule Call-Cl.

  Thus, $S; C \vdash \mathsf{call}\ s_{\text{func}}(i, j) : ti_0^*\ ti_3^*; l_1; \phi_1 \to ti_0^*\ ti_4^*; l_2; \phi_2$ by Rule Stack-Poly and Rule Subtyping.

- Case: $S; C \vdash (\mathsf{i32.const}\ j)\ \mathsf{call\_indirect}\ ti_3^*; l_3; \phi_3 \to ti_4^*; l_4; \phi_4$
  $: ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$
  $\wedge\ s; (\mathsf{i32.const}\ j)\ \mathsf{call\_indirect}\ ti_3^*; l_3; \phi_3 \to ti_4^*; l_4; \phi_4 \hookrightarrow_i \mathsf{call}\ s_{\text{tab}}(i, j)$
  where $s_{\text{tab}}(i, j)_{\text{code}} = (\mathsf{func}\ tfi_0\ \mathsf{local}\ t^*\ e^*)$ and $tfi_0 <: ti_3^*; l_3; \phi_3 \to ti_4^*; l_4; \phi_4$

  We want to show that $\mathsf{call}\ s_{\text{tab}}(i, j) : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$.

  By Lemma Inversion on Rule Composition, Rule Const, and Rule Call-Indirect, we know that $ti_1^* = ti_0^*\ ti_3^*$ and $ti_2^* = ti_0^*\ ti_4^*$ for some ti_0^*, $l_1 = l)2$, $\phi_1 \implies \phi_3$, and $\phi_4 \implies \phi_2$.

We know $S \vdash s_{\text{tab}}(i,j) : tfi_0$ since it is a premise of $\vdash s : S$ which we have assumed to hold.

Then, $S; C \vdash \mathsf{call}\ s_{\text{tab}}(i,j) : tfi_0$ by Rule Call-Cl.

$S; C \vdash \mathsf{call}\ s_{\text{tab}}(i,j) : ti_3^*; l_1; \phi_3 \to ti_4^*; l_2; \phi_4$ by Rule Subtyping.

Therefore, $S; C \vdash \mathsf{call}\ s_{\text{tab}}(i,j) : ti_0^*\ ti_1^*; l_1; \phi_1 \to ti_0^*\ ti_1^*; l_2; \phi_2$ by Rule Stack-Poly.

- Case: $S; C \vdash (\mathsf{i32.const}\ j)\ \mathsf{call\_indirect}\ tfi : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$
  $\wedge\ s; (\mathsf{i32.const}\ j)\ \mathsf{call\_indirect}\ tfi \hookrightarrow_i \mathsf{trap}$.

  Trivially, $S; C \vdash \mathsf{trap} : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$ by Rule Trap.

- Case: $S; C \vdash v^n\ \mathsf{call}\ cl : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$
  $\wedge\ s; v^n\ \mathsf{call}\ cl \hookrightarrow_i \mathsf{local}_m\ \{i; v^n\ (t.\mathsf{const}\ 0)^k\}$
  $\mathsf{block}\ (\epsilon; ti_3^n\ (t\ a)^n; \phi_3, (t\ a)^k, (=\ a\ (t\ 0))^k \to ti_4^m; l_4; \phi_4)\ e^*\ \mathsf{end}$
  end

  where $cl_{\text{code}} = \mathsf{func}\ (ti_3^n; \epsilon; \phi_3 \to ti_4^m; l_4; \phi_4)\ \mathsf{local}\ t^k\ e^*$ and $cl_{\text{inst}} = i$

  Let

  $$\begin{aligned} tfi_0 &= ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2, \\ tfi_1 &= \epsilon; ti_3^n\ (t\ a)^n; \phi_3, (t\ a)^k, (=\ a\ (t\ 0))^k \to ti_4^m; l_4; \phi_4, \text{ and} \\ tfi_2 &= ti_3^n; \epsilon; \phi_3 \to ti_4^m; l_4; \phi_4 \end{aligned}$$

  We want to show that

  $$\begin{aligned} &S; C \vdash_i \mathsf{local}_m\ \{i; v^n\ (t.\mathsf{const}\ 0)^k\} \\ &\qquad \mathsf{block}\ \epsilon; ti_3^n\ (t_2\ a_2)^n; \phi_3 \to ti_4^m; l_4; \phi_4\ e^*\ \mathsf{end} \\ &\quad \mathsf{end} \\ &: ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2 \end{aligned}$$

  By Lemma Inversion on Rule Composition, Rule Const, and Rule

Call-Cl, we know $l_2 = l_1$, $ti_2^* = ti_1^*\ ti_4^m$, $\phi_1, (t_2\ a_2), (\mathsf{eq}\ a_2\ (t_2\ c)) \implies \phi_3$, $\phi_4 \implies \phi_2$, and $S \vdash cl : tfi_1$.

We also know that

$$S;\ C \vdash (t_2.\mathsf{const}\ c)^n : ti_1^*;\ l_1;\ \phi_1 \to ti_1^*\ ti_5^n;\ l_1;\ \phi_1, (t_2\ a_2), (=\ a_2\ (t_2\ c))$$

where $v^n = (t_2.\mathsf{const}\ c)^n$, and

$$S;\ C \vdash \mathsf{call}\ cl : ti_1^*\ ti_5^n;\ l_1;\ \phi_1, (t_2\ a_2)^n, (=\ a_2\ (t_2\ c))^n \to ti^*\ ti_2^m;\ l_2;\ \phi_2$$

because they are premises of Rule Composition which we have assumed to hold.

We have $C \vdash \mathsf{func}\ tfi_2\ \mathsf{local}\ t^k\ e^* : tfi_2$ because it is a premise of $S \vdash cl : tfi_2$.

Then,

$$S;\ C, \mathrm{local}\ t_2^n\ t^k, \mathrm{label}\ (ti_4^m;\ l_4;\ \phi_4), \mathrm{return}\ (ti_4^m;\ l_4;\ \phi_4) \vdash e^* : tfi_1$$

because it is a premise of the above derivation.

We can now reconstruct the type after reduction.

$S;\ C, \mathrm{local}\ t_2^n\ t^k, \mathrm{return}\ (ti_4^m;\ l_4;\ g_4;\ \phi_4) \vdash\ \mathsf{block}\ tfi_1\ e^*\ \mathsf{end} : tfi_1$ by Rule Block.

$\vdash v : (t_2\ a_2);\ \circ, (t_2\ a_2), (\mathsf{eq}\ a_2\ (t_2\ c)))^n$ by Rule Admin-Const, and $(\vdash (t\mathsf{const}\ 0) : (t\ a);\ \circ, (t\ a), (\mathsf{eq}\ a\ (t\ 0)))^k$ by Rule Admin-Const.

Then, $S;\ (ti_4^m;\ l_4;\ \phi_4)\ \vdash v^n\ (t\mathsf{const}\ 0)^k;\ \mathsf{block}\ tfi_2\ e^*\ \mathsf{end} : ti_4^m;\ l_4;\ \phi_4$ by Rule Code.

Recall that $\phi_4 \implies \phi_2$.

Then, $S;\ (ti_4^m;\ l_4;\ \phi_4)\ \vdash v^n\ (t\mathsf{const}\ 0)^k;\ \mathsf{block}\ tfi_2\ e^*\ \mathsf{end} : ti_4^m;\ l_4;\ \phi_2$ by Rule Subtyping.

$S;\ C \vdash \mathsf{local}_m\{j;\ v^n\ (t.\mathsf{const}\ 0)^k\}\ \mathsf{block}\ tfi_2\ e^*\ \mathsf{end}\ \mathsf{end} : \epsilon;\ l_1;\ \phi_1 \to \epsilon\ ti_4^m;\ l_1;\ \phi_1, \phi_2$ by Rule Local.

$S; C \vdash \mathsf{local}_m\{j;\, v^n\ (t.\mathsf{const}\ 0)^k\}\ \mathsf{block}\ tfi_2\ e^*\ \mathsf{end}\ \mathsf{end} : tfi_0$ by Rule Stack-Poly.

- Case: $S; C \vdash \mathsf{local}_n\{i;\, v_l^*\}\ \mathsf{trap}\ \mathsf{end} : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$
  $\wedge\ \mathsf{local}_n\{i;\, v_l^*\}\ \mathsf{trap}\ \mathsf{end} \hookrightarrow\ \mathsf{trap}$

  Trivially, $S; C \vdash \mathsf{trap} : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$ by Rule Trap.

- Case: $S; C \vdash \mathsf{local}_n\{i;\, v_l^*\}\ L^k[(t.\mathsf{const}\ c)^n\ \mathsf{return}]\ \mathsf{end} : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$
  $\wedge\ \mathsf{local}_n\{i;\, v_l^*\}\ L^k[(t.\mathsf{const}\ c)^n\ \mathsf{return}]\ \mathsf{end} \hookrightarrow_j\ (t.\mathsf{const}\ c)^n$

  We want to show that $(t.\mathsf{const}\ c)^n : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$

  This proof is similar to the br case above, but with a few extra steps.

  First, we derive the type of $(t.\mathsf{const}\ c)^n$ from the precondition of return.

  $ti_2^* = ti_1^*\ ti^n$, $l_1 = l_2$, $S; (ti_3^n; l_2; \phi_3) \vdash_i v_l^*; L^k[(t.\mathsf{const}\ c)^n\ \mathsf{return}] : ti_3^n; l_2; \phi_3$, and $\phi_1, \phi_3 \implies \phi_2$ by Lemma Inversion on Rule Local.

  $(\vdash v_l : ti_l; \phi_l)^*$ and $S; C_l \vdash L^k[(t.\mathsf{const}\ c)^n\ \mathsf{return}] : \epsilon; ti_l^*; \phi_l^* \to ti^n; l_3; \phi_3$, where $C_l = S_{\mathsf{inst}}(i), \mathsf{local}\ t^*, \mathsf{return}\ (ti^n; l_3; \phi_3)$, because they are premises of Rule Code that we have assumed to hold.

  $ti_l^* = (t_l\ a_l)^*$ because it is a premise of Rule Admin-Const which we have assumed to hold.

  By Lemma Inversion on Rule Composition and Rule Return, $S; C_l \vdash (t.\mathsf{const}\ c)^n : ti_4^*; l_4; \phi_4 \to ti_3^n; l_3; \phi_3$, and $S; C_l \vdash \mathsf{return} : ti_3^n; l_3; \phi_3 \to ti_\emptyset^*; l_\emptyset; \phi_\emptyset$.

  By Lemma Inversion on Rule Const, $l_4 = l_3$ and $\phi_4, (t\ a)^n, (= a\ (t\ c))^n \implies \phi_3$.

  Now we can show that $(t.\mathsf{const}\ c)^n : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$

  We have $S; C \vdash (t.\mathsf{const}\ c)^n : \epsilon; l_3; \phi_4 \to ti_3^n; l_3; \phi_4, (t\ a)^n, (= a\ (t\ c))^n$ by Rule Const.

  Then, $S; C \vdash (t.\mathsf{const}\ c)^n : \epsilon; ti_l^*; \phi_l^* \to ti_3^n; l_3; \phi_4, (t\ a)^n, (= a\ (t\ c))^n$ by Lemma Lift-Consts.

By Lemma Inversion on Rule Const, $\phi_l^* \implies \phi_4$. Further, since a_l^* are fresh, $\circ \implies \phi_l^*$.

Thus, $S; C \vdash (t.\mathsf{const}\ c)^n \epsilon; ti_l^*; \circ \to ti_3^n; l_3; \phi_4, (t\ a)^n, (= a\ (t\ c))^n$ by Rule Subtyping.

We know then that $(t\ a)^n, (= a\ (t\ c))^n \implies \phi_4, (t\ a)^n, (= a\ (t\ c))^n$ by Lemma Inversion on Rule Const, and therefore that ϕ_4 only contains constraints on fresh variables.

Then, $S; C \vdash (t.\mathsf{const}\ c)^n : \epsilon; l_1; \phi_1 \to ti_3^n; l_1; \phi_1, (t\ a)^n, (= a\ (t\ c))^n$ by Rule Const.

Then, $S; C \vdash (t.\mathsf{const}\ c)^n : \epsilon; l_1; \phi_1 \to ti_3^n; l_2; \phi_1, \phi_4, (t\ a)^n, (= a\ (t\ c))^n$, $S; C \vdash (t.\mathsf{const}\ c)^n : \epsilon; l_1; \phi_1 \to ti^n; l_2; \phi_1, \phi_3$, and finally $S; C \vdash (t.\mathsf{const}\ c)^n : \epsilon; l_1; \phi_1 \to ti^n; l_2; \phi_2$ by Rule Subtyping.

Therefore, $S; C \vdash (t.\mathsf{const}\ c)^n : ti_1^*; l_1; \phi_l^* \to ti_2^*; l_2; \phi_2$ by Rule Stack-Poly.

- Case: $S; C \vdash v\ (\mathsf{tee\_local}\ j) : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$
  $\wedge\ v\ (\mathsf{tee\_local}\ j) \hookrightarrow v\ v\ (\mathsf{set\_local}\ j)$

  Note: We can include tee_local here because it does not actually need to reason about locals since it gets reduced to a set_local , so we only have to do the reasoning in the set_local case.

  We want to show that $v\ v\ (\mathsf{set\_local}\ j) : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$.

  As usual, we start by figuring out what $ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$ looks like.

  By Lemma Inversion on Rule Composition, we know that $S; C \vdash v : ti_1^*; l_1; \phi_1 \to ti_3^*; l_3; \phi_3$, and $S; C \vdash \mathsf{tee\_local}\ j : ti_3^*; l_3; \phi_3 \to ti_2^*; l_2; \phi_2$.

  By Lemma Inversion on Rule Tee-Local, we also know that $ti_3^* = ti^*\ (t\ a)$, $ti_2^* = ti^*\ (t\ a_2)$, $l_2 = l_3[j := (t\ a)]$, and $\phi_3, (t\ a_2), (= a_2\ a) \implies \phi_2$.

  Then, by Lemma Inversion on Rule Const, $t.\mathsf{const}\ c = v$, $ti_1^* = ti^*$, $l_3 = l_1$, and $\phi_1, (t\ a), (= a\ (t\ c)) \implies \phi_3$.

Now, we can show that $v\ v\ (\mathsf{set\_local}\ j) : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$.

We have $S; C \vdash v\ v : \epsilon; l_1; \phi_1 \to (t\ a_2)\ (t\ a); l_1; \phi_1, (t\ a_2), (= a_2\ (t\ c)), (t\ a), (= a\ (t\ c))$ by Rule Const.

We also have $(t\ a_2), (= a_2\ (t\ c)), (t\ a), (= a\ (t\ c)) \implies (t\ a_2), (= a_2\ a), (t\ a), (= a\ (t\ c))$ by $\implies$.

Then, $S; C \vdash v\ v : \epsilon; l_1; \phi_1 \to (t\ a_2)\ (t\ a); l_1; \phi_2$ by Rule Subtyping.

We have $S; C \vdash \mathsf{set\_local}\ j : (t\ a); l_1; \phi_2 \to \epsilon; l_1[j := (t\ a)]; \phi_2$ by Rule Set-Local.

Therefore, $S; C \vdash v\ v\ (\mathsf{set\_local}\ j) : ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$ by Rule Composition and Rule Stack-Poly.

□