Indexed Types for a Statically Safe WebAssembly

ADAM T. GELLER, University of British Columbia, Canada
JUSTIN P. FRANK, University of British Columbia, Canada and University of Maryland, USA
WILLIAM J. BOWMAN, University of British Columbia, Canada

We present Wasm-precheck, a superset of WebAssembly (Wasm) that uses indexed types to express and check simple constraints over program values. This additional static reasoning enables safely removing dynamic safety checks required by Wasm, such as memory bounds checks. We implement Wasm-precheck as an extension of the Wasmtime compiler and runtime, evaluate the run-time and compile-time performance of Wasm-precheck vs Wasm, and find an average run-time performance gain of 1.72x faster in the widely used PolyBenchC benchmark suite, for a small overhead in binary size (7.18% larger) and type-checking time (1.4% slower). We also prove type and memory safety of Wasm-precheck, prove Wasm safely embeds into Wasm-precheck ensuring backwards compatibility, prove Wasm-precheck type-erases to Wasm, and discuss design and implementation trade-offs.

1 INTRODUCTION

WebAssembly (Wasm) is a low-level language designed to work well in the browser environment [Haas et al. 2017]. It has a small binary footprint and supports streaming execution (*i.e.,* execution can safely begin before the entire program has been downloaded). Wasm is also designed to be fast, outperforming JavaScript, and can be used to speed-up intensive computations within webpages. It is also safe—Wasm enforces a separation of code and data, uses a simple static types system, and is proven type and memory safe.

Although Wasm is type and memory safe, it relies on potentially costly dynamic checks for these safety guarantees in some important instructions. Errors raised by these checks are always fatal for Wasm, ensuring safety, although possibly complicating software development. They can be costly in terms of run-time performance, too. Jangda et al. [2019] measured that Wasm runs between 1.45–1.55x slower than corresponding native code. Their root cause analysis attributes part of this to the dynamic checks required by Wasm runtimes, particularly dynamic checks on indirect function calls. Our analysis, discussed in Section 6, finds dynamic memory bounds checks cause an average of 1.77x slowdown in some scenarios.

To mitigate the costs of dynamic checks on memory operations, the runtime for a Wasm module can reserve sufficient virtual memory to represent an entire 32-bit address space (4GiB[1]), and mark addresses outside the memory bounds as inaccessible. In many environments, such as in the browser, this works well, since browsers often use a lot of memory and run on end-user machines with a 64-bit address space and therefore plenty of virtual memory. In practice, this makes memory bounds checks essentially free[2].

[1]In practice, implementations reserve 8GiB to mitigate compiler bugs.
[2]Except for the omnipresent cost of virtual memory [Zagieboylo et al. 2020].

Authors' addresses: Adam T. Geller, University of British Columbia, Canada, atgeller@cs.ubc.ca; Justin P. Frank, University of British Columbia, Canada and University of Maryland, USA, justinpfrank@protonmail.com; William J. Bowman, University of British Columbia, Canada, wjb@williamjbowman.com.

XXXX-XXXX/2023/9-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

While this approach may work in the browser, it relies on the following assumptions:

- Wasm modules can address *only* a 32-bit address space.
- Wasm modules are running on a 64-bit architecture and operating system.
- 4GiB of virtual memory is available.
- The system provides efficient virtual memory with permissions.

However, these assumptions do not hold in some contexts, and may not continue to hold in general. Wasm has become popular as an intermediate language and efficient virtual machine/sandbox for many purposes, and has continued to grow beyond its original design. The Memory64 proposal[3] extends Wasm to include 64-bit addressable memory. Some embedded systems only provide 32-bit (or smaller) address spaces[4]. Wasm is being experimented with as a replacement for high-overhead containers in serverless applications[5]; in this context, limiting virtual memory is useful to provide a hard resource limit to a Wasm module. Wasm is used as an intermediate language for optimizing GPGPU computations [Ginzburg et al. 2023], and GPUs do not provide the necessary virtual memory abstractions for efficient bounds checks.

We claim that Wasm can be redesigned with a stronger type system to mitigate the performance overhead of dynamic safety checks without the above assumptions of the runtime environment. To test our hypothesis, we design, implement, and evaluate Wasm-precheck, an extension of Wasm with an *indexed type system* that can statically check the safety preconditions for each Wasm instruction that requires dynamic checks. Indexed types equip a type system with the ability to statically enforce constraints on run-time values, refining a type to a subset of values of that type [Zenger 1997]. This ability is key to static reasoning about the low-level patterns in Wasm.

Using Wasm-precheck's stronger type system, we can safely remove dynamic checks both in theory and in practice. We prove type safety of Wasm-precheck Section 4.3, which implies well-typed programs without (some or all) dynamic checks never get "stuck" (or access uninitialized memory). We implement Wasm-precheck in an extension of the Wasmtime compiler [Bytecode Alliance 2019] and evaluate run-time and compile-time performance on the PolyBenchC benchmark suite [Pouchet and Yuki 2016]. The type system enables safely removing most dynamic checks, in practice by moving checks out of loops. This yields an average performance speed-up of 1.72x (with a best-case speed-up of 3x) for a small overhead in binary size (7.18% larger) and time taken to type-check the program (1.4% slower) Section 6.

We pay attention to design decisions that would affect adoption and implementation of Wasm-precheck. To ensure backwards compatibility, we show that Wasm programs can be automatically embedded into Wasm-precheck, and that all Wasm-precheck programs erase to well-typed Wasm programs (possibly with more dynamic checks). Wasm-precheck does not fix a particular constraint solving algorithm, although we provide a prototype implementation; developers may choose their own trade-offs between the compile-time cost of a more expressive type system vs. the benefits of additional static reasoning. We elaborate on this in Section 2 and Section 3.3.2. We also discuss how Wasm-precheck could be interpreted as a specification of a sound static analysis of Wasm, in case the addition of annotations to the surface language is undesirable or infeasible (Section 8).

In short, our contributions are:

- The formal model of Wasm-precheck, an extension of Wasm that, provided sufficient type annotations, requires no dynamic checks for type and memory safety (Section 3).

[3]https://github.com/WebAssembly/memory64

[4]WAMR supports 32-bit architectures used in the embedded and IoT space: https://github.com/bytecodealliance/wasm-micro-runtime.

[5]This was the explicit goal of Fastly's Lucet project, now replaced by Wasmtime: https://www.fastly.com/blog/announcing-lucet-fastly-native-webassembly-compiler-runtime.

- An implementation of Wasm-precheck in an extension of the Wasmtime compiler, and a reference implementation of the Wasm-precheck formal model in Redex (Section 5).
- A performance analysis Wasm vs Wasm-precheck (Section 6).
- A proof of type safety for Wasm-precheck (Section 4.3).
- A proof of backwards compatibility—all well-typed Wasm programs automatically embed into Wasm-precheck, possibly with more dynamic checks than necessary (Appendix B.1).
- A proof that Wasm-precheck introduces no new dynamic behaviours—all well-typed Wasm-precheck programs erase to well-typed Wasm, potentially with extra dynamic checks (Section 4.2).

2 MAIN IDEAS

Wasm is unusual for low-level performance-oriented languages in that it provides a strong type safety guarantee. Wasm programs are guaranteed to be type and memory safe—if a well-typed program terminates, it either runs to a value of the expected type, or raises a well-defined dynamic error [Haas et al. 2017]. Importantly, this rules out undefined behaviour such as accessing out-of-bounds memory, or casting integers to labels.

Unfortunately, not all undefined behaviour can be caught statically by Wasm's type system. Consider the reduction rule for a binary operation.

$$(t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ t.\mathit{binop} \hookrightarrow t.\mathsf{const}\ c \quad \text{where } c = \mathit{binop}(c_1, c_2)$$

This small-step relation $e^* \hookrightarrow e^*$ reduces instructions in a stack machine. A sequence e^* (0 or more instructions e) represents the stack of values and instructions to be executed. The instruction $t.\mathsf{const}\ c_2$ indicates a value c_2 of type t on the stack. The $t.\mathit{binop}$ instruction expects 2 operands of type t on the top of the stack, and reduces them to the value c produced by the binary operation. Since all programs are well typed, this operation must succeed and produce a constant of type t, so our semantics need not perform any dynamic checks on the values c_1 and c_2.

Except when binop is division, in which case there is a well-typed value for which $\mathit{binop}(c_1, c_2)$ is undefined, namely, when $c_2 = 0$. 0 is a valid value of type $\mathsf{i32}$, so the type system allows an undefined operation. We require a second reduction rule for binop to make division well defined.

$$(t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ t.\mathit{binop} \hookrightarrow \mathsf{trap} \quad \text{where } \mathit{binop} = \mathsf{div} \text{ and } c_2 = 0$$

This is unfortunate; now every division operation must perform a dynamic check, possibly raising an error in production we could have caught in development, and costing run-time performance (although, this cost is irrelevant for division).

Idea: The safe dynamic semantics are a specification for a static reasoning system.

While Wasm has a strong static type system, it only provides coarse reasoning about types such as $\mathsf{i32}$, but cannot express the fine-grained precondition for the well-definedness of division. The type judgement has the shape $C \vdash e^* : t_1^* \rightarrow t_2^*$, which checks that the instruction sequence e^* has the given instruction type $t_1^* \rightarrow t_2^*$ under the environment C. The instruction type $t_1^* \rightarrow t_2^*$ expresses a precondition that values of type t_1^* are on the stack prior to executing the instruction (sequence), and a postcondition that values of type t_2^* are on the stack after execution.

For example, this is the Wasm typing rule for binary operations:

$$\frac{}{C \vdash t.\mathit{binop} : t\ t \rightarrow t}$$

A binary operation, such as division, expects two values of type t (either 32-bit integer or float) on the stack; after executing the operation, there should be a single value of type t on the stack. This is insufficient to reason about whether $c_2 = 0$.

We use division as a running example, but Wasm features several instructions with the same problem. These include memory accesses, which require dynamic memory bounds checks, and indirect function calls, which require dynamic type checks.

Wasm has a strong type system, but it is simple (in the sense of simply-typed λ-calculus). It is capable only of expressing invariants such as "a binary operation takes two integers", but not all safety conditions required by the run-time system. In particular, it is insufficient to express the true type of division: a binary operation on two integers such that the divisor is non-zero. However, there are some kinds type systems that *are* capable of expressing such invariants.

Idea: An indexed type system suffices to express the conditions under which dynamic checks can be removed from each Wasm instruction.

An indexed type system essentially changes the language of types from simple types to a (typically decidable) predicate logic with constraints between (representations of) values. Types are indexed by a name representing the run-time value for the term of that type. The type system collects and solves constraints between these values. For example, the following is (a simplification of) the typing rule for statically safe division.

$$\frac{\phi \Rightarrow \neg(=\alpha_2\ (t\ 0))}{C \vdash t.\mathsf{div}\checkmark\ :(t\ \alpha_1)\ (t\ \alpha_2);\ \phi \rightarrow (t\ \alpha_3);\ \phi, (=\alpha_3\ (\mathsf{div}\ \alpha_1\ \alpha_2))}$$

A type $(t\ \alpha)$ represents a value α of type t on the stack. The name α is initially unconstrained, representing an unknown value. We modify the type system to collect a system of constraints ϕ between these names. In this typing rule, the safe division operation is well typed if the constraint set ϕ guarantees that the second operand, named α_2 in the type system, cannot be 0. This is easily decided by a solver for our logic. In the postcondition, we add a new constraint that α_3 is equal to α_1 divided by α_2. This does not perform the division at type checking time, since α_1 and α_2 may not have known concrete values, but adds this constraint to the constraint set.

This typing rule prechecks all the safety criteria for the division instruction statically; once the type system is satisfied, this instruction can never be the source of a trap, and requires no dynamic checks. Formally, we see this by needing only the one reduction rule, instead of the two for div.

$$(t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ t.\mathsf{div}\checkmark\ \hookrightarrow t.\mathsf{const}\ c \quad \text{where } c = div(c_1, c_2)$$

We define similar rules for memory accesses without dynamic bounds checks and indirect function calls without dynamic type checks. We prove type safety for Wasm-precheck, guaranteeing that the new typing rules are sufficient to imply the well-definedness of these reduction rules.

Statically proving that a dynamic value is non-zero may be difficult in general, so we keep the original div instruction with its dynamic check and under-specified typing rule. While we could replace the original instruction with div✓ in all cases, and require that the check is inserted explicitly when necessary, it is useful for Wasm-precheck to remain a strict superset of Wasm.

Idea: Wasm-precheck need not be *implemented* as a new language with a separate syntax, but could be read as a specification for a sound static analysis over Wasm.

While we formalize Wasm-precheck as a language, one could also view it as a specification for a proven-correct static analysis. Wasm-precheck is fully backwards compatible with Wasm, meaning that any well-typed Wasm program is also well typed in Wasm-precheck (although old code may not automatically inherit improved static reasoning). Thus, programmers are not required to fight with the new type system. We prove this formally: all Wasm programs embed trivially into the new type system and run to the same result (Appendix B.1). However, one could implement a static analysis over Wasm, which provides Wasm-precheck annotations without developer intervention and without modifying the surface syntax of Wasm. Using Wasm-precheck in this way would be a conservative approximation of the type system. If a tool can infer these annotations, or the

$$testop ::= \mathsf{eqz} \qquad binop ::= \mathsf{add} \mid \mathsf{sub} \mid \mathsf{shl} \mid \mathsf{or} \mid \dots \qquad relop ::= \mathsf{eq} \mid \mathsf{ne} \mid \mathsf{gt} \mid \mathsf{ge} \mid \dots$$

$$\begin{aligned} e ::= {} & \mathsf{unreachable} \mid \mathsf{nop} \mid \mathsf{drop} \mid \mathsf{select} \mid \mathsf{block}\ t_1^* \to t_2^*\ e^*\ \mathsf{end} \mid \mathsf{loop}\ t_1^* \to t_2^*\ e^*\ \mathsf{end} \\ & \mid \mathsf{if}\ t_1^* \to t_2^*\ e^*\ \mathsf{else}\ e^*\ \mathsf{end} \mid \mathsf{br}\ i \mid \mathsf{br\_if}\ i \mid \mathsf{br\_table}\ i^+ \mid \mathsf{return} \mid \mathsf{call}\ i \mid \mathsf{call\_indirect}\ ti_1^*; \phi_1 \to ti_2^*; \phi_2 \\ & \mid \mathsf{get\_local}\ i \mid \mathsf{set\_local}\ i \mid \mathsf{tee\_local}\ i \mid \mathsf{get\_global}\ i \mid \mathsf{set\_global}\ i \mid \mathsf{current\_memory} \mid \mathsf{grow\_memory} \\ & \mid t.\mathsf{const}\ c \mid t.testop \mid t.relop \mid t.binop \mid t.\mathsf{load}\ (tp\_sx)^?\ o \mid t.\mathsf{store}\ tp^?\ o \\ & \mid t.\mathsf{div}✓ \mid t.\mathsf{call\_indirect}✓\ ti_1^*; \phi_1 \to ti_2^*; \phi_2 \mid t.\mathsf{load}✓\ (tp\_sx)^?\ o \mid t.\mathsf{store}✓\ tp^?\ o \end{aligned}$$

Fig. 1. Wasm-precheck instruction syntax

programmer is willing to add annotations or rewrite code to help the type system, then Wasm-precheck can remove some dynamic checks while type and memory safety are still guaranteed for all programs. We discuss this further in Section 8.

Idea: Wasm-precheck improves performance in practice, as well as in theory.

We implement Wasm-precheck and show it can improve performance by reducing the number of dynamic checks required while maintaining safety. Compared to Wasm, we can use Wasm-precheck to remove 97% of the performance overhead of the dynamic checks on average, resulting in an average performance speed-up of 1.72x. The performance evaluation uses PolyBenchC, memory-intensive benchmarks, which are manually annotated with sufficient type information to check, as well as a few explicit dynamic checks when insufficient information is available statically. In effect, the type system enables moving dynamic checks out of the loop, replacing them with a single dynamic check before the loop. The programs used in the evaluation were the output of a the Emscripten compiler from C to Wasm, showing that Wasm-precheck can support patterns in compiled output, which is important since Wasm is generally used as a compiler target.

3 Wasm-precheck

3.1 Syntax

Wasm-precheck is a superset of Wasm with a different representation of types and four statically safe versions of Wasm instructions added. Figure 1 shows the syntax of Wasm-precheck, with changes compared to Wasm highlighted. Like Wasm, Wasm-prechk is a stack-based language. Dynamic operands to instructions are passed on the stack and are not part of the instruction syntax. Since Wasm-precheck uses an indexed type system, some type annotations are enriched compared to Wasm type annotations; we discuss these differences later in Section 3.3.1.

The key changes to the syntax are four new instructions, referred to as *prechecked* instructions and denoted with a ✓ at the end of the operator. Prechecked instructions are equivalent to their Wasm counterparts, but they don't require dynamic checks. We discuss how their safety is statically checked later in Section 3.3. But first, we present the dynamic semantics of Wasm-precheck.

3.2 Dynamic Semantics

Wasm-precheck's reduction relation has the same structure as Wasm. We briefly explain the dynamic semantics of all Wasm-precheck instructions; however, since most instructions are unchanged from Wasm, we only present the formal rules of new instructions and some helpful for understanding the indexed type system. For full definitions of Wasm instructions, see Figure 2 of Haas et al. [2017].

The reduction relation, $s; v^*; e^* \hookrightarrow_i s'; v'^*; e'^*$ is defined on *configurations* consisting of a run-time store s, which holds module instance information (the i decorating the reduction arrow indicates which module instance is being reduced); a sequence of values v^* representing local variables; and the instruction stack e^*. We ignore the module instance information, which is not critical for our work. A value v is represented by the constant instruction ($t.\mathsf{const}\ c$). As in Wasm,

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

$$\begin{array}{rll}
(t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ t.binop & \hookrightarrow t.\mathsf{const}\ c & \text{if } c = binop(c_1, c_2) \\
(t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ t.binop & \hookrightarrow \mathsf{trap} & \text{otherwise} \\
s;\ (\mathsf{i32.const}\ j)\ \mathsf{call\_indirect}\ (ti_1^*;\ \phi_1 \to ti_2^*;\ \phi_2) & \hookrightarrow_i \mathsf{call}\ s_{\text{tab}}(i,j) & \\
 & \multicolumn{2}{l}{\text{if } s_{\text{tab}}(i,j)_{\text{code}} = (\mathsf{func}\ (ti_1^*;\ \phi_1 \to ti_2^*;\ \phi_2)\ \ldots)} \\
s;\ (\mathsf{i32.const}\ j)\ \mathsf{call\_indirect}\ (ti_1^*;\ \phi_1 \to ti_2^*;\ \phi_2) & \hookrightarrow_i \mathsf{trap} & \text{otherwise} \\
s;\ (\mathsf{i32.const}\ k)\ (t.\mathsf{load}\ o) & \hookrightarrow_i t.\mathsf{const}\ const_t(b^*) & \text{if } s_{\text{mem}}(i, k+o, |t|) = b^* \\
s;\ (\mathsf{i32.const}\ k)\ (t.\mathsf{load}\ o) & \hookrightarrow_i \mathsf{trap} & \text{otherwise} \\
s;\ (\mathsf{i32.const}\ k)\ (t.\mathsf{const}\ c)\ (t.\mathsf{store}\ o) & \hookrightarrow_i s';\ \epsilon & \text{if } s' = s \text{ with mem}(i, k+o, |t|) = bits_t(c) \\
s;\ (\mathsf{i32.const}\ k)\ (t.\mathsf{const}\ c)\ (t.\mathsf{store}\ o) & \hookrightarrow_i \mathsf{trap} & \text{otherwise}
\end{array}$$

New rules

$$\begin{array}{rll}
(t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ t.\mathsf{div}\checkmark & \hookrightarrow t.\mathsf{const}\ c & \text{where } c_2 \neq 0 \text{ and } c = c_1/c_2 \\
s;\ (t.\mathsf{const}\ j)\ t.\mathsf{call\_indirect}\checkmark\ (ti_1^*;\ \phi_1 \to ti_2^*;\ \phi_2) & \hookrightarrow_i \mathsf{call}\ s_{tab}(i,j) & \\
 & \multicolumn{2}{l}{\text{where } s_{tab}(i,j) = \mathsf{func}\ (ti_1^*;\ \phi_1 \to ti_2^*;\ \phi_2)\ \mathsf{local}\ t^*\ e^*} \\
s;\ (\mathsf{i32.const}\ k)\ (t.\mathsf{load}\checkmark\ a\ o) & \hookrightarrow_i t.\mathsf{const}\ const_t(b^*) & \text{where } s_{mem}(i, k+o, |t|) = b^* \\
s;\ (\mathsf{i32.const}\ k)\ (t.\mathsf{const}\ c)\ (t.\mathsf{store}\checkmark\ a\ o) & \hookrightarrow_i s';\ \epsilon & \text{where } s' = s \text{ with mem}(i, k+o, |t|) = bits_t^{|t|}(c)
\end{array}$$

Fig. 2. Wasm reduction rules (with dynamic checks) and their Wasm-precheck counterparts

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

$$\begin{array}{rcl}
\mathsf{nop} & \hookrightarrow & \epsilon \\
\mathsf{unreachable} & \hookrightarrow & \mathsf{trap} \\
v^n\ \mathsf{block}\ t_1^n \to t_2^m\ e^*\ \mathsf{end} & \hookrightarrow & \mathsf{label}_m\{\epsilon\}\ v^n\ e^*\ \mathsf{end} \\
v^n\ \mathsf{loop}\ t_1^n \to t_2^m\ e^*\ \mathsf{end} & \hookrightarrow & \mathsf{label}_n\{\mathsf{loop}\ t_1^n \to t_2^m e^*\ \mathsf{end}\}\ v^n\ e^*\ \mathsf{end} \\
\mathsf{i32.const}\ 0\ \mathsf{if}\ t_1^n \to t_2^m\ e_1^*\ \mathsf{else}\ e_2^*\ \mathsf{end} & \hookrightarrow & \mathsf{block}\ t_1^n \to t_2^m\ e_2^*\ \mathsf{end} \\
\mathsf{i32.const}\ k+1\ \mathsf{if}\ t_1^n \to t_2^m\ e_1^*\ \mathsf{else}\ e_2^*\ \mathsf{end} & \hookrightarrow & \mathsf{block}\ t_1^n \to t_2^m\ e_1^*\ \mathsf{end} \\
\mathsf{label}_n\{e^*\}\ v^*\ \mathsf{end} & \hookrightarrow & v^* \\
\mathsf{label}_n\{e^*\}\ \mathsf{trap}\ \mathsf{end} & \hookrightarrow & \mathsf{trap} \\
\mathsf{label}_n\{e^*\}\ L^j[v^n\ \mathsf{br}\ j]\ \mathsf{end} & \hookrightarrow & v^n\ e^* \\
v^n\ (\mathsf{call}\ cl) & \hookrightarrow_i & \mathsf{local}_m\{cl_{\text{inst}};\ v^n\ (t.\mathsf{const}\ 0)^k\}\ \mathsf{block}\ (\epsilon \to t_2^m)\ e^*\ \mathsf{end}\ \mathsf{end} \\
 & & \text{where } cl_{\text{func}} = (\mathsf{func}\ t_1^n;\ \phi_1 \to t_2^m;\ \phi_2\ \mathsf{local}\ t^*\ e^*)
\end{array}$$

Fig. 3. Wasm-precheck reduction rules (excerpts)

the stack is represented as a sequence of values at the head of the instruction sequence e^*. Following Wasm, s, v, and the instance subscript i are elided when they are unchanged and unused.

Prechecked instructions require no dynamic checks, since their safety preconditions are enforced statically by the Wasm-precheck type system. This can be seen in the reduction rules for the prechecked instructions in Figure 2: unlike their non-prechecked counterparts, prechecked instructions do not have rules to trap (a trap is the Wasm run-time error).

Figure 3 shows instructive excerpts unchanged from Wasm. The simplest are nop, which removes itself from the stack, and unreachable, which unconditionally evaluates to trap. When trap appears as an operand or operator, all evaluation rules produce trap; trap is a fatal error.

Most instructions manipulate values on the stack. The constant instruction t.const c intuitively pushes a value onto the stack, but formally it *is* a value on the stack. Numeric operators, *binop*, *testop*, *unop*, and *relop*, consume either one or two values from the stack, and push one value as the result. We present the *binop* instructions at the top of Figure 2. The division operator div traps and the second argument is 0, whereas in the div✓ instruction, the second operand c_2 is statically guaranteed to be non-zero. drop consumes a value from the top of the stack and does nothing with it. Finally, select is a ternary operator that consumes three values and pushes either the first or second value based on the truthiness of the third value—0 is false, and other values are truthy.

There are three control flow blocks that introduce a label—block, loop, and if. Their reduction rules, given in Figure 3, are unchanged from Wasm, but we explain them to clarify how labels are introduced, as indexed typing for labels is tricky. Each block instruction introduces a new evaluation context, which binds a label as a de Bruijn index to a sequence of instructions. Instructions e^* in the body of the block are reduced in the this evaluation context. Intuitively, labels point to where evaluation should continue when jumped to. The loop block binds the label to the loop itself, while block (and therefore if) bind the label to an empty instruction sequence. Jumping to a loop's label repeats the loop; control exits the loop by default. Jumping to a block's label exits the block early.

Branching (br j) takes some values v^n, jumps to the jth (zero-indexed) label in the evaluation context of the instruction, and continues executing with v^n on the stack but the labels discarded. Execution continues with the code bound to the label of the jth outer block, inside the remaining evaluation context, as seen in the last rule of Figure 3. We elide formal rules for other branching instructions, but explain them briefly. The conditional branch, br_if, consumes a value from the stack and only branches if the value is truthy. Finally, br_table is essentially a br j where j is determined by indexing into a statically provided table (no relation to the function table) of branching indices i^+ based on the instruction's dynamic operand (br_table resembles a `switch` statement). Returning (return) is similar to branching, but jumps to a separate class of label introduced by a function call.

Direct function calls call i introduce a return label, consume the function arguments from the stack and begin executing the function body, binding the arguments as local variables in the store. A function call shadows the entire store, replacing all values with the arguments. This is represented by a new execution context, local, similar to the one introduced by control flow blocks. The new execution context also includes a module pointer, cl_{inst}, to the module the function belongs to, dictacting which global variables, table, and memory can be accessed by the function body. When applying the reduction rule to instructions e^* inside a local execution context, the i in $s; v^*; e^* \hookrightarrow_i s'; v'^*; e'^*$ comes from the module pointer in the local. Functions need not explicitly return; control continues at the instruction after the call in either case.

The instructions for local and global variables are similar to each other, except for scope: local variables are local to functions, whereas global variables are global to all functions in a module instance. Both have instructions to push the value of the ith variable onto the stack (get_local i and get_global i). Although there are mutation instructions for both kinds of variables, (set_local i and set_global i), not all global variables are mutable, whereas all local variables are. The tee_local is a combined set_local and get_local that consumes and returns a value while also setting the ith local variable to that value, like the Unix tool `tee`; this instruction only exists for local variables.

An indirect function call, call_indirect, consumes a value from the stack, and attempts to call the function at that index in the *table*—a list of functions, defined statically as part of the module. Since the target of call_indirect is not necessarily statically known, indirect calls use a run-time check against the statically provided expected type $(ti_1^*; \phi_1 \rightarrow ti_2^*; \phi_2)$. call_indirect✓ relies on the fact that the function from the table has the expected type $(ti_1^*; \phi_1 \rightarrow ti_2^*; \phi_2)$ (see Figure 2).

Memory in Wasm-precheck is a linear sequence of bytes. There are the standard instructions for loading and storing values, load and store, respectively. The prechecked memory operations load✓ and store✓ rely on static bounds checks (see Figure 2). Memory operations also include static operands: the representation of the value being loaded or stored, tp_sx or tp, respectively; the offset, o; and alignment, a. The current_memory returns the current memory size, while grow_memory can increase size, returning either the new size of memory, or -1 if memory cannot be increased.

3.3 Type System

3.3.1 Index Language. In Wasm, instruction types $t^* \rightarrow t^*$ express the number and types of values expected on the stack before and after an instruction. In Wasm-precheck, the instruction type has

$$\begin{array}{ll} ti ::= (t\ \alpha) & \phi ::= \emptyset \mid \phi, P \\ l ::= ti^* & tfi ::= ti^*; \phi \rightarrow ti^*; \phi \end{array}$$

$$C ::= \{\text{func } tfi^*,\ \text{global } (\text{mut}^?\ t)^*,\ \text{table } (n, tfi^*)^?,\ \text{memory } m^?, \text{local } t^*, \text{label}(ti^*; l; \phi)^*,\ \text{return } (ti^*; \phi)^?\}$$

Fig. 4. Wasm-precheck Typing Syntax

$$\begin{array}{l} t ::= \text{i32} \mid \text{i64} \\ \alpha ::= IndexVariable \\ r ::= \alpha \mid (t\ c) \mid (unop\ r) \mid (binop\ r\ r) \mid (testop\ r) \mid (relop\ r\ r) \\ P ::= (=\ r\ r) \mid (\text{if } P\ P\ P) \mid \neg P \mid P \wedge P \mid P \vee P \end{array}$$

Fig. 5. Wasm-precheck Index Language Syntax

the form $ti^*; l; \Gamma; \phi \rightarrow ti^*; l; \Gamma; \phi$, whose non-terminals are defined in Figure 4. The *stack type* is a sequence of indexed types ti^*, representing the number and types of values on the stack. The *local variable environment l* tracks the indexed types of local variables, so constraints can refer to local variables. The locals environment has the same representation as a stack type: a sequence of indexed types. A *constraint set ϕ* is, well, a set of constraints between index terms. The index environment Γ describes which index variables are in scope before or after the instruction executes. This is a formal detail used to reason about the scope of index variables; we omit it from the presentation of typing rules in this section. The full typing details are in Appendix A.

Constraints in an instruction type are written in the *index language*, given in Figure 5:

- P is a *constraint* about index terms: either an equality constraint, or a proposition combining constraints using a simple first-order logic;
- r is a, including index variables.n *index term*: either an index type variable, a constant with an explicit value type, or a model of a Wasm operation on values;
- α is an *index variable*, representing a specific run-time value;
- t is a *value type*, which coarsely classifies a run-time value;

Finally, the whole module instance is typed under a *module environment C*, with type information about the module and the execution context. C is a partial record containing:

- C_{func}, the types of functions in the module;
- C_{global}, the types of global variables in the module;
- C_{table}, the number and types of functions in the table if the module has one, and undefined otherwise;
- C_{memory}, the (initial) size of memory if the module has one, and undefined otherwise;
- C_{local}, the value types of the local variables, which is defined when typing a function body (this is redundant with the local variables environment in the instruction type, but retained for backwards compatibility);
- C_{label}, the stack of label types, which is used for typing branching instructions. Label types are either the precondition for loops, and postcondition for other blocks.
- C_{return}, the return type used to type the return instruction. Return types are just the postcondition stack type and constraint set, since local variables leave scope after a return.

3.3.2 Implication. Unlike in a simple type system, we cannot simply syntactically compare a postcondition to precondition to type check two instructions. For example, a function expecting a value greater than zero might be given a value that is greater than ten. That should be fine, as semantically a value greater than ten is greater than zero, but these types differ syntactically.

We use a notion of logical implication for the logic corresponding to our index language for checking agreement between constraint sets. We define logical implication in Wasm-precheck as follows: $\Gamma \vdash \phi_1 \implies \phi_2$ if every valid variable assignment for ϕ_1 is also valid for ϕ_2. Formally:

$\Gamma \vdash \phi_1 \Rightarrow \phi_2 = \forall(t\ \alpha) \in \Gamma.\ \exists y \in t.\ \phi_1[\alpha := y]$ implies $\phi_2[\alpha := y]$. This can be read as saying that a constraint set ϕ_1 *implies* another constraint set ϕ_2 under Γ if, given the type declarations for index variables in Γ, the set of possible assignments to those variables under ϕ_1 is a subset of the set of possible assignments under ϕ_2.

The type system is parameterized by the implementation of $\Rightarrow$, denoted using $\rightsquigarrow$. We do not require $\rightsquigarrow$ to be complete (always returning true if one constraint set does in fact imply another), but we do require it to be sound (never returning true when one constraint set does not imply another), allowing any implementation $\rightsquigarrow$ to be an *under-approximation* of $\Rightarrow$. Formally: $\forall \Gamma, \phi_1, \phi_2.\ \Gamma \vdash \phi_1 \rightsquigarrow \phi_2$ implies $\Gamma \vdash \phi_1 \Rightarrow \phi_2$ This allows flexibility, as an implementation can use a faster constraint solver that may not be as precise as the theoretical notion of implication. While this may reduce the static reasoning ability, safety is maintained.

3.3.3 Typing Judgement. The typing judgment $C \vdash e^* : ti_1^*; l_1; \phi_1 \rightarrow ti_2^*; l_2; \phi_2$ states that under the module environment C, the instruction sequence e^* produces the configuration with described by $ti_2^*; l_2; \phi_2$ if it is executed in a configuration described by $ti_1^*; l_1; \phi_1$. The stack must have type ti_2^* after execution if it had type ti_1^* before execution; the local variable types must be l_2 if they were l_1; the constraint set ϕ_2 must be satisfiable if ϕ_1 was. We gradually present the (simplified to elide Γ) typing rules inline; the complete definitions are in Appendix A.

We first discuss some simple rules that do not use indexed type information. Rule Unreachable accepts any precondition and guarantees any postcondition since it causes a trap. The instruction nop makes no changes from the pre to the postcondition because the instruction does nothing. Rule Drop consumes the top value from the stack, represented by α, and does not change anything.

$$\textsc{Unreachable}\quad \frac{}{C \vdash \mathsf{unreachable} : ti_1^*; l_1; \phi_1 \rightarrow ti_2^*; l_2; \phi_2} \qquad \textsc{Nop}\quad \frac{}{C \vdash \mathsf{nop} : \epsilon; l; \phi \rightarrow \epsilon; l; \phi} \qquad \textsc{Drop}\quad \frac{}{C \vdash \mathsf{drop} : (t\ \alpha); l; \phi \rightarrow \epsilon; l; \phi}$$

The constant instruction is a simple example of indexed types. Intuitively, $t.\mathsf{const}\ c$ pushes the constant value c of type t onto the stack. The typing rule Rule Const reflects this: in the postcondition, the first value on the stack has indexed type $(t\ \alpha)$ for fresh index variable α. The postcondition includes a constraint that α is equal to the constant c, resulting in constraint set $\phi, (=\alpha\ (t\ c))$. The locals environment l is unchanged.

$$\textsc{Const}\ \frac{\alpha\ \texttt{fresh}}{C \vdash t.\mathsf{const}\ c : \epsilon; l; \phi \rightarrow (t\ \alpha); l; \phi, (=\alpha\ (t\ c))}$$

Typing rules for binary, test, and relational operations are all similar except for the operators and number of operations; we explain binary operations in detail. Rule Binop adds constraints between new and old program values. In the post condition, the fresh index variable α_3 is constrained to be equal to the result of applying the operator to the two index variables (α_1 and α_2) on the stack in the precondition: $(=\alpha_3\ (\|binop\|\ \alpha_1\ \alpha_2)$. We use $\|binop\|$ to indicate that we are moving *binop* (or *relop* or *testop*) from a Wasm-precheck to the index language, where it is modeled as a function rather than a stack machine instruction. Again, the locals environment l is unchanged.

Rule Div-Prechk, for the prechecked division operator, requires that the second operand is non-zero. The premise $\phi \rightsquigarrow \neg(=\ \alpha_2\ 0)$ requires that the index constraints satisfy the proposition $\alpha_2 \neq 0$. Since divide-by-zero is proven absent statically, it is safe to use div✓ without dynamic checks.

$$\textsc{Binop}\ \frac{\alpha_3\ \texttt{fresh}}{C \vdash t.binop : (t\ \alpha_1)\ (t\ \alpha_2);\ l;\ \phi \rightarrow (t\ \alpha_3);\ l;\ \phi, (=\alpha_3\ (\|binop\|\ \alpha_1\ \alpha_2))}$$

$$\textsc{Div-Prechk}\ \frac{\phi \rightsquigarrow \neg(=\ \alpha_2\ 0) \qquad \alpha_3\ \texttt{fresh}}{C \vdash t.\text{div}\checkmark\ : (t\ \alpha_1)\ (t\ \alpha_2);\ l;\ \phi \rightarrow (t\ \alpha_3);\ l;\ \phi, (=\alpha_3\ (\text{div}\ \alpha_1\ \alpha_2))}$$

$$\textsc{Relop}\ \frac{\alpha_3\ \texttt{fresh}}{C \vdash t.relop : (t\ \alpha_1)\ (t\ \alpha_2);\ l;\ \phi \rightarrow (t\ \alpha_3);\ l;\ ;\ \phi, (=\alpha_3\ (\|relop\|\ \alpha_1\ \alpha_2))}$$

$$\textsc{Testop}\ \frac{\alpha_2\ \texttt{fresh}}{C \vdash t.testop : (t\ \alpha_1);\ l;\ \phi \rightarrow (t\ \alpha_2);\ l;\ \phi, (=\alpha_2\ (\|testop\|\ \alpha_1))}$$

$$\textsc{Unop}\ \frac{\alpha_2\ \texttt{fresh}}{C \vdash t.unlop : (t\ \alpha_1);\ l;\ \phi \rightarrow (t\ \alpha_2);\ l;\ \phi, (=\alpha_2\ (\|unop\|\ \alpha_1))}$$

Recall that select is a ternary operator that consumes three values from the stack (α_1, α_2, and α_3) and returns the first value, α_1, if the third value, α_3 is truthy (non-0), and otherwise returns the second value α_2. The third value must be an i32. Rule Select uses the type-level "if" to constraint the result variable α to depend on the truthiness of α_3: (if $(=\alpha_3\ (\text{i32}\ 0))\ (=\alpha\ \alpha_2)\ (=\alpha\ \alpha_1)$).

$$\textsc{Select}\ \frac{\alpha\ \texttt{fresh}}{C \vdash \text{select} : (t\ \alpha_1)\ (t\ \alpha_2)\ (\text{i32}\ \alpha_3);\ l, \phi \rightarrow (t\ \alpha);\ l;\ \phi, (\text{if}\ (=\alpha_3\ (\text{i32}\ 0))\ (=\alpha\ \alpha_2)\ (=\alpha\ \alpha_1))}$$

Control flow blocks. The three block instructions check their bodies with additional information added to the environment C to handle branching instructions within the bodies. This causes a minor different between our model and the implementation. In the model, we merely require the existence of a constraint set relating the label type to the pre or postcondition of the block. In practice, we require this be a user-provided annotation, discussed in Section 5. It may also be possible to find such a set using an inference algorithm, discussed in Section 8.

In Rule Block, the premise says that executing the body, e^*, must be well typed at the current precondition, $(t_1\ \alpha_1)^*;\ (t_l\ \alpha_{l1})^n;\ \phi_1$, and then results in some postcondition $(t_2\ \alpha_2)^m;\ (t_l\ \alpha_{l2})^*;\ \phi_2$, which is then the postcondition for the block as a whole. However, the body, e^*, is not type checked with the same module type context C of the block, but rather a modified context with a *label type* pushed onto the stack of label types C_{label}. Any branching instruction within the block is typed against the new C_{label}. For block, the added label type is similar, but stronger than, to the postcondition: $(t_2\ \ \alpha_2)^*;\ (t_l\ \ \alpha_{l2})^*;\ \phi_3$. The end of the block can be reached through a branching instruction, or the body e^* being evaluated to a sequence of values. Thus, the label type and postcondition of the body e^* must somehow agree, so that the postcondition of the block is guaranteed to hold no matter what path out of the block taken. To ensure this, the label type and postcondition must have the same stack and index local store, and the constraint set in the label type, ϕ_2 (which is also the constraint set of the overall postcondition of the block), must be implied the constraint set ϕ_3 from the postcondition of the body e^*. Thus, ϕ_2 represents the point of agreement between executing the body, e^*, and any branching instruction from within the body.

Rule If similarly checks both possible branches with an updated context, but with extra information based on whether the condition variable α is true or not, depending on the branch. In the "true" branch e_1^*, the index variable α consumed by the if is known to be truthy, *i.e.*, non-zero, whereas in the "false" branch e_2^*, α is zero. Thus, the two branches do start from the same precondition ϕ_1, but with the added constraint $\neg(=\ \alpha\ (\text{i32}\ 0))$ in the true branch, and $(=\ \alpha\ (\text{i32}\ 0))$ in the false branch.

For if, the two branches must agree as well, so they are required to have the same postcondition, except for the resulting constraint sets, which both must imply an agreed upon constraint set ϕ_2 from the label type.

Branching from within a loop re-executes the loop, so when type checking the loop body, the added label type is the precondition. Thus, instead of the label type and postcondition needing to agree, the label type and precondition need to agree. Therefore, instead of the body e^* being checked against the precondition of the whole loop, $(t_1\ \alpha_1)^*;\ (t_l\ \alpha_{l1})^n;\ \phi_1$, it is instead checked with the precondition $(t_1\ \alpha_1)^*;\ (t_l\ \alpha_{l1})^n;\ \phi_3$, which is reachable either from a branching instruction or the first time the loop is executed.

$$\textsc{Block}\ \frac{C, \text{label}\ ((t_2\ \alpha_2)^*;\ (t_l\ \alpha_{l2})^*;\ \phi_2) \vdash e^* : (t_1\ \alpha_1)^*;\ (t_l\ \alpha_{l1})^*;\ \phi_1 \rightarrow (t_2\ \alpha_2)^*;\ (t_l\ \alpha_{l2})^*;\ \phi_3 \qquad \phi_3 \rightsquigarrow \phi_2}{C \vdash \text{block}\ (t_1^* \rightarrow t_2^*)\ e^*\ \text{end} : (t_1\ \alpha_1)^*;\ (t_l\ \alpha_{l1});\ \phi_1 \rightarrow (t_2\ \alpha_2)^*;\ (t_l\ \alpha_{l2})^*;\ \phi_2}$$

$$\textsc{Loop}\ \frac{\begin{array}{c} C, \text{label}\ ((t_1\ \alpha_1)^*;\ (t_l\ \alpha_{l1})^*;\ \phi_3) \vdash e_1^* : (t_1\ \alpha_1)^*;\ (t_l\ \alpha_{l1})^*;\ (t_1\ \alpha_1)^*\ (t_l\ \alpha_{l1})^*;\ \phi_3 \rightarrow (t_2\ \alpha_2)^*;\ (t_l\ \alpha_{l2})^*;\ \phi_4 \\ \phi_1 \rightsquigarrow \phi_3 \qquad \phi_4 \rightsquigarrow \phi_2 \end{array}}{C \vdash \text{loop}\ t_1^* \rightarrow t_2^*\ e^*\ \text{end} : (t_1\ \alpha_1)^*;\ (t_l\ \alpha_{l1});\ \phi_1 \rightarrow (t_2\ \alpha_2)^*;\ (t_l\ \alpha_{l2})^*;\ \phi_2}$$

$$\textsc{If}\ \frac{\begin{array}{c} C, \text{label}\ ((t_2\ \alpha_2)^*;\ (t_l\ \alpha_{l2})^*;\ \phi_2) \vdash e_1^* : (t_1\ \alpha_1)^*;\ (t_l\ \alpha_{l1})^*;\ \phi_1, \neg(=\alpha\ (\text{i32}\ 0)) \rightarrow (t_2\ \alpha_2)^*;\ (t_l\ \alpha_{l2})^*;\ \phi_3 \\ C, \text{label}\ ((t_2\ \alpha_2)^*;\ (t_l\ \alpha_{l2})^*;\ \phi_2) \vdash e_2^* : (t_1\ \alpha_1)^*;\ (t_l\ \alpha_{l1})^*;\ \phi_1, (=\alpha\ (\text{i32}\ 0)) \rightarrow (t_2\ \alpha_2)^*;\ (t_l\ \alpha_{l2})^*;\ \phi_4 \\ \phi_3 \rightsquigarrow \phi_2 \qquad \phi_4 \rightsquigarrow \phi_2 \end{array}}{C \vdash \text{if}\ t_1^* \rightarrow t_2^*\ e_1^*\ \text{else}\ e_2^*\ \text{end} : (\text{i32}\ \alpha)\ (t_1\ \alpha_1)^*;\ (t_l\ \alpha_{l1});\ \phi_1 \rightarrow (t_2\ \alpha_2)^*;\ (t_l\ \alpha_{l2})^*;\ \phi_2}$$

Recall that branching (br j) consumes values v^n and jumps to the jth label in the evaluation context, continuing executing with v^n on the stack. Thus, the precondition of br checks against the j-th (counting backwards from the top) label type on the stack C_{label}. As with control flow blocks, the precondition of the branch may be stronger than the label type. The instructions following a branch are not executed, so the postcondition, $ti_2^*;\ l_2;\ \phi_2$, is arbitrary.

Rule Return is similar to Rule Br, except that return is checked against the current return type C_{return} instead of against a label type. Return types do not include a locals environment, since local variables are only scoped within functions; after a return, they all go out of scope. Like br, code after a return is dead, so the postcondition of return is arbitrary: $ti_2^*;\ l_2;\ \phi_2$.

Recall that br_if consumes a value α from the stack and branches if it is truthy, so Rule Br-If adds the constraint that α is zero to the postcondition. In contrast to Rule Br, execution can continue after br_if, specifically when α is zero and branching doesn't occur. If the consumed value is constrained to be non-zero in the type system, then this causes a contradiction in the constraint set ϕ, indicating dead code.

Finally, br_table is essentially a br j where j is determined by indexing into a statically provided list of branching indices i^+ using the operand from the stack. We must ensure that every possible label type that might be the target is implied by the precondition of the br_table instruction. Like br, br_table must branch, so the postcondition is arbitrary.

$$\textsc{Return}\ \frac{C_{\text{return}} = ti_3^*;\ \phi_3 \qquad \phi_1 \rightsquigarrow \phi_3}{C \vdash \text{return} : ti_1^*\ ti_3^*;\ l_1;\ \phi_1 \rightarrow ti_2^*;\ l_2;\ \phi_2} \qquad \textsc{Br}\ \frac{C_{\text{label}}(i) = ti_3^*;\ l_1;\ \phi_3 \qquad \phi_1 \rightsquigarrow \phi_3}{C \vdash \text{br}\ i : ti_1^*\ ti_3^*;\ l_1;\ \phi_1 \rightarrow ti_2^*;\ l_2;\ \phi_2}$$

$$\textsc{Br-If}\ \frac{C_{\text{label}}(i) = ti_1^*;\ l;\ \phi_3 \qquad \phi_1, \neg(=\alpha\ (\text{i32}\ 0)) \rightsquigarrow \phi_3}{C \vdash \text{br\_if}\ i : ti_1^*\ (\text{i32}\ \alpha);\ l;\ \phi_1 \rightarrow ti_1^*;\ l;\ \phi_1, (=\alpha\ (\text{i32}\ 0))} \qquad \textsc{Br-Table}\ \frac{(C_{\text{label}}(i) = ti_1^*;\ l_1;\ \phi_i)^* \qquad \phi_1 \rightsquigarrow \phi_i^*}{C \vdash \text{br\_table}\ i^+ : ti_1^*\ (\text{i32}\ \alpha);\ l_1;\ \phi_1 \rightarrow ti_2^*;\ l_2;\ \phi_2}$$

Direct function calls call i look up the type annotation of the function, $ti_1^*;\ \phi_3 \rightarrow ti_2^*;\ \phi_2$, in the environment C. The current state, ϕ_1 must satisfy the precondition constraints of the function, ϕ_3, which ensures that the assumptions made by the function hold. We omit the locals environment from the type annotation, since local variables are not preserved or accessible across function calls. The postcondition of call extends the constraint set from the precondition, ϕ_1, with the postcondition from the function we are calling ϕ_2. This is because the type annotation on the function can only contain constraints about the arguments, so simply copying the postcondition from the annotation would result in the loss of information about all other index variables.

Typing an indirect function call call_indirect, is similar to typing a direct call, except that the expected type is based on the statically provided type annotation. This is because call_indirect does not know, statically, the type of the function being called, so must be provided with the expected type to be able to check the type and compute the resulting state. Rule Call-Indirect also ensures that there is a table defined for the module using the side condition $C_{\text{table}} = (n, tfi^n)$, which ensures that a table is present for the module which contains n functions and provides the associated function types tfi^n.

Proving the safety of a prechecked indirect function call is more complex. This involves statically checking that the actual precondition satisfies the precondition on every possible function that could be called. Rule Call-Indirect-Prechk checks that the type of every function at every possible index value has a subtype of the expected type: $\forall 0 \leq i < n.(\phi \rightsquigarrow \neg(= (\text{i32}\ i)\ \alpha)) \vee ((ti_1^*; \phi_1 \to ti_2^*; \phi_2)^*)(i) = (ti_3^*; \phi_3 \to ti_4^*; \phi_4)$. $((ti_1^*; \phi_1 \to ti_2^*; \phi_2)^*)(i)$ is a shorthand for looking up the ith function type in the sequence $(ti_1^n; \phi_1 \to ti_2^*; \phi_2)^*$. Note that the $\forall$ and $\vee$ are at the meta level and not within the index language, and that the size of the table n is statically known. The rule also checks that the operand is within the table bounds: $\phi \rightsquigarrow (\text{gt}\ n\ \alpha)$.

$$\textsc{Call}\ \frac{C_{\text{func}}(i) = ti_1^*; \phi_3 \to ti_2^*; \phi_2 \qquad \phi_1 \rightsquigarrow \phi_3}{C \vdash \text{call}\ i :\ ti_1^*; l; \phi_1 \to ti_2^*; l; \phi_1 \cup \phi_2}$$

$$\textsc{Call-Indirect}\ \frac{C_{\text{table}} = (n, tfi^n) \qquad \phi_1 \rightsquigarrow \phi_2}{C \vdash \text{call\_indirect}\ (ti_1^*; \phi_2 \to ti_2^*; \phi_3) : ti_1^*\ (\text{i32}\ \alpha); l; \phi_1 \to ti_3^*; l; \phi_1 \cup \phi_3}$$

$$\textsc{Call-Indirect-Prechk}$$
$$\frac{C_{\text{table}} = (n, tfi^n) \qquad \phi_1 \rightsquigarrow \phi_2 \qquad \forall i < n.\ (\phi_1 \rightsquigarrow \neg(= (\text{i32}\ i)\ \alpha)) \vee\ tfi^*(i) = ti_1^*; \phi_2 \to ti_2^*; \phi_3}{C \vdash \text{call\_indirect}✓\ (ti_1^*; \phi_2 \to ti_2^*; \phi_3) : ti_1^*\ (\text{i32}\ \alpha); l; \phi_1 \to ti_2^*; l; \phi_1 \cup \phi_3}$$

Typing instructions for local variables, Rule Get-Local, Rule Set-Local, and Rule Tee-Local, all dereference the type from the locals environment l at the statically de Bruijn index i, denoting the ith local variable. Rule Get-Local puts a fresh index variable, α_2, on the stack with the value type t of the ith local, and constrains it to be equal to the ith local variable. Rule Set-Local works in the reverse direction, replacing the index variable associated with the local variables being assigned. For set_local, we replace the indexed type of the local variable with the indexed type from stack; since the index variables in these types identify the value from the stack, this reflects the value being moved from the stack to the store. Finally, Rule Tee-Local is a combination of the above two rules, as the instruction is a combination of get_local and set_local.

$$\textsc{Get-Local}\ \frac{C_{\text{local}}(i) = t \qquad l(i) = (t\ \alpha) \qquad \alpha_2\ \texttt{fresh}}{C \vdash \text{get\_local}\ i : \epsilon; l; \phi \to (t\ \alpha_2); l; \phi, (= \alpha\ \alpha_2)}$$

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

$$\textsc{Tee-Local}\ \frac{C_{\text{local}}(i) = t \qquad l_2 = l_1[i := (t\ \alpha)] \qquad \alpha_2\ \texttt{fresh}}{C \vdash \text{tee\_local}\ i : (t\ \alpha); l_1; \phi \to (t\ \alpha_2); l_2; ; \phi, (= \alpha\ \alpha_2)}$$

Global variables are shared between modules and can be mutable, so we do not track constraints on globals; we discuss this limitation more in Section 8. Rule Set-Global checks that the global being set is mutable and has the same type as the operand. Rule Get-Global introduces a fresh index variable α with the type t of the ith global variable from the context.

$$\textsc{Get-Global}\ \frac{C_{\text{global}}(i) = \text{mut}^{?}\ t \qquad \alpha\ \texttt{fresh}}{C \vdash \text{get\_global}\ i : \epsilon;\ l;\ \phi \longrightarrow (t\ \alpha);\ l;\ \phi} \qquad \textsc{Set-Global}\ \frac{C_{\text{global}}(i) = \text{mut}\ t}{C \vdash \text{set\_global}\ i : (t\ \alpha);\ l;\ \phi \longrightarrow \epsilon;\ l;\ \phi}$$

We do not reason about the contents of memory, so the non-prechecked memory instructions do not add constraints. All the memory instruction typing rules ensure the module has a declared memory using the side condition $C_{\text{memory}} = n$, which looks up the initial size of memory in the module type context C. Rule Mem-Load and Rule Mem-Store ensure that the alignment fits the type being loaded or stored $2^a \leq (|tp| <)^{?}|t|$. Rule Mem-Store simply checks that the second operand has the expected type t. Rule Grow-Memory simply consumes a 32-bit integer (the additional amount of memory the user would like to allocate), and returns a 32-bit integer value, representing the updated amount of memory if the allocation was successful, and -1 otherwise.

Prechecked memory instructions are statically checked to take place within the static memory bounds. We currently do not reason about dynamically increasing memory size, which we discuss further in Section 8. The initial memory size is some number of 64 Ki pages (65, 536 bytes), so we check that the constraint set in the precondition implies that the memory index α plus the static offset o is less than $65,536 - width$ (memory indices are unsigned, so they cannot be less than 0). $width$ is a shorthand for the number of bytes being stored or loaded. It is equal to either is the length in bytes of the type value t if tp is not provided ($tp^{?} = \epsilon$), and otherwise equal the length in bytes of tp, a packed type (used to load or store a slice of 8 bits, 16 bits, or 32 bits).

$$\textsc{Mem-Load}\ \frac{C_{\text{memory}} = n \qquad \alpha_2\ \texttt{fresh}}{C \vdash t.\text{load}\ (tp\_sx)^{?}\ o : (\text{i32}\ \alpha_1);\ l;\ \phi \longrightarrow (t\ \alpha_2);\ l;\ \phi} \qquad \textsc{Mem-Store}\ \frac{C_{\text{memory}} = n}{C \vdash t.\text{store}\ tp^{?}\ o : (\text{i32}\ \alpha_1)\ (t\ \alpha_2);\ l;\ \phi \longrightarrow \epsilon;\ l;\ \phi}$$

$$\textsc{Load-Prechk}\ \frac{C_{\text{memory}} = n \qquad \alpha_2\ \texttt{fresh} \qquad \phi \rightsquigarrow (\text{le}\ (\text{add}\ \alpha_1\ (\text{i32}\ o + width))\ (\text{i32}\ n * 64\text{Ki}))}{C \vdash t.\text{load}✓\ (tp\_sx)^{?}\ o : (\text{i32}\ \alpha_1);\ l;\ \phi \longrightarrow (t\ \alpha_2);\ l;\ \phi}$$

$$\textsc{Store-Prechk}\ \frac{C_{\text{memory}} = n \qquad \phi \rightsquigarrow (\text{le}\ (\text{add}\ \alpha_1\ (\text{i32}\ o + width))\ (\text{i32}\ n * 64\text{Ki}))}{C \vdash t.\text{store}✓\ tp^{?}\ o : (\text{i32}\ \alpha_1)\ (t\ \alpha_2);\ l;\ \phi \longrightarrow \epsilon;\ l;\ \phi}$$

$$\textsc{Current-Memory}\ \frac{C_{\text{memory}} = n \qquad \alpha\ \texttt{fresh}}{C \vdash \text{current\_memory} : \epsilon;\ l;\ \phi \longrightarrow (\text{i32}\ \alpha);\ l;\ \phi} \qquad \textsc{Grow-Memory}\ \frac{C_{\text{memory}} = n \qquad \alpha_2\ \texttt{fresh}}{C \vdash \text{grow\_memory} : (\text{i32}\ \alpha_1);\ l;\ \phi \longrightarrow (\text{i32}\ \alpha_2);\ l;\ \phi}$$

The last rules handle composing sequences of instructions. Rule Empty types the empty instruction sequence ϵ, which simply has the same pre and postcondition $\epsilon;\ l;\ \phi$. Rule Stack-Poly allows a prefix of the stack to be ignored (or added, depending on your perspective); this adds polymorphism in "the rest" of the stack to all the other typing rules. Rule Composition composes a sequence of instructions e_1^* with another instruction e_2, checking that pre and postconditions match up.

$$\textsc{Empty}\ \frac{}{C \vdash \epsilon : \epsilon;\ l;\ \phi \longrightarrow \epsilon;\ l;\ \phi} \qquad \textsc{Stack-Poly}\ \frac{C \vdash e^* : ti_1^*;\ l_1;\ \phi_1 \longrightarrow ti_2^*;\ l_2;\ \phi_2}{C \vdash e^* : ti^*\ ti_1^*;\ l_1;\ \phi_1 \longrightarrow ti^*\ ti_2^*;\ l_2;\ \phi_2}$$

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

4 METATHEORY

First, we show how to automatically translate Wasm programs to Wasm-precheck programs and vice versa. Then, we prove the type safety of Wasm-precheck. We provide key insights and details here; complete definitions and proofs are provided in the anonymous supplementary material.

4.1 Embedding Wasm into Wasm-precheck

The embedding function takes a Wasm module and replaces all type annotations with indexed types that have no constraints on the index variables. Intuitively, this works because the type annotations are the only part of the surface syntax of Wasm that differs in Wasm-precheck, and the constraints are only necessary to type check prechecked instructions. While this embedding requires no additional developer effort, it provides no information to the indexed type system beyond what can be trivially inferred, so it may not automatically improve static reasoning, and does not automatically provide prechecked instructions. More sophisticated embeddings could attempt to insert prechecked instructions; we discuss this Section 8.

First, we define embedding over modules: the top-level object of both the Wasm and Wasm-precheck surface syntax. Embedding a module m means embedding all functions f^* and globals $glob^*$ in the module. The definition of embedding is not interesting; we recur over the syntax looking for type annotations, and enriching them to indexed types with fresh index variables and empty constraint sets. The definition can be found in the anonymous supplementary material. We do not transform the table $tab^?$, or the memory $mem^?$ as Wasm and Wasm-precheck use the same syntax to define them (although Wasm-precheck represents the types of tables differently).

For clarity, we typeset Wasm-precheck instructions in a blue sans serif font and Wasm instructions in a **bold red serif font**.

Definition 1. $\boxed{embed_m(m) = m}$

$$embed_m(\textbf{module}\ f^*\ glob^*\ tab^?\ mem^?) \quad = \quad \mathsf{module}\ embed_f(f)^*\ embed_g(glob)^*\ tab^?\ mem^?$$

For example, the Wasm program on the left below embeds into the Wasm-precheck program seen on the right below.

```
module                              module
  func (i32 → i32) local ε            func ((i32 α₁); ∅ → (i32 α₂); ∅) local ε
    get_local 0                          get_local 0
    i32.const 1                          i32.const 1
    i32.div                              i32.div
  end                                  end
```

Theorem 1 (Well Typed Embedding).
If $\vdash$ **module** $f^*\ glob^*\ tab^?\ mem^?$, then $\vdash embed_m($**module** $f^*\ glob^*\ tab^?\ mem^?)$

Proof. (Sketch) The proof follows by induction on the structure of the Wasm typing derivation. The full proof is available in Appendix B.1 □

4.2 Erasing Wasm-precheck Annotations

We provide an erasure function from Wasm-precheck programs to Wasm programs by discarding the extra type information and replacing prechecked instructions with their non-prechecked counterparts. Erasure is defined not just for the surface syntax, but also for typing constructs (such as the module environment), administrative instructions, and run-time data structures (such as the store). This extended definition of erasure allows us to reason about the behavior of Wasm-precheck run-time programs in Wasm, which is useful in the type safety proof.

Erasure is best illustrated with an example. Full erasure definitions and proofs can be found in the anonymous supplementary material, but their formal details are not insightful. The function annotation constrains the input α_1 to be greater than 0. This lets a div✓ be used with the input as a divisor. The annotation also includes the primitive Wasm types, which is the only information needed for type checking under Wasm, so we get rid of all other information to produce a Wasm type annotation, as well as replacing div✓ with the Wasm instruction **div**.

```
module
  func ((i32 α1); (= (i32 α1) (i32.gt_u α1 (i32 0)))
       → (i32 α2); (i32.gt_u α2 (i32 0))) local ϵ
    get_local 0
    i32.const 1
    i32.div✓
  end
```

```
module
  func (i32 → i32) local ϵ
    get_local 0
    i32.const 1
    i32.div
  end
```

The key theorem is that erasing a well-typed Wasm-precheck (run time) machine configuration produces a well-typed Wasm (run time) machine configuration, so all Wasm-precheck programs erase to running, type safe Wasm programs. This is useful in showing type safety, since intuitively, no reduction rule is type directed, so if erasing the types results in a type safe Wasm program, then reduction in Wasm-precheck is also type safe. Note that $erase_v(v^*) = v^*$ trivially. The proofs are available in Section 4.2.

Theorem 2 (Erasure Preserves Typing). If $\vdash_i s; v^*; e^* : (t\ \alpha)^*; l; \phi$, then $\vdash_i erase_s(s); v^*; erase_{e^*}(e^*) : t^*$

For compile-time typing, the key lemma is that erasure for instructions preserves typing.

Lemma 1 (Instruction Erasure Preserves Typing). If $C \vdash e^* : (t_1\ \alpha_1)^*; l_1; \phi_1 \longrightarrow (t_2\ \alpha_2)^*; l_2; \phi_2$, then $erase_C(C) \vdash erase_{e^*}(e^*) : t_1^* \longrightarrow t_2^*$

4.3 Type Safety

Type safety is the property that a well-typed machine state either reduces to another well-typed state (perhaps infinitely), a sequence of values, or evaluates to the well-defined error trap. Type safety of Wasm-precheck guarantees a number of important properties, including memory safety.

To reason about the run-time store s, a run-time store type S is introduced. The store context S contains the type information for everything in s: module instances, tables, and memories. Every module instance in s has an associated module type context in S, for example, the ith module instance would have the type $S_{\text{inst}}(i)$. The module type context $S_{\text{inst}}(i)$ is familiar to us as C.

Additional administrative typing judgments necessary for the proof are discussed in Appendix A.2.

For the type safety proof, we define an evaluation function $eval(s; v^*; e^*)$ using $\hookrightarrow_i^*$, the transitive, reflexive closure of $s; v^*; e^* \hookrightarrow_i s'; v'^*; e'^*$. The evaluation function has three possible outcomes: the program may terminate, returning a sequence of values $\hat{v}^*$; the program may trap, returning the trap instruction which represents a fatal run-time error; or the program may not terminate.

Definition 2. $\boxed{eval(s; v^*; e^*)}$

$$\begin{array}{rcl} eval(s; v^*; e^*) & = & \hat{v}^*, \text{ if } s; v^*; e^* \hookrightarrow_i^* s'; v'^*; \hat{v}^* \\ eval(s; v^*; e^*) & = & \text{trap}, \text{ if } s; v^*; e^* \hookrightarrow_i^* s'; v'^*; \text{trap} \end{array}$$

Theorem 3 (Type Safety). If $\vdash_i s; v^*; e^* : ti^*; l; \phi$, then either $eval(s; v^*; e^*) = \hat{v}^*$, $eval(s; v^*; e^*) =$ trap, or $eval(s; v^*; e^*)$ doesn't terminate.

Proof. Follows from Lemma 25 () and Lemma 2 (Subject Reduction). □

Subject reduction, also known as *type preservation*, ensures that if a machine state $s; v^*; e^*$ has a given type, then the machine state $s'; v'^*; e'^*$ after a reduction step ($s; v^*; e^* \hookrightarrow_i s'; v'^*; e'^*$) will have an equivalent type. The main theorem for subject reduction allows the machine state after reduction, $s'; v'^*; e'^*$, to have the same type up to implication (the reduced expression may have a stronger postcondition).

Lemma 2 (Subject Reduction)**.**
If $\vdash_i s; v^*; e^* : ti^*; l; \Gamma; \phi, (= a\ c)^*$ and $s; v^*; e^* \hookrightarrow_i s'; v'^*; e'^*$, then $\vdash_i s'; v'^*; e'^* : ti^*; l; \Gamma; \phi$.

Proof. (Sketch) We use our inversion lemmas to gain information about the type of the store s and the local variables v^*, then hand that information to Lemma 3, which does most of the work.
The full proof is available in Section 4.3. □

Lemma 3 is the main lemma for subject reduction, and is the body of the "loop" that is type safety. We show that if a machine state $s; v^*; e^*$, reduces to $s'; v'^*; e'^*$, then the type of the new machine state matches. Formally, this means either e'^* has the same type, or it has a different locals environment l_1 in the precondition that matches the types of the locals v'^* after reduction. The local variables v^* are mutable, so the constraints on them ϕ_v^*, are not preserved. Instead, the initial state, ϕ_1, must have the initial constraints ϕ_v^* as part of it, and ϕ_3 will instead have the new constraints $\phi_v'^*$, as expressed by $\phi_3 = \phi_1 \bigcup \phi_v'^*$. In addition, if the initial store s has store type S, as stated by $\vdash s : S$, then the updated store s' has the same store type S, as stated by $\vdash s' : S$.

Lemma 3 (Subject Reduction for Instructions)**.** If $S; S_{\text{inst}}(i) \vdash e^* : ti_1^*; l_1; \phi_1 \longrightarrow ti_2^*; l_2; \phi_2$ and $s; v^*; e^* \hookrightarrow_i s'; v'^*; e'^*$

(1) $ti_1^*; l_1; \phi_1 \longrightarrow ti_2^*; l_2; \phi_2$ is well formed (ϕ_1 and ϕ_2 only reference bound index variables),
(2) $\vdash s : S$,
(3) $(\vdash v : (t_v\ \alpha_v); \phi_v)^*$, where $l_1 = (t_v\ \alpha_v)^*$ and $\phi_v \subset \phi_1$ (the local variables are well-typed and their constraints are included in the precondition)

then $S; S_{\text{inst}}(i) \vdash e'^* : ti_1^*; l_3; \phi_3, (= \alpha\ (t\ c))^* \longrightarrow ti_2^*; l_2; \phi_4$ for some α^*, t^*, and c^* where

(1) $ti_1^*; l_3; \phi_3, (= \alpha\ (t\ c))^* \longrightarrow ti_2^*; l_2; \phi_4$ is well formed,
(2) $\vdash s' : S$,
(3) $(\vdash v' : (t_v\ \alpha_v'); \phi_v')^*$, $l_3 = (t_v'\ \alpha_v')^*$, and $\phi_3 = \phi_1 \bigcup \phi_v'^*$ (the resulting local variables are well-typed),
(4) and $\phi_4 \Rightarrow \phi_2$ (the postcondition may be stronger, and imply the original postcondition)

Proof. (Sketch) The proof proceeds by case analysis on the reduction relation $s; v^*; e^* \hookrightarrow s'; v'^*; e'^*$.
The full proof is presented in Section 4.3. □

The main difficulty is reasoning about stack values consumed as a program reduces. Intuitively, after reduction, the constraints will be more specific, and thus stronger, than before reduction. We can weaken the types to recover the original types.

Lemma 25 () ensures that if a machine state is well typed then it either: entirely consists of values, is a trap, or it takes a step to another machine state. Lemma 25 is the key property that shows the static guarantees allow ✓-tagged instructions to reduce without dynamic checks. By proving that well-typed ✓-tagged instructions reduce, we are sure there is no undefined behaviour by leaving out a reduction rule for division-by-zero, for example.

Lemma 4 (Progress)**.** If $\vdash_i s; v^*; e^* : ti^*; l; \phi$ then either $e^* = \hat{v}^*$, $e^* = \text{trap}$, or $s; v^*; e^* \hookrightarrow_i s'; v'^*; e'^*$.

As with subject reduction, the main lemma is showing progress for individual instructions. In addition to the main typing premise, the lemma relies on some premises guaranteeing the well-formedness of program states. These express that there is some well-typed value prefix on the stack, that branches are statically well-bound, that the module instance's run-time memory, table, and store are well-typed w.r.t. to the module environment.

Lemma 5 (Progress for Instructions)**.** If $S; S_{\text{inst}}(i) \vdash e^* : ti_2^*; l_2; \phi_2 \longrightarrow ti_3^*; l_3; \phi_3$, where $e^* \neq (t.\text{const } c_2)^*$ (the instructions left on the stack are well typed),
and $S; S_{\text{inst}}(i) \vdash (t.\text{const } c)^* : \epsilon; l_1; \phi_1 \longrightarrow ti_2^*; l_2; \phi_2$ (the value prefix on the stack is well typed),
and if $(t.\text{const } c)^*\ e^* = L^k[\text{br } i]$, then $i \leq k$ (branches are well bound),
and $s_{\text{inst}}(i)_{\text{mem}} = b^n$, where $C_{\text{memory}} = n$ (memory is well sized),
and $s_{\text{inst}}(i)_{\text{tab}} = \{\text{inst } i, \text{func } \text{func } tfi \ldots\}^n$, where $C_{\text{table}} = (n, tfi^n)$ (the table is well sized and well typed),
and $(\vdash v : (t_v\ \alpha_v); \phi_v)^*$, where $S_{\text{inst}}(i)_{\text{local}} = t_v^*$ (locals are well typed),
then, either $\exists s'; v'^*; e'^*.\ s; v^*; (t.\text{const } c)^*\ e^* \hookrightarrow_i s'; v'^*; e'^*$ (the instruction take a step),
or $e^* = \epsilon$ (evaluation has finished with values $(t.\text{const } c)^*$),
or $e^* = \text{trap}$ and $(t.\text{const } c)^* = \epsilon$ (e^* has finished evaluating to a trap).

Proof. (Sketch) By induction on $S; S_{\text{inst}}(i) \vdash e^* : ti_2^*; l_2; \Gamma_2; \phi_2 \longrightarrow ti_3^*; l_3; \Gamma_3; \phi_3$ (where $S_{\text{inst}}(i)$ is a module type context, usually denoted by C). Since most Wasm-precheck instructions have the same dynamic semantics as in Wasm, and every Wasm-precheck type includes all the information of a Wasm type, we conclude that the Wasm-precheck term takes a step by translating to Wasm and using Wasm's type safety proof. The intuition is that most Wasm-precheck instructions have the same reduction rules in Wasm, so we can erase to Wasm, where Wasm's type safety guarantees the instructions satisfies progress. This does not work for prechecked instructions or inductive cases. The full proof is available in Section 4.3. □

5 IMPLEMENTATION

We implement Wasm-precheck as an extension of Wasmtime [Bytecode Alliance 2019], a fast, secure, compilant, runtime system for Wasm with JIT and AOT compilation. The implementation is straightforward, following the formal models. However, there are two details of interest: how we resolve join-points, and how we implemented constraint solving.

5.1 Type Annotations for Join-Points

Recall from Section 3.3 that block, if, and loop all introduce code points reachable from multiple paths due to branching. In these cases, we must find a set of constraints—the postcondition constraint set of blocks and ifs, and precondition constraint set for loops—that is implied by every path.

In our declarative formal model, we require only that such a set exists—the set does not come from the program syntax or from a subderivation. In the implementation, we require user provided type annotations specifying pre or postconditions on blocks to resolve these join-points.

The syntax of the annotations largely follows from the theory, except for how variables are referred to within the constraints. Within the annotations, the user can refer to stack variables by name, and locals by name or de Bruijn index.

For example, the annotation below denotes a type that takes a parameter a on the stack, and asserts that, in the precondition, the parameter a is less than the local variable b. If the local variable b was known to be the 0th local variable, then it could alternatively be referenced using `(local 0)`. Because the index language can only express constraints through equality, and the less than operator `i32.lt_u` returns a 32-bit integer, the output of the operator is explicitly checked for equality against the number 1, effectively checking for truthiness.

```
(type (func (param a i32) (pre (eq (i32 1) (i32.lt_u a (local b))))))
```

Similar to the model, annotations are not checked against the current state syntactically, but up to implication. The actual (as calculated by the type system) precondition at the start of the block must imply the expected precondition given by the type annotation. Similarly, the actual postcondition guaranteed by the body of the block must imply the expected postcondition from the type annotation. The overall type of the block is the actual precondition (stronger than the annotation) and the expected postcondition (weaker than the actual postcondition).

Type annotations are required to be well formed: a type annotation can only constrain the values consumed and produced by a function/block (also the local variables for blocks). Formally, in the pre and postcondition, all variables in the constraint set must appear in the stack type environment or local index store of that type annotation. Further, the precondition constraint set can only refer to parameters: variables that are part of the precondition stack or locals. However, the postcondition constraint set can refer to variables that are either part of the precondition or postcondition, to express relationships between parameters and results.

5.2 Constraint Solving

Like Wasm-precheck is parameterized by implication $\leadsto$, so is our implementation in Wasmtime. Any constraint solver can be used that implements the interface between the constraint solver and the index language. This interface is a Rust trait in our Wasmtime extension.

We choose Z3 for constraint solving for ease of use [De Moura and Bjørner 2008]. Our implementation uses Z3's bitvectors, resulting in a straightforward 1-to-1 relationship between Wasm-precheck operators and Z3 operators. In our Wasmtime implementation, we currently only support prechecked memory instructions (so call_indirect✓ and div✓ are disabled) for reasons discussed in Section 6 and Section 8. However, we have separately implemented typing rules for these instructions via an encoding to Z3 in our Redex model.

5.3 Redex Model

We provide a reference implementation of the formal model of Wasm-precheck in Redex [Felleisen et al. 2009], which also uses Z3 for constraint solving. Our implementation includes a model of the type system that checks whether a given typing derivation is valid in our model, and a syntax-directed algorithm for generating typing derivations from Wasm-precheck programs. The former can be used to validate type-inference algorithms for Wasm-precheck. The implementation also includes each of these for plain Wasm, which are reused in the implementation of Wasm-precheck.

The key challenge in the reference implementation was encoding constraints for the function table and indirect function calls. Recall that for call_indirect✓ *tfi*, we have to encode constraints about which functions in a table can be called. To encode this, we construct a Z3 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, *i.e.,* is a subtype of the expected type *tfi*, and `false` otherwise. Finally, we assert all of the translated constraints from the constraint set about the table index, and constrain that the value in the array at the table index is `true`.

6 EVALUATION

Our evaluation seeks to answer the following questions:

(1) What is the best case performance speed-up of removing dynamic checks from Wasm?
(2) What speed-up can we realistically get using Wasm-precheck?
(3) What is the added cost of the Wasm-precheck type system compared to Wasm's?

6.1 General Setup

We use the PolyBenchC benchmark suite [Pouchet and Yuki 2016] and the Wasmtime runtime and compiler [Bytecode Alliance 2019] to perform our evaluation.

We compare three versions of Wasmtime: a "Wasm" version with virtual memory guard pages disabled and dynamic bounds checks enabled (the baseline we want to improve); a "no-checks" version with all safety checks disabled (used for run-time performance comparison, providing a frame of reference for best case improvements); and the "Wasm-precheck" version, our extension of Wasmtime implementing Wasm-precheck. We emphasize that this evaluation is just as much an evaluation of the Wasmtime implementations as it is of the Wasm and Wasm-precheck languages, and other compilers may perform differently.

Name	Wasmtime Version
Wasm	Wasmtime with guard pages removed, dynamic bounds checks enabled
Wasm-no-checks	Wasmtime with dynamic memory checks disabled (unsafe)
Wasm-precheck	Wasmtime extended to implement Wasm-precheck

We use two versions of the PolyBenchC suite: one unmodified version compiled from C to Wasm with Emscripten with `emcc -Os` [Emscripten Contributors 2015], and one version manually ported from Wasm to Wasm-precheck with type annotations added and some instructions modified. The unmodified version is used with both the "Wasm" and the "no-checks" versions of Wasmtime.

The ported version is used in the Wasm-precheck version of Wasmtime. In the manually ported version, we leave the Emscripten runtime unchanged, but do modify the generated functions for each benchmark. For each benchmark, we annotate two of the functions: one which initialized the data and one which performed the benchmark computation. In addition to adding type annotations, we add an explicit dynamic check to the top of each benchmark function, which is necessary to type check the dynamically allocated data. The type system tracks constraints from this explicit dynamic check, and is able to use this one check to eliminate many checks.

For all benchmarks, we used Wasmtime in ahead-of-time (AOT) mode: first pre-compiling benchmarks to `.cwasm` files using `wasmtime compile`, then measuring the run time of executing the pre-compiled file using `wasmtime -allow-precompiled`.

Benchmarks. PolyBenchC focuses on the performance of arithmetic and memory instructions, and was used in the original Wasm work by Haas et al. [2017]. PolyBenchC benchmarks initialize vectors and matrices (represented using arrays), and then compute over these structures. These benchmarks perform many memory and arithmetic operations in tight, often nested, loops. They may benefit more from Wasm-precheck than the average program. However, they are not unrealistic, as we expect some computationally intensive Wasm programs to follow this pattern. For example, the demo image classifier microservice for Dapr/WasmEdge has a similar structure.[6]

Only dynamic memory bounds checks. Our run-time performance evaluation studies only dynamic memory bounds checks. When surveying Wasm code, we found that memory accesses were abundant, while indirect calls and integer division-by-zero checks seldom occurred. Checked integer division was rare, in part, because the predominant datatype was floating point numbers.

We also found that memory access were the most expensive. We prototyped with pathological microbenchmarks, which repeatedly execute instructions with dynamic checks in loop. We found that memory bounds checks had much larger slowdown than dynamic type checks on indirect calls, and measured no overhead on the integer division-by-zero check.

We believe dynamic memory bounds checks are the most expensive because they require the most effort to check in comparison to the cost of the instruction they guard. They require a comparison

[6]https://github.com/second-state/dapr-wasm

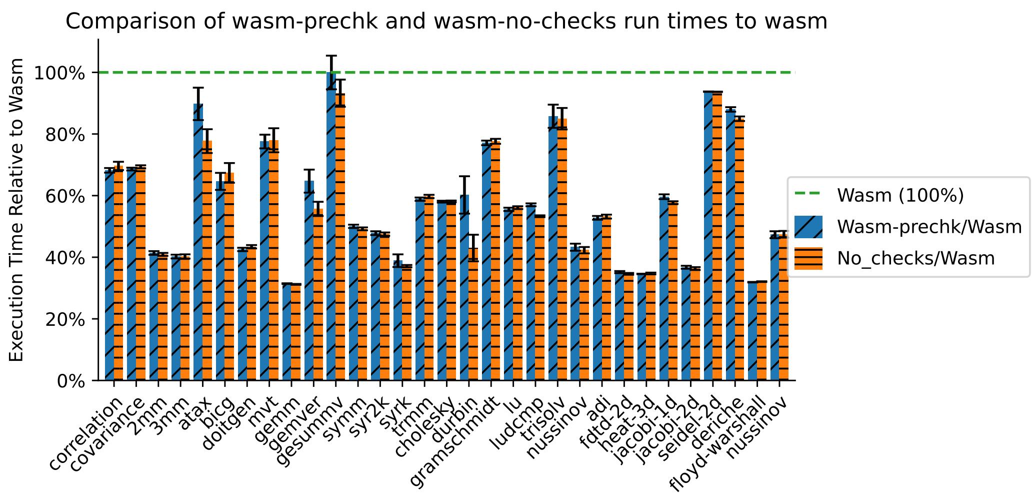

Fig. 6. Comparison of the average run time of PolyBenchC programs; Wasm vs Wasm-no-checks and Wasm vs Wasm-prechk. The error bars show the Standard Error of the Mean.

per operation. The check involves loading the current size of memory and performing an integer check, which we found usually amounts to using an extra register; this agrees with Jangda et al. [2019], who cite increased register pressure due to dynamic checks as a cause of performance issues. While the run-time type check on call_indirect also requires an extra comparison per instruction, it is likely to be insignificant compared to all the computation involved in a function call.

Notably, Jangda et al. [2019] identify dynamic checks on indirect function calls as a significant cost, which disagrees with our findings. There are several possible reasons for this disagreement. First and most importantly is the difference in studying memory bounds checks. Jangda et al. [2019] did not disable virtual memory guard pages to study dynamic memory bounds checks, and therefore would not have seen overhead on such checks. Second, the implementation of indirect function calls in the compiler we study could be more optimized then the version studied by Jangda et al. [2019]. Third, it could be a difference in workflow, where our "pathological" microbenchmark is not in fact the worst case scenario.

Therefore, we focus on eliminating dynamic memory bounds checks in our evaluation, and ignore the other dynamic checks. We emphasize that Wasm-precheck is theoretically capable of eliminating the cost of other dynamic checks, if those costs exist in practice.

Hyperfine. In general, for the benchmarks requiring timing data, we use Hyperfine (`hyperfine`) [Peter 2023], a benchmarking tool that helps to control for noise. Using `hyperfine`, we fix the number of warmups to 3, to ensure the benchmark program was warm in disk cache, and the number of runs to 10, to account for variations in background processes, process randomization, etc.

Machine details. The benchmark machine is a cloud instance running on OpenStack version Ussuri. The instance has 120 GiB of RAM, and 16 vCPUs (Intel Xeon Processor E5-2680 v4). The instance runs Ubuntu 22.04.2-Jammy-x64-2023-02, with Rust 1.68.2 (6feb7c9cf 2023-03-26), PolyBenchC 4.2.1, Wasmtime version v0.30.0, and `wasm-tools` (a subproject used by Wasmtime, containing the type checker) version 1.0.16. The appendix (see Appendix D.2) includes environment variables, which have been shown to affect cache behaviour [Curtsinger and Berger 2013].

6.2 Run-time Performance Analysis

Wasm-no-checks vs Wasm. Compared to Wasm, the unsafe removal of dynamic checks in Wasm-no-checks achieved an average speed-up of 1.77x, up to a maximum of 3.2x (Figure 6). This is a significant increase and demonstrates the value of safely removing dynamic bounds checks. However, how close can we get, safely, with Wasm-precheck?

Wasm-precheck vs Wasm-no-checks and Wasm. Compared to Wasm, the safe removal of dynamic checks in Wasm-precheck led to an average speed-up of 1.72x, up to a maximum of 3.18x (Figure 6). That is about 97% of the speed-up achieved by Wasm-no-checks on average.

Discussion. Wasm-precheck can remove a large percentage of the overhead of dynamic memory bounds checks. This is achieved without attempting to remove every dynamic check. Instead, we focused on loops in computationally-intensive functions. In practice, Wasm-precheck enables the type system to propagate information from dynamic checks outside a loop, so the loops can be free of dynamic checks. The speed-up is achieved in compiled code that includes a memory manager, modified only to include type annotations and an explicit dynamic check at the top of each function.

The PolyBenchC suite are memory intensive programs that access memory in nested loops. Thus, the results of our benchmarks are not generally applicable: it is not fair to expect an arbitrary program to benefit as much as our benchmarks. That said, memory intensive programs are a prominent and useful class of programs, and we can see that Wasm-precheck can safely reap significant performance benefits for such programs. For such programs in environments where memory bounds check through virtual memory guard pages are unavailable, Wasm-precheck can mitigate most of the costs of dynamic checks on memory accesses.

6.3 Type Checking Cost Analysis

For Wasm, fast compilation and a small binary footprint are key design points. Wasm-precheck can achieve significant speed-ups over Wasm, but at the cost of a more complex type checker, additional type annotations, and the possible addition of explicit dynamic checks in the code. To quantify these costs, we measure and compare the compilation time and binary size of our PolyBenchC suites. For this analysis, we only compare the baseline Wasm and the Wasmtime implementations.

6.3.1 Binary Footprint. We compare the size in bytes of the annotated Wasm-precheck program to the size of the unmodified Wasm version. These sizes are of Wasm binary format, not the binary output of the Wasmtime AOT compiler.

Results. On average, the total binary size is about 7.18% larger for Wasm-precheck then Wasm (column 4 of Table 1). We also compare the code and type sections separately (available in the anonymous supplementary material Section D.1). The code sections are barely larger, on average about 0.84%. The type sections were significantly larger, with an average of 642%, as these required explicit pre and postconditions. The smallest Wasm-precheck type section was only about 250% larger than its Wasm counterpart (from 176 to 420 bytes), whereas the largest was a little over 18x as large (from 206 to 3761 bytes).

Discussion. The added footprint of type annotations is relatively small compared to the full file size. The overall increase in size of the final binary is small and is probably worth the improved performance. The overall file size included runtime code added by `emcc`. Recall we only add annotations for two functions. For programs where intensive computation is focused in a few functions, the addition of type annotations for those functions is minor compared to the overall size of the program. However, for a module with less runtime support code compared to code for intensive computation, the increase in binary size may be more significant.

In addition, we found that some programs benefited from reuse between annotations, significantly reducing annotation overhead. For example, in `floyd-warshall` we reuse the annotations between

Benchmark	Compilation Time (s)		Validation Time (s)		Binary Size (bytes)	
	Wasm	prechk	Wasm	prechk	Wasm	prechk
`correlation`	16.49 ± 0.44	17.17 ± 0.20	14.00 ± 0.13	14.37 ± 0.14	20580	22377
`covariance`	16.03 ± 0.36	16.33 ± 0.28	14.17 ± 0.14	14.31 ± 0.22	20381	21556
`2mm`	16.43 ± 0.31	16.73 ± 0.28	14.75 ± 0.24	15.38 ± 0.26	20612	23340
`3mm`	15.91 ± 0.34	16.10 ± 0.30	14.91 ± 0.21	14.96 ± 0.43	20719	24586
`atax`	15.69 ± 0.30	15.17 ± 0.21	15.55 ± 0.21	16.00 ± 0.39	20188	21280
`bicg`	15.41 ± 0.45	15.03 ± 0.37	16.37 ± 0.27	15.40 ± 0.31	20300	21525
`doitgen`	16.09 ± 0.25	15.98 ± 0.42	14.95 ± 0.35	15.06 ± 0.21	20370	22141
`mvt`	15.75 ± 0.22	15.65 ± 0.31	15.69 ± 0.35	14.81 ± 0.12	20397	22021
`gemm`	13.86 ± 0.07	14.10 ± 0.05	13.81 ± 0.08	14.93 ± 0.27	20374	22081
`gemver`	14.12 ± 0.11	14.24 ± 0.09	14.60 ± 0.13	15.38 ± 0.18	20627	24343
`gesummv`	13.78 ± 0.11	13.72 ± 0.10	15.04 ± 0.17	14.80 ± 0.19	20277	21391
`symm`	13.84 ± 0.07	13.94 ± 0.09	15.67 ± 0.38	15.57 ± 0.47	20455	21938
`syr2k`	13.91 ± 0.09	13.85 ± 0.09	16.39 ± 0.37	16.52 ± 0.21	20375	21838
`syrk`	13.94 ± 0.10	14.02 ± 0.11	15.61 ± 0.18	17.44 ± 0.32	20283	21294
`trmm`	13.64 ± 0.10	13.64 ± 0.05	16.34 ± 0.26	15.73 ± 0.21	20242	21084
`cholesky`	14.00 ± 0.25	13.91 ± 0.08	16.44 ± 0.26	17.10 ± 0.30	20520	22003
`durbin`	14.26 ± 0.41	15.95 ± 0.31	16.26 ± 0.34	16.64 ± 0.18	20147	20792
`gramschmidt`	15.71 ± 0.35	16.24 ± 0.30	16.35 ± 0.35	16.38 ± 0.21	20560	21979
`lu`	15.68 ± 0.42	15.60 ± 0.16	17.02 ± 0.31	16.00 ± 0.24	20502	21899
`ludcmp`	16.72 ± 0.47	17.05 ± 0.36	16.18 ± 0.42	16.16 ± 0.20	20823	23533
`trisolv`	15.17 ± 0.17	15.57 ± 0.25	14.91 ± 0.16	15.95 ± 0.41	20113	20950
`deriche`	16.06 ± 0.24	16.72 ± 0.33	16.15 ± 0.41	17.38 ± 0.49	21343	22915
`floyd-warshall`	13.72 ± 0.11	13.88 ± 0.24	13.49 ± 0.19	13.89 ± 0.17	16758	17065
`nussinov`	12.97 ± 0.21	13.62 ± 0.20	14.52 ± 0.27	14.35 ± 0.39	16925	17602
`adi`	15.48 ± 0.17	16.16 ± 0.18	15.22 ± 0.31	16.77 ± 0.32	20725	22500
`fdtd-2d`	16.22 ± 0.27	16.43 ± 0.23	16.81 ± 0.31	16.41 ± 0.37	20802	22972
`heat-3d`	16.96 ± 0.29	15.77 ± 0.32	15.92 ± 0.36	15.33 ± 0.20	20639	21578
`jacobi-1d`	15.42 ± 0.13	16.28 ± 0.37	15.43 ± 0.28	15.75 ± 0.32	20083	20568
`jacobi-2d`	16.31 ± 0.38	15.48 ± 0.14	15.25 ± 0.31	15.42 ± 0.27	20332	20963
`seidel-2d`	16.18 ± 0.35	16.23 ± 0.32	15.05 ± 0.23	15.35 ± 0.17	20205	20608
Average	15.19 ± 0.25	15.35 ± 0.22	15.43 ± 0.27	15.65 ± 0.27	20222	21691

Table 1. Comparison of compile time, validation time, and binary size Wasm vs Wasm-precheck.

the two functions, as they had extremely similar structures. This reuse in type annotations led to a much smaller than average increase in overall code size of 1.83%.

While the overhead of the type annotations may seem large, the encoding of annotations in our implementation of Wasm-precheck is unoptimized. We believe that we can improve the encoding and remove unnecessary annotations to reduce the size. Alternatively, annotations as a whole could be omitted in favor of type inference/static analysis. We discuss these possibilities in Section 8.

6.3.2 Type Checking and Compile Time. We separately compare compile time and type-checking time. Type-checking time is measured using `wasm-tools validate` on the file in Wasm text. Compile time is measured using `wasmtime compile` on the file in Wasm text. We expect type-checking time to dominate the additional increased cost of compilation, but other factors, such as reading or compiling the larger binary, could increase compile time separately from type-checking time. A significant part of the cost in the Wasm-precheck type checker is the constraint checker, so these measurements are tied to the performance of our implementation with Z3.

Results. We found that the overhead in type-checking was relatively small, with Wasm-precheck taking an average of 1.4%, or 220ms, longer (Table 1).

Unexpectedly, this overhead did not seem to be larger for programs with more type annotations. For example, `3mm` had above average binary size overhead in Wasm-precheck vs Wasm, but only took an average of 47ms longer to type check with Wasm-precheck then Wasm, a hardly significant difference since the standard error of the mean was approximately 400ms with Wasm-precheck and 200ms with Wasm. By contrast, `syrk`, with only 1100 lines of annotations (a bit below average), had the most type-checking overhead at 1.8s, or 12%.

The overhead in compilation time was slightly smaller still, with Wasm-precheck taking an average of 1%, or 160ms, longer to compile the benchmarks (Table 1). Interestingly, the compilation time is slightly lower than the type-checking time for both Wasm-precheck and Wasm on average. There could be difference in performance of the frontend between `wasm-tools` and `wasmtime`, which use the same backend typechecker, but the difference is witin margin of error.

Discussion. The overhead of type checking and compiling is small, and probably worth the run-time performance improvement. This may be related to the simple structure of the benchmarks, with tight loops and simple constraints in the annotations keeping the constraint solving queries small and simple. However, if large or complex constraints sets caused significant compile-time overhead, a faster special purpose solver might provide less overhead, and this is supported by both our theory and implementation.

7 RELATED WORK

Using types to improve static reasoning of low-level and compiler intermediate languages is not a new idea. Tarditi et al. [1996] used strongly typed intermediate languages (TIL) to enable optimization of SML code. Compiling SML involves many translations among intermediate languages, and by preserving type information across those translations Tarditi et al. [1996] were able to safely perform additional compiler optimizations. Using TIL led to up to 50% faster programs.

Morrisett et al. [1999] demonstrated how to preserve types through five representative compilation passes to get from System F (a model of a high-level functional language) to a typed assembly language (TAL). The focus of TAL was on safety. Morrisett et al. [1999] demonstrated that untrusted code could be safely executed, so long as it was well typed and type checked first. Although Morrisett et al. [1999] argued that the type-preserving compilation passes would permit similar optimizations to TIL, they didn't include further optimizations based on TAL.

The most closely related work is Xi and Harper [2001], which developed an indexed type system for an assembly language, DTAL, that enabled static guarantees and optimizations such as safely removing 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 was intended to be a target for supporting type-preserving compilation from Dependent ML (an indexed typed SML) and SML. DTAL is a register machine language, and the type system focuses on the flow of constraints between registers and memory. By contrast, Wasm is a stack-based language, so Wasm-precheck focuses on stack-based reasoning. Wasm also includes more structured control flow operations, which pose some unique challenges. One of the major static reasoning hurdles in Wasm, the call_indirect instruction, is not present in DTAL, and the ability to reason about a call_indirect-like instruction statically is novel to Wasm-precheck.

An alternative to typed intermediate languages is proof-carrying code (PCC), which uses a logical framework over low-level code to statically prove safety properties [Necula 1997]. While typed intermediate languages require types as part of the language, PCC uses a separate logical framework, allowing more flexibility to use the approach with an existing language. A PCC approach to removing dynamic safety checks from Wasm would still require Wasm to be extended with instructions that lack dynamic checks, and would otherwise be quite similar.

Like the Wasm-precheck type system, Liquid Types are able to ensure safety properties and eliminate dynamic checks, but unlike Wasm-precheck use sophisticated inference to reduce the annotation burden. Liquid Types were introduced as an indexed type system in OCaml, focused on combining strong static reasoning about program variables with low developer effort [Kawaguchi et al. 2009]. Like Wasm, OCaml already had a strong static type system, but Liquid Types allowed the efficient verification of a large set of libraries with low developer annotations. Since their introduction, Liquid Types have been applied to many languages, including Haskell, Ruby, C, JavaScript [Chugh et al. 2012; Kazerounian et al. 2018; Rondon et al. 2012, 2010; Vazou et al. 2014a,b; Vekris et al. 2016]. It is possible that the Wasm-precheck type system could be converted to a Liquid Type system, removing the need for annotations in the implementation of Wasm-precheck.

An alternate approach to the safety/performance tradeoff by Popescu et al. [2021] starts with unsafe Rust code, and attempts to make it safer while maintaining the performance. They introduce a tool which identifies code that has explicitly omitted dynamic safety checks, and reintroduce such checks until a specified performance overhead threshold is met. In this way, more expensive dynamic checks are less likely to be added. This approach allows library users to have more fine-grained control over the safety/performance tradeoff that would normally be decided by the library developer. However, safety is not maintained using this approach.

8 DISCUSSION AND FUTURE WORK

We briefly discuss limitations in the current model and implementation of Wasm-precheck and how they may be addressed in future work.

Table Mutation. Wasm-precheck assumes that function tables are immutable by a Wasm module, as they were in the original specification [Haas et al. 2017]. However, the most recent Wasm specification now supports table mutation [Rossberg 2022][7]. Furthermore, the table could always be mutated by the host environment (*e.g.*,, using the JavaScript API in a browser). Table mutation violates static guarantees for prechkeced indirect function calls (call_indirect✓), and functions with incompatible types may be called. This is a limitation in safety of the Wasm-precheck model, although not of our implementation, since call_indirect✓ is disabled.

The most straightforward solution is to introduce a separate immutable table, and only allow call_indirect✓ with the immutable table. Attempting to use mutating instructions with an immutable table could result in a static type error. Alternatively, instead of a static type error, every call_indirect✓ on the mutable table could be transparently downgraded to a call_indirect instead, invalidating the optimizations from Wasm-precheck, but allowing the safe execution of the code. This approach allows fine-grained control over the table, providing additional static guarantees. The host environment would be required to respect immutable tables.

Another approach is to modify the implementation of Wasm-precheck rather than specification. JIT implementations could re-type-check programs when the table is mutated, either by instructions or by the host environment. If the check fails, affected call_indirect✓s on that table could be downgraded to a call_indirect. This idea is simpler for developers and requires no language redesign, but could be intractable if table mutation occurs often. If table mutation is infrequent (*e.g.,* only at the beginning of execution for dynamic linking), then this strategy could produce good results.

Global Variables. We do not support constraints on global variables because we cannot compositionally track constraints across module boundaries. This is a limitation in expressivity, but not in safety. 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. Concretely, imagine that the jth module calls a function f_i that was imported from the ith module.

[7]This was added in the reference types proposal https://github.com/WebAssembly/reference-types.

The call instruction is reduced to call $\{\text{inst } i,\ \text{func } f_i\}$ where i is the index for the module instance where f_i is defined. f_i cannot modify the global variables in the jth module directly. However, f_i may call a function imported from jth module that modifies the globals in the jth module. We have to assume the worst and can make no assumptions about the global variables after f_i returns.

We might address this limitation with an effect system to track how functions modify global variables. However, this could be undesirable or difficult to accomplish if global variables should not be exposed as part of an interface.

Dynamic Resizing of Memory. Wasm-precheck only supports type checking ✓-tagged loads and stores based on the static size of memory, but memory can grow monotonically via grow_memory. This is a limitation in expressivity, but not in safety. It should be possible to statically reason about the dynamic size of memory by tracking 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 is equal to the new memory size. For example, we could introduce an index variable α_m to track the size of memory, and after a grow_memory, constrain the size of memory to be $\alpha_m = \mathit{oldsize}$ if the result is -1, and $\alpha_m = \mathit{newsize}$ otherwise. Then, in load✓ and store✓, the bounds check would be performed against α_m. This would likely require passing the size of memory in the instruction type rather than the module environment.

Support for Streaming Compilation. Although not an explicit goal, Wasm-precheck, like Wasm, should support streaming execution. All type annotations are declared before the code section, and this information is propagated forward during type checking (computing the strongest postcondition, rather than reasoning backwards to compute the weakest precondition). Each instruction is checked, and its constraints solved, before checking the next instruction.

This relies on sufficient type annotations before execution, so it may be difficult to combine with type inference. However, if steaming compilation is needed with type inference, one option is to use the Wasm type system as a fast first compiler, and then run the Wasm-precheck type checker in the background to eliminate dynamic checks where possible. This would mirror Firefox's current Wasm implementation, which provides a straightforward fast compiler to begin execution as soon as possible, and a slower, optimizing compiler whose result is used once it finishes compilation.

Type Annotations. There is significant room for improvement in reducing the size of the annotations in the implementation. Currently, each part of the index language is simply encoded into binary, without any optimization, compression, sharing, or eliding parts that could be trivially and locally inferred. Some space savings could be obtained by extending the binary format to explicitly encode common forms, saving some bytes. Additionally, there is significant constraint reuse, as nested blocks usually reuse the preconditions of the outer blocks with some additions. Most postconditions simply specify that locals remain unchanged, even though no instructions inside the annotated block can mutate them. This situation should be easily detectable by the typechecker, which could allow omitting such constraints for local variables which are not modified.

Type Inference. Given the relatively straightforward constraints in Wasm-precheck, type inference may be sufficiently effective to improve performance Wasm programs without developer effort. For Wasm-precheck's type system, this amounts to performing static analysis over Wasm that approximates Wasm-precheck's type system and outputs relevant type annotations. Wasm-precheck's index language encodes a logic that corresponds to a straightforward data flow analysis, so implementing an optimizing embedding should not be difficult using standard analysis techniques Flanagan and Leino [2001]. Such techniques have been widely adapted to indexed and refinement type systems, *e.g.*, see Rondon et al. [2010] or Jhala and Vazou [2020] (Section 5). Wasm-precheck provides correctness guarantees about any such analysis via type safety.

Alternative Constraint Solvers. Wasm-precheck is parametric over the definition of implication, allowing us to use different constraint solvers with different tradeoffs between effectiveness and efficiency. In our prototype implementation, we used Z3, which works well in practice but not in theory. We conjecture that the octagonal abstract domain is sufficient for constraint satisfaction for most our benchmarks [Mine 2001]. The octagonal abstract domain has a polynomial worst case complexity, compared to the current exponential worst case complexity of arbitrary Z3 queries.

9 CONCLUSION

We introduce Wasm-precheck, a low-level language that uses an indexed type system to improve static guarantees and therefore performance of Wasm code. To ensure the safety of Wasm-precheck, we have proven the type safety of Wasm-precheck as well as showing backwards compatibility with Wasm through a sound type erasure to Wasm and automatic embedding from Wasm to Wasm-precheck. We implement Wasm-precheck in an extension of Wasmtime, and achieved an average performance gain of 1.71x over Wasm in the widely used PolyBenchC benchmark suite. This demonstrates our hypothesis that Wasm can be equipped with a type system that, by improving static guarantees to remove unnecessary dynamic checks, can be used to improve performance while maintaining safety.

REFERENCES

Bytecode Alliance. 2019. Wasmtime: A fast and secure runtime for WebAssembly. https://wasmtime.dev/. Accessed: 2023-06-29.

Ravi Chugh, David Herman, and Ranjit Jhala. 2012. Dependent types for JavaScript. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. https://doi.org/10.1145/2384616.2384659

Charlie Curtsinger and Emery D. Berger. 2013. STABILIZER: statistically sound performance evaluation. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/2451116.2451141

Leonardo De Moura and Nikolaj Bjørner. 2008. 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)*. https://dl.acm.org/doi/10.5555/1792734.1792766

Emscripten Contributors. 2015. emscripten. https://emscripten.org/. Accessed: 2023-06-29.

Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. 2009. *Semantics engineering with PLT Redex*. https://redex.racket-lang.org/

Cormac Flanagan and K. Rustan M. Leino. 2001. Houdini, an Annotation Assistant for ESC/Java. In *International Symposium of Formal Methods Europe (FME): Formal Methods for Increasing Software Productivity*. https://doi.org/10.5555/647540.730008

Samuel Ginzburg, Mohammad Shahrad, and Michael J. Freedman. 2023. VectorVisor: A Binary Translation Scheme for Throughput-Oriented GPU Acceleration. In *USENIX Annual Technical Conference (USENIX ATC)*. https://www.usenix.org/conference/atc23/presentation/ginzburg

Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and J. F. Bastien. 2017. Bringing the web up to speed with WebAssembly. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/3062341.3062363

Abhinav Jangda, Bobby Powers, Emery Berger, and Arjun Guha. 2019. Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code. *login Usenix Mag.* 44, 3 (2019). https://www.usenix.org/publications/login/fall2019/jangda

Ranjit Jhala and Niki Vazou. 2020. Refinement Types: A Tutorial. (2020). arXiv:2010.07763 [cs.PL] https://arxiv.org/abs/2010.07763

Ming Kawaguchi, Patrick Maxim Rondon, and Ranjit Jhala. 2009. Type-based data structure verification. In *International Conference on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/1542476.1542510

Milod Kazerounian, Niki Vazou, Austin Bourgerie, Jeffrey S. Foster, and Emina Torlak. 2018. Refinement Types for Ruby. In *Verification, Model Checking, and Abstract Interpretation - 19th International Conference, VMCAI 2018, Los Angeles, CA, USA, January 7-9, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10747)*, Isil Dillig and Jens Palsberg (Eds.). Springer, 269–290. https://doi.org/10.1007/978-3-319-73721-8_13

A. Mine. 2001. The octagon abstract domain. https://doi.org/10.1109/WCRE.2001.957836

J. Gregory Morrisett, David Walker, Karl Crary, and Neal Glew. 1999. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 21, 3 (1999). https://doi.org/10.1145/319301.319345

George C. Necula. 1997. Proof-Carrying Code. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/263699.263712

David Peter. 2023. *hyperfine*. https://github.com/sharkdp/hyperfine

Natalie Popescu, Ziyang Xu, Sotiris Apostolakis, David I. August, and Amit Levy. 2021. Safer at Any Speed: Automatic Context-Aware Safety Enhancement for Rust. *Proc. ACM Program. Lang.* 5, OOPSLA (oct 2021). https://doi.org/10.1145/3485480

Louis-Noel Pouchet and Tomofumi Yuki. 2016. PolyBench/C: The Polyhedral benchmark suite, v4.2.1. https://sourceforge.net/projects/polybench/files/polybench-c-4.2.1-beta.tar.gz/download. Accessed: 2023-06-29.

Patrick Rondon, Alexander Bakst, Ming Kawaguchi, and Ranjit Jhala. 2012. CSolve: Verifying C with liquid types. In *Computer Aided Verification (CAV)*. Springer, 744–750. https://doi.org/10.1007/978-3-642-31424-7_59

Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. 2010. Low-level liquid types. In *Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/1706299.1706316

Andreas Rossberg. 2022. WebAssembly Core Specification. https://www.w3.org/TR/wasm-core-2/ https://webassembly.github.io/spec/core/_download/WebAssembly.pdf.

David Tarditi, J. Gregory Morrisett, Perry Cheng, Christopher A. Stone, Robert Harper, and Peter Lee. 1996. TIL: A Type-Directed Optimizing Compiler for ML. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/231379.231414

Niki Vazou, Eric L. Seidel, and Ranjit Jhala. 2014a. LiquidHaskell: Experience with Refinement Types in the Real World. In *Symposium on Haskell*. https://doi.org/10.1145/2633357.2633366

Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon L. Peyton Jones. 2014b. Refinement types for Haskell. In *International Conference on Functional Programming (ICFP)*. https://doi.org/10.1145/2628136.2628161

Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. 2016. Refinement Types for TypeScript. In *International Conference on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/2908080.2908110

Conrad Watt. 2018. Mechanising and verifying the WebAssembly specification. In *Proceedings of the ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*. https://doi.org/10.1145/3167082

Hongwei Xi and Robert Harper. 2001. A Dependently Typed Assembly Language. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*. https://doi.org/10.1145/507635.507657

Drew Zagieboylo, G. Edward Suh, and Andrew C. Myers. 2020. The Cost of Software-Based Memory Management Without Virtual Memory. *CoRR* abs/2009.06789 (2020). arXiv:2009.06789 https://arxiv.org/abs/2009.06789

Christoph Zenger. 1997. Indexed types. *Theoretical Computer Science* 187, 1 (1997). https://doi.org/10.1016/S0304-3975(97)00062-5

A COMPLETE WASM-PRECHECK TYPING JUDGMENT DEFINITION

$$\boxed{C \vdash e^* : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2}$$

$$\textsc{Unreachable}\quad \frac{}{C \vdash \mathsf{unreachable} : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2} \qquad \textsc{Nop}\quad \frac{}{C \vdash \mathsf{nop} : \epsilon; l; \Gamma; \phi \longrightarrow \epsilon; l; \Gamma; \phi}$$

$$\textsc{Drop}\quad \frac{}{C \vdash \mathsf{drop} : (t\ \alpha); l; \Gamma; \phi \longrightarrow \epsilon; l; \Gamma; \phi}$$

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

$$\frac{\alpha_3 \notin \Gamma}{C \vdash t.binop : (t\ \alpha_1)\ (t\ \alpha_2); l; \Gamma; \phi \longrightarrow (t\ \alpha_3); l; \Gamma, (t\ \alpha_3); \phi, (=\alpha_3\ (\|binop\|\ \alpha_1\ \alpha_2))}\ \textsc{Binop}$$

$$\textsc{Div-Prechk}\quad \frac{\Gamma \vdash \phi \rightsquigarrow \neg(=\ \alpha_2\ 0) \qquad \alpha_3 \notin \Gamma}{C \vdash t.\mathsf{div}✓ : (t\ \alpha_1)\ (t\ \alpha_2); l; \Gamma; \phi \longrightarrow (t\ \alpha_3); l; \Gamma, (t\ \alpha_3); \phi, (=\alpha_3\ (\mathsf{i32.div}\ \alpha_1\ \alpha_2))}$$

$$\frac{\alpha_3 \notin \Gamma}{C \vdash t.relop : (t\ \alpha_1)\ (t\ \alpha_2); l; \Gamma; \phi \longrightarrow (t\ \alpha_3); l; \Gamma, (t\ \alpha_3); \phi, (=\alpha_3\ (\|relop\|\ \alpha_1\ \alpha_2))}\ \textsc{Relop}$$

$$\frac{\alpha_2 \notin \Gamma}{C \vdash t.testop : (t\ \alpha_1); l; \Gamma; \phi \longrightarrow (t\ \alpha_2); l; \Gamma, (t\ \alpha_2); \phi, (=\alpha_2\ (\|testop\|\ \alpha_1))}\ \textsc{Testop}$$

$$\frac{\alpha_2 \notin \Gamma}{C \vdash t.unop : (t\ \alpha_1); l; \Gamma; \phi \longrightarrow (t\ \alpha_2); l; \Gamma, (t\ \alpha_2); \phi, (=\alpha_2\ (\|unop\|\ \alpha_1))}\ \textsc{Unop}$$

$$\frac{\alpha \notin \Gamma}{C \vdash \mathsf{select} : (t\ \alpha_1)\ (t\ \alpha_2)\ (\mathsf{i32}\ \alpha_3); l; \Gamma, \phi \longrightarrow (t\ \alpha); l; \Gamma, (t\ \alpha); \phi, (\text{if } (=\alpha_3\ (\mathsf{i32}\ 0))\ (=\alpha\ \alpha_2)\ (=\alpha\ \alpha_1))}\ \textsc{Select}$$

$$\textsc{Block}\quad \frac{\begin{array}{c} C, \text{label } ((t_2\ \alpha_2)^*; (t_l\ \alpha_{l2})^*; \phi_2) \vdash e^* : (t_1\ \alpha_1)^*; (t_l\ \alpha_{l1})^*; \Gamma_1; \phi_1 \longrightarrow (t_2\ \alpha_2)^*; (t_l\ \alpha_{l2})^*; \Gamma_2; \phi_3 \\ \Gamma_2 \vdash \phi_3 \rightsquigarrow \phi_2 \end{array}}{C \vdash \mathsf{block}\ (t_1^* \longrightarrow t_2^*)\ e^*\ \mathsf{end} : (t_1\ \alpha_1)^*; (t_l\ \alpha_{l1}); \Gamma_1; \phi_1 \longrightarrow (t_2\ \alpha_2)^*; (t_l\ \alpha_{l2})^*; \Gamma_2; \phi_2}$$

$$\textsc{Loop}\quad \frac{\begin{array}{c} C, \text{label } ((t_1\ \alpha_1)^*; (t_l\ \alpha_{l1})^*; \phi_3) \vdash e_1^* : (t_1\ \alpha_1)^*; (t_l\ \alpha_{l1})^*; \Gamma_1\ (t_1\ \alpha_1)^*\ (t_l\ \alpha_{l1})^*; \phi_3 \longrightarrow (t_2\ \alpha_2)^*; (t_l\ \alpha_{l2})^*; \Gamma_2; \phi_4 \\ \Gamma_1 \vdash \phi_1 \rightsquigarrow \phi_3 \qquad \Gamma_2 \vdash \phi_4 \rightsquigarrow \phi_2 \end{array}}{C \vdash \mathsf{loop}\ t_1^* \longrightarrow t_2^*\ e^*\ \mathsf{end} : (t_1\ \alpha_1)^*; (t_l\ \alpha_{l1}); \Gamma_1; \phi_1 \longrightarrow (t_2\ \alpha_2)^*; (t_l\ \alpha_{l2})^*; \Gamma_2; \phi_2}$$

$$\frac{\begin{array}{c} C, \text{label } ((t_2\ \alpha_2)^*; (t_l\ \alpha_{l2})^*; \phi_2) \vdash e_1^* : (t_1\ \alpha_1)^*; (t_l\ \alpha_{l1})^*; \Gamma_1; \phi_1, \neg(=\alpha\ (\mathsf{i32}\ 0)) \longrightarrow (t_2\ \alpha_2)^*; (t_l\ \alpha_{l2})^*; \Gamma_2; \phi_3 \\ C, \text{label } ((t_2\ \alpha_2)^*; (t_l\ \alpha_{l2})^*; \phi_2) \vdash e_2^* : (t_1\ \alpha_1)^*; (t_l\ \alpha_{l1})^*; \Gamma_1; \phi_1, (=\alpha\ (\mathsf{i32}\ 0)) \longrightarrow (t_2\ \alpha_2)^*; (t_l\ \alpha_{l2})^*; \Gamma_2; \phi_4 \\ \Gamma_2 \vdash \phi_3 \rightsquigarrow \phi_2 \qquad \Gamma_2 \vdash \phi_4 \rightsquigarrow \phi_2 \end{array}}{C \vdash \mathsf{if}\ t_1^* \longrightarrow t_2^*\ e_1^*\ \mathsf{else}\ e_2^*\ \mathsf{end} : (\mathsf{i32}\ \alpha)\ (t_1\ \alpha_1)^*; (t_l\ \alpha_{l1}); \Gamma_1; \phi_1 \longrightarrow (t_2\ \alpha_2)^*; (t_l\ \alpha_{l2})^*; \Gamma_2; \phi_2}\ \textsc{If}$$

$$\frac{C_{\text{return}} = (t_3\ \alpha_4)^*; \phi_3 \qquad \Gamma_1 \vdash \phi_1 \rightsquigarrow \phi_3[\alpha_4 \mapsto \alpha_3]^*}{C \vdash \mathsf{return} : ti_1^*\ ((t_3\ \alpha_3)^*)\ l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2}\ \textsc{Return}$$

$$\frac{C_{\text{label}}(i) = (t_3\ \alpha_4)^*; (t_l\ \alpha_{l4})^*; \phi_3 \qquad \Gamma_1 \vdash \phi_1 \rightsquigarrow \phi_3[\alpha_4 \mapsto \alpha_3]^*[\alpha_{l4} \mapsto \alpha_{l3}]^*}{C \vdash \mathsf{br}\ i : ti_1^*\ (t_3\ \alpha_3)^*; (t_l\ \alpha_{l3})^*; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2}\ \textsc{Br}$$

$$\frac{C_{\text{label}}(i) = (t_1\ \alpha_3)^*;\ (t_l\ \alpha_{l3})^*;\ \phi_3 \qquad \Gamma_1 \vdash \phi_1, \neg(=\alpha\ (\text{i32}\ 0)) \rightsquigarrow \phi_3[\alpha_3 \mapsto \alpha_1]^*[\alpha_{l3} \mapsto \alpha_{l1}]^*}{C \vdash \text{br\_if}\ i : (t_1\ \alpha_1)^*;\ (t_l\ \alpha_{l1})^*;\ \Gamma_1;\ \phi_1 \longrightarrow (t_1\ \alpha_1)^*;\ (t_l\ \alpha_{l1})^*;\ \Gamma_1;\ \phi_1, (=\alpha\ (\text{i32}\ 0))}\ \textsc{Br-If}$$

$$\frac{(C_{\text{label}}(i) = (t_1\ \alpha_i)^*;\ (t_l\ \alpha_{li})^*;\ \phi_i)^+ \qquad (\Gamma_1 \vdash \phi_1 \rightsquigarrow \phi_i[\alpha_i \mapsto \alpha_1]^*[\alpha_{li} \mapsto \alpha_{l1}]^*)^*}{C \vdash \text{br\_table}\ i^+ : (t_1\ \alpha_1)^*\ (\text{i32}\ \alpha);\ (t_l\ \alpha_{l1})^*;\ \Gamma_1;\ \phi_1 \longrightarrow ti_2^*;\ l_2;\ \Gamma_2;\ \phi_2}\ \textsc{Br-Table}$$

$$\frac{C_{\text{func}}(i) = (t_1\ \alpha_3)^*;\ \phi_3 \longrightarrow (t_2\ \alpha_4)^*;\ \phi_4 \qquad \Gamma_1 \vdash \phi_1 \rightsquigarrow \phi_3[\alpha_3 \mapsto \alpha_1]^* \qquad (t_2\ \alpha_4)^* \notin \Gamma_1}{C \vdash \text{call}\ i : (t_1\ \alpha_1)^*;\ l;\ \Gamma_1;\ \phi_1 \longrightarrow (t_2\ \alpha_4)^*;\ l;\ \Gamma_1, (t_2\ \alpha_4)^*;\ \phi_1 \cup \phi_4[\alpha_3 \mapsto \alpha_1]^*}\ \textsc{Call}$$

$$\frac{C_{\text{table}} = (j, tfi^*) \qquad \Gamma_1 \vdash \phi_1 \rightsquigarrow \phi_3[\alpha_3 \mapsto \alpha_1]^* \qquad (t_2\ \alpha_4)^* \notin \Gamma_1}{\begin{array}{l} C \vdash\ \text{call\_indirect}\ ((t_1\ \alpha_3)^*;\ \phi_3 \longrightarrow (t_2\ \alpha_4)^*;\ \phi_4) \\ \quad :\ (t_1\ \alpha_1)^*\ (\text{i32}\ \alpha);\ l;\ \Gamma_1;\ \phi_1 \\ \qquad \longrightarrow (t_2\ \alpha_4)^*;\ l;\ (\Gamma_1\ (t_2\ \alpha_4))^*;\ \phi_1 \cup \phi_4[\alpha_3 \mapsto \alpha_1]^* \end{array}}\ \textsc{Call-Indirect}$$

$\textsc{Call-Indirect-Prechk}$

$$\frac{\begin{array}{c} C_{table}(i) = (n, tfi^*) \qquad C_{\text{table}} = (j, tfi^*) \qquad \Gamma_1 \vdash \phi_1 \rightsquigarrow \phi_3[\alpha_3 \mapsto \alpha_1]^* \\ (t_2\ \alpha_4)^* \notin \Gamma_1 \forall i \le n.\ (\Gamma_1 \vdash \phi_1 \rightsquigarrow \neg(=(\text{i32}\ i)\ a))\ \vee\ (tfi^*)(i) = ((t_1\ \alpha_3)^*;\ \phi_3 \longrightarrow (t_2\ \alpha_4)^*;\ \phi_4) \end{array}}{\begin{array}{l} C \vdash\ \text{call\_indirect}\checkmark\ ((t_1\ \alpha_3)^*;\ \phi_3 \longrightarrow (t_2\ \alpha_4)^*;\ \phi_4) \\ \quad :\ (t_1\ \alpha_1)^*\ (\text{i32}\ \alpha);\ l;\ \Gamma_1;\ \phi_1 \\ \qquad \longrightarrow (t_2\ \alpha_4)^*;\ l;\ \Gamma_1, (t_2\ \alpha_4)^*;\ \phi_1 \cup \phi_4[\alpha_3 \mapsto \alpha_1]^* \end{array}}$$

$$\frac{C_{\text{local}}(i) = t \qquad l(i) = (t\ \alpha) \qquad \alpha_2 \notin \Gamma}{C \vdash \text{get\_local}\ i : \epsilon;\ l;\ \Gamma;\ \phi \longrightarrow (t\ \alpha_2);\ l;\ \Gamma, (t\ \alpha_2);\ \phi, (=\alpha\ \alpha_2)}\ \textsc{Get-Local}$$

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

$$\frac{C_{\text{local}}(i) = t \qquad l_2 = l_1[i := (t\ \alpha)] \qquad \alpha_2 \notin \Gamma}{C \vdash \text{tee\_local}\ i : (t\ \alpha);\ l_1;\ \Gamma;\ \phi \longrightarrow (t\ \alpha_2);\ l_2;\ \Gamma, (t\ \alpha_2);\ \phi, (=\alpha\ \alpha_2)}\ \textsc{Tee-Local}$$

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

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

$$\frac{C_{\text{memory}} = n \qquad 2^a \le (|tp|\ <)^?|t| \qquad \alpha_2 \notin \Gamma}{C \vdash t.\text{load}\ (tp\_sx)^?\ a\ o : (\text{i32}\ \alpha_1);\ l;\ \Gamma;\ \phi \longrightarrow (t\ \alpha_2);\ l;\ \Gamma, (t\ \alpha_2);\ \phi}\ \textsc{Mem-Load}$$

$\textsc{Load-Prechk}$

$$\frac{C_{\text{memory}} = n \qquad 2^a \le (|tp|\ <)^?|t| \qquad \alpha_3 \notin \Gamma \qquad \Gamma \vdash \phi \rightsquigarrow (\text{le}\ (\text{add}\ \alpha_1\ (\text{i32}\ o + width))\ (\text{i32}\ n * 64\text{Ki}))}{C \vdash t.\text{load}\checkmark\ (tp\_sx)^?\ a\ o : (\text{i32}\ \alpha_1);\ l;\ \Gamma;\ \phi \longrightarrow (t\ \alpha_2);\ l;\ \Gamma, (t\ \alpha_2);\ \phi}$$

$$\frac{C_{\text{memory}} = n \qquad 2^a \le (|tp|\ <)^?|t|}{C \vdash t.\text{store}\ tp^?\ a\ o : (\text{i32}\ \alpha_1)\ (t\ \alpha_2);\ l;\ \Gamma;\ \phi \longrightarrow \epsilon;\ l;\ \Gamma;\ \phi}\ \textsc{Mem-Store}$$

$\textsc{Store-Prechk}$

$$\frac{C_{\text{memory}} = n \qquad 2^a \le (|tp|\ <)^?|t| \qquad \Gamma \vdash \phi \rightsquigarrow (\text{le}\ (\text{add}\ \alpha_1\ (\text{i32}\ o + width))\ (\text{i32}\ n * 64\text{Ki}))}{C \vdash t.\text{store}\checkmark\ tp^?\ a\ o : (\text{i32}\ \alpha_1)\ (t\ \alpha_2);\ l;\ \Gamma;\ \phi \longrightarrow \epsilon;\ l;\ \Gamma;\ \phi}$$

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

$$\frac{C_{\text{memory}} = n \qquad \alpha_2 \notin \Gamma}{C \vdash \text{grow\_memory} : (\text{i32}\ \alpha_1);\ l;\ \Gamma;\ \phi \rightarrow (\text{i32}\ \alpha_2);\ l;\ \Gamma, (\text{i32}\ \alpha_2);\ \phi}\ \textsc{Grow-Memory}$$

$$\frac{C \vdash e^* : ti_1^*;\ l_1;\ \Gamma_1;\ \phi_1 \rightarrow ti_2^*;\ l_2;\ \Gamma_2;\ \phi_2}{C \vdash e^* : ti^*\ ti_1^*;\ l_1;\ \Gamma_1;\ \phi_1 \rightarrow ti^*\ ti_2^*;\ l_2;\ \Gamma_2;\ \phi_2}\ \textsc{Stack-Poly}$$

$$\textsc{Empty}\quad \frac{}{C \vdash \epsilon : \epsilon;\ l;\ \Gamma;\ \phi \rightarrow \epsilon;\ l;\ \Gamma;\ \phi}$$

$$\frac{C \vdash e_1^* : ti_1^*;\ l_1;\ \Gamma_1;\ \phi_1 \rightarrow ti_2^*;\ l_2;\ \Gamma_2;\ \phi_2 \qquad C \vdash e_2 : ti_2^*;\ l_2;\ \Gamma_2;\ \phi_2 \rightarrow ti_3^*;\ l_3;\ \Gamma_3;\ \phi_3}{C \vdash e_1^*\ e_2 : ti_1^*;\ l_1;\ \Gamma_1;\ \phi_1 \rightarrow ti_3^*;\ l_3;\ \Gamma_3;\ \phi_3}\ \textsc{Composition}$$

A.1 Module Types

The complete module typing rules are in Figure 7 (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 $ti_2^*;\ l_2;\ \Gamma_2;\ \phi_2$ 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.

A.2 Administrative Typing Rules

While we have shown the Wasm-precheck 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-precheck program), and can only appear as an intermediate term during reduction. Figure 9 shows the Wasm-precheck typing rules for module instances *inst*, the run time store s, and various data structures contained within s. There 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 8) to ensure that a Wasm-precheck 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 9), to ensure that a reducible Wasm-precheck program is well typed. Rule Code checks that a sequence of instructions is well typed with an empty stack, the

$$\frac{\begin{array}{c} C_2 = C, \text{local } t_1^* \; t^*, \text{label } (ti_2^*; l_2; \phi_2), \text{return } (ti_2^*; \phi_2) \\ C_2 \vdash e^* : \epsilon; (t_1 \; \alpha_0)^* \; (t \; \alpha)^*; \emptyset, (t_1 \; \alpha_0)^*, (t \; \alpha)^*; (\phi_1, (= \alpha \; (t \; 0)^*))[\alpha_1 \mapsto \alpha_0] \to (t_2 \; \alpha_3)^*; l_2; \Gamma_3; \phi_3 \\ \vdash (t_1 \; \alpha_1)^*; \phi_1 \to (t_2 \; \alpha_2)^*; \phi_2 \qquad \Gamma_3 \vdash \phi_3 \rightsquigarrow \phi_2[\alpha_2 \mapsto \alpha_3] \end{array}}{C \vdash ex^* \; \text{func } (t_1 \; \alpha_1)^*; \phi_1 \to (t_2 \; \alpha_2)^*; \phi_2 \; \text{local } t^* \; e^* : ex^* \; (t_1 \; \alpha_1)^*; \phi_1 \to (t_2 \; \alpha_2)^*; \phi_2} \; \textsc{Func}$$

$$\frac{}{C \vdash ex^* \; \text{func } tfi \; im : ex^* \; tfi} \; \textsc{Function-Import}$$

$$\frac{tg = mut^? \; t \qquad ex^* = \epsilon \lor tg = t \qquad C \vdash e^* : \epsilon; \epsilon; \emptyset; \emptyset \to (t \; a); \epsilon; \Gamma_2; \phi_2}{C \vdash ex^* \; \text{global } tg \; e^* : ex^* \; tg} \; \textsc{Global}$$

$$\frac{tg = t}{C \vdash ex^* \; \text{global } tg \; im : ex^* \; tg} \; \textsc{Global-Import} \qquad \frac{(C_{\text{func}}(i) = tfi)^n}{C \vdash ex^* \; \text{table } n \; i^n : ex^* \; (n, tfi^n)} \; \textsc{Table}$$

$$\frac{}{C \vdash ex^* \; \text{table } (n, tfi^n) \; im : ex^* \; (n, tfi^n)} \; \textsc{Table-Import} \qquad \frac{}{C \vdash ex^* \; \text{memory } n : ex^* \; n} \; \textsc{Memory}$$

$$\frac{}{C \vdash ex^* \; \text{memory } n \; im : ex^* \; n} \; \textsc{Memory-Import}$$

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

Fig. 7. Indexed Module Typing Rules

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

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

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

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

$$\frac{\begin{array}{c} (\vdash v : (t \; \alpha); \phi_v)^* \qquad (\alpha_2 \notin (t \; \alpha)^*)^* C = S_{\text{inst}}(i), \text{local } t^*, \text{return } (ti^n; \phi)^? \\ S; C \vdash e^* : \epsilon; (t \; \alpha)^*; \emptyset, (t \; \alpha)^*, (t_2 \; \alpha_2)^*; \phi_v^*, (= \alpha_2 \; (t_2 \; c_2))^* \to ti^n; l; \Gamma; \phi \qquad \Gamma \vdash \phi \rightsquigarrow \phi_2 \end{array}}{S; (ti^n; \phi)^? \vdash_i v^*; e^* : ti^n; l; \Gamma; \phi_2} \; \textsc{Code}$$

Fig. 8. Wasm-precheck Program Typing Rules

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 10) 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

$$\boxed{\vdash s : S}$$

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

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

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

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

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

$$\boxed{S \vdash \mathit{cl} : ti_1^*;\ \phi_1 \longrightarrow ti_2^*;\ \phi_2}$$

$$\frac{S_{\text{inst}}(i) \vdash f : ti_1^*;\ \phi_1 \longrightarrow ti_2^*;\ \phi_2}{S \vdash \{\text{inst } i, \text{code } f\} : ti_1^*;\ \phi_1 \longrightarrow ti_2^*;\ \phi_2} \textsc{Closure}$$

Fig. 9. Wasm-precheck Store Typing Rules

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 7 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 10. The administrative typing judgment $S; C \vdash e^* : \mathit{tfi}$ extends the Wasm-precheck typing rules for instructions to include administrative instructions and the store context S. Every rule of the judgment $C \vdash e^* : \mathit{tfi}$ 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 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.

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

$$\frac{S; (ti_2^n; \phi_2) \vdash_i v_l^*; e^* : ti_2^n; l_2; \Gamma_2; \phi_2 \qquad ti_2^n \notin \Gamma_1}{S; C \vdash \mathsf{local}_n\{i; v_l^*\}\ e^*\ \mathsf{end} : \epsilon; l; \Gamma_1; \phi_1 \rightarrow ti_2^n; l; \Gamma_1, ti_2^n; \phi_1 \cup \phi_2}\ \textsc{Local}$$

$$\frac{S \vdash cl : (t_2\ \alpha_2)^*; \phi_2 \rightarrow (t_3\ \alpha_3)^*; \phi_3 \qquad \Gamma_1 \vdash \phi_1 \rightsquigarrow \phi_2[\alpha_2 \mapsto \alpha_1] \qquad (t_3\ \alpha_4)^* \notin \Gamma_1}{S; C \vdash \mathsf{call}\ cl : (t_2\ \alpha_1)^*; l; \Gamma_1; \phi_1 \rightarrow (t_2\ \alpha_2)^*; l; \Gamma_1, (t_3\ \alpha_4)^*; \phi_1 \cup \phi_3[\alpha_2 \mapsto \alpha_1][\alpha_3 \mapsto \alpha_4]}\ \textsc{Call-Cl}$$

$$\frac{}{S; C \vdash \mathsf{trap} : tfi}\ \textsc{Trap}$$

$$\frac{\Gamma_4 \vdash \phi_4 \rightsquigarrow \phi_2 \qquad \begin{array}{c} S; C \vdash e_0^* : ti_3^*; l_3; \Gamma_3; \phi_3 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_4 \qquad \Gamma_3 \supseteq \Gamma_1, ti_3^*, l_3 \\ S; C, \text{label}\ (ti_3^*; l_3; \phi_3) \vdash e^* : \epsilon; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_5 \end{array} \qquad \Gamma_2 \vdash \phi_5 \rightsquigarrow \phi_2}{S; C \vdash \mathsf{label}\{e_0^*\}\ e^*\ \mathsf{end} : \epsilon; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2}\ \textsc{Label}$$

Fig. 10. Wasm-precheck Administrative Instruction Rules

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.

B ERASURE AND EMBEDDING DEFINITIONS AND PROOFS

B.1 Embedding Wasm in Wasm-precheck

We present a way to embed Wasm programs in Wasm-precheck, and prove that it will generate well-typed Wasm-precheck programs when embedding well-typped Wasm programs.

Embedding works purely over the surface syntax of the languages. 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-precheck, 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.

First, we define embedding over modules: the pinnacle syntactic objects of both the Wasm and Wasm-precheck surface syntax hierarchies. Embedding a module *module* means embedding all of the functions f^* in the module. We explain how to embed functions f in Definition 6 below. We do not have to embed globals $glob^*$, the table $tab^?$, or the memory $mem^?$ as Wasm and Wasm-precheck use the same syntax to define them (although Wasm-precheck represents the types of tables differently).

We currently cannot typecheck Wasm imported tables under the Wasm-precheck type system since we do not have access to what the function types are for an imported table, which is necessary for typechecking it in Wasm-precheck. Unfortunately, this is currently a limitation that would require the developer to copy over the annotations.

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

Definition 3. $\boxed{embed_m(m) = m}$

$$embed_m(\mathbf{module}\ f^*\ glob^*\ tab^?\ mem^?) \quad = \quad \mathsf{module}\ embed_f(f)^*\ embed_g(glob)^*\ tab^?\ mem^?$$

We use lemmas that show that that well-typed Wasm global variable and function definitions embed into well-typed global variables and functions, respectively, in Wasm-precheck.

Theorem 4. Well Typed Wasm Programs Embedded in Wasm-precheck are Well Typed
If $\vdash$ **module** $f^* \ glob^* \ tab^? \ mem^?$,
then $\vdash embed_m($**module** $f^* \ glob^* \ tab^? \ mem^?)$

Proof. We must show that every premise of Rule Module holds on the embedding module.

- $(embed_C(C) \vdash embed_f(f) : ex_f^* \ tfi)^*$
  Since $(C \vdash f : ex_f^* \ tfi)^*$ is a premise of $\vdash$ **module** $f^* \ glob^* \ tab^? \ mem^?$, then we have $(embed_C(C) \vdash embed_f(f) : ex_f^* \ tfi)^*$ by Lemma Sound-Embedding-of-Functions.
- $(embed_C(C_i) \vdash embed_g(glob) : ex_g^* \ tg)^*$
  Since $(C \vdash glob : ex_g^* \ tg)^*$ is a premise of $\vdash$ **module** $f^* \ glob^* \ tab^? \ mem^?$, then we have $(embed_C(C) \vdash embed_g(glob) : ex_g^* \ tg)^*$ by Lemma Sound-Embedding-of-Globals.
- $(embed_C(C) \vdash tab : ex_t^* \ (n, tfi^n))^?$
  We proceed by case analysis of $tab^?$:
  - Case ϵ
    This case is vaucuous, so we have nothing to prove.
  - Case (ex_t^*) **table** $n \ i^n$
    We want to show that $embed_C(C) \vdash ex_t^*$ table $n \ i^n : ex_t^* \ (n, tfi^m)$
    To do so, we must show that $(embed_C(C)_{\text{func}}(i) = tfi)^n$.
    We have $(C_{\text{func}}(i) = tf)^n$, as it is a premise of $\vdash$ **module** $f^* \ glob^* \ tab^? \ mem^?$.
    Thus, we have $(embed_C(C)_{\text{func}}(i) = tfi)^n$ by definition of $embed_C$.
  - Case (ex_t^*) **table** $n \ im$
    Unfortunately, we do not currently support embedding imported tables.
- $(embed_C(C) \vdash mem : ex_t^* \ n^?$
  We proceed by case analysis of $mem^?$:
  - Case ϵ
    This case is vaucuous, so we have nothing to prove.
  - Case (ex_m^*) **memory** n
    Trivially, $embed_C(C) \vdash (ex_m^*)$ memory $n : ex_m^* \ n$ by Rule Memory
  - Case (ex_m^*) **memory** $n \ im$
    Trivially, $embed_C(C) \vdash (ex_m^*)$ memory $n \ im : ex_m^* \ n$ by Rule Memory-Import
- $embed_C(C) = \{\text{func } tfi^*, \ \text{global } tg^*, \ \text{table } (n, tfi^n)^?, \ \text{memory } n^?\}$
  We have $C = \{\text{func } tf^*, \ \text{global } tg^*, \ \text{table } n^?, \ \text{memory } n^?\}$, as it is a premise of $\vdash$ **module** $f^* \ glob^* \ tab^? \ mem^?$. Then this holds by definition of $embed_C$.
- $embed_C(C_i) = \{ \text{ global } tg^{i-1}\}$
  We have $C_i = \{ \text{ global } tg^{i-1}\}$, as it is a premise of $\vdash$ **module** $f^* \ glob^* \ tab^? \ mem^?$. Then this holds by definition of $embed_C$.
- $ex_f^*, ex_g^*, ex_t^*, ex_m^*$ distinct
  We know this to be true since it is a premise of $\vdash$ **module** $f^* \ glob^* \ tab^? \ mem^?$.

□

The proof relies on the definition of the embedding of module type contexts, which we provide below.

Definition 4. $\boxed{embed_C(C) = C}$

$$\begin{aligned} embed_C(\{&\text{func } (t_4^* \to t_5^*)^*, &= \{&\text{func } ((t_4 \ \alpha_4)^*; \emptyset \to (t_5 \ \alpha_5)^*; \emptyset)^*,\\ &\text{global } tg^*, \ \text{table } n^?, \ \text{memory } n^?, & &\text{global } tg^*, \ \text{table } (n, tfi^*)^?, \ \text{memory } n^?,\\ &\text{local } t_1^*, \ \text{label } (t_2^*)^*, & &\text{local } t_1^*, \ \text{label } ((t_2 \ \alpha_2)^*; (t_1 \ \alpha_1)^*; \emptyset)^*,\\ &\text{return } (t_3^*)^?\}) & &\text{return } ((t_3 \ \alpha_3)^*; \emptyset)^?\} \end{aligned}$$

The first lemma shows that embedding well-typed Wasm global variable definitions and import declarations results in well-typed Wasm-precheck global variables. Before we show the lemma, we first show the definition of embedding a global variable.

Definition 5. $\boxed{embed_g(glob) = glob}$

$$\begin{aligned} embed_g(ex^* \ \mathbf{global} \ tg \ e^*) &= ex^* \ \mathrm{global} \ tg \ embed_{e^*}(e^*)^\epsilon \\ embed_g(ex^* \ \mathbf{global} \ tg \ im) &= ex^* \ \mathrm{global} \ tg \ im \end{aligned}$$

This proof relies on a lemma about embedded instructions being well typed, Lemma Sound-Embedding-of-Instructions, to ensure that the instructions that initiate the global produce a value of the correct type. This lemma is introduced and defined below, along with the definition of the embedding function for instructions $embed_{e^*}$.

Lemma 6. Sound-Embedding-of-Globals
If $C \vdash glob : ex^* \ tg$, then $embed_C(C) \vdash embed_g(glob) : ex^* \ tg$.

Proof. We proceed by case anaylsis on $C \vdash glob : ex^* \ tg$.

- $C \vdash \mathbf{global} \ tg \ im : ex^* \ tg$
  We want to show that $embed_C(C) \vdash \mathrm{global} \ tg \ im : ex^* \ tg$. To do so, we must show that $tg = t$, which is a premise of $C \vdash \mathbf{global} \ tg \ im : ex^* \ tg$, so we have $embed_C(C) \vdash \mathrm{global} \ tg \ im : ex^* \ tg$.
- $C \vdash \mathbf{global} \ tg \ e^* : ex^* \ tg$
  We have to show that $embed_C(C) \vdash \mathrm{global} \ tg \ embed_{e^*}(e^*)^\epsilon : ex^* \ tg$. To do so, we must show that $tg = \mathrm{mut}^? \ t$, $ex^* = \epsilon \vee tg = t$, and $embed_C(C) \vdash embed_{e^*}(e^*)^\epsilon : \epsilon; \epsilon; \emptyset; \emptyset \longrightarrow (t \ \alpha); \epsilon; \Gamma; \phi$.
  We have $g = \mathrm{mut}^? \ t$ and $ex^* = \epsilon \vee tg = t$ because they are premises of $C \vdash \mathbf{global} \ tg \ e^* : ex^* \ tg$.
  Further, we know $C \vdash e^* : \epsilon \longrightarrow t$ since it is also a premise of $C \vdash \mathbf{global} \ tg \ e^* : ex^* \ tg$.
  Then, we know that $embed_C(C) \vdash embed_{e^*}(e^*)^\epsilon : \epsilon; \epsilon; \emptyset; \emptyset \longrightarrow (t \ \alpha); \epsilon; \Gamma; \phi$, for some Γ and ϕ, by Lemma Sound-Embedding-of-Instructions.

□

The embedding of functions, Definition 6, both must construct a pre- and post-condition 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^*. In the precondition, the index variable context contains fresh index variables α_1^*, with associated Wasm types t_1^*, to represent the values passed as arguments. In the postcondition, the index variable context contains the index variables generated to represent the arguments passed to the function, as well as fresh index variables α_2^*, with associated Wasm types t_2^*, to represent the values returned from the function. The index constraint context is empty in both the pre- and post-condition.

Definition 6. $\boxed{embed_f(f) = \mathsf{f}}$

$$\begin{aligned} embed_f(\textbf{func } (t_1^* \to t_2^*) \textbf{ local } t^* \; e^*) &= \mathsf{func} \;\; ((t_1\ \alpha_1)^*; \emptyset \to (t_2\ \alpha_2)^*; \emptyset) \\ &\qquad\quad \mathsf{local}\ t^*\ (embed_e(e)^{(t_1^*\ t^*)})^* \\ &\quad\ \mathsf{end} \\ embed_f(\textbf{func } (t_1^* \to t_2^*)\ im) &= \mathsf{func} \;\; ((t_1\ \alpha_1)^*; \emptyset \to (t_2\ \alpha_2)^*; \emptyset) \\ &\qquad\quad im \\ &\quad\ \mathsf{end} \end{aligned}$$

Now we prove that embedding a well-typed Wasm function produces a well-typed Wasm-precheck function. Similar to the proof for global variables, this proof relies on a lemma about embedded instructions being well typed, Lemma Sound-Embedding-of-Instructions, to ensure that the embedded function body is well typed. We also require an extra requirement on the implementation of implication, that any constraint set ϕ implies the empty constraint set .

Lemma 7. Sound-Embedding-of-Functions
If $C \vdash f : ex^*\ t_1^* \to t_2^*$,
then $embed_C(C) \vdash embed_f(f) : ex^*\ ((t_1\ \alpha_1)^*; \emptyset \to (t_2\ \alpha_2)^*; \emptyset)$.

Proof. We proceed by case anaylsis on $C \vdash f : ex^*\ tg$.

- $C \vdash \textbf{func } (t_1^* \to t_2^*)\ im : ex^*\ t_1^* \to t_2^*$
  Trivially, $embed_C(C) \vdash \mathsf{func}\ ((t_1\ \alpha_1)^*; \emptyset \to (t_2\ \alpha_2)^*; \emptyset) im : ex^*\ (t_1\ \alpha_1)^*; \emptyset \to (t_2\ \alpha_2)^*; \emptyset$
- $C \vdash \textbf{func } (t_1^* \to t_2^*) \textbf{ local } t^*\ e^* : ex^*\ t_1^* \to t_2^*$
  We want to show that $embed_C(C) \vdash \mathsf{func}\ ((t_1\ \alpha_1)^*; \emptyset \to (t_2\ \alpha_2)^*; \emptyset)$
  $\mathsf{local}\ t^*\ embed_{e^*}(e^*)^{t_1^*\ t^*} : ex^*\ (t_1\ \alpha_1)^*; \emptyset \to (t_2\ \alpha_2)^*; \emptyset$
  To do so, we must show that

$$C_2 \vdash embed_{e^*}(e^*)^{t_1^*\ t^*} : \epsilon; (t_1\ \alpha_1)^*\ (t\ \alpha)^*; (\emptyset, (t_1\ \alpha_1)^*, (t\ \alpha)^*); \emptyset, (= \alpha\ (t\ 0))^* \to (t_2\ \alpha_4)^*; l_2; \Gamma_2; \phi_2$$

  where $\Gamma_2 \vdash \phi_2 \rightsquigarrow \emptyset[\alpha_2 \mapsto \alpha_4][\alpha_1 \mapsto \alpha_3]$ and

$$C_2 = embed_C(C), \mathrm{local}(t_1^*\ t^*), \mathrm{return}((t_2\ \alpha_2)^*; \emptyset), \mathrm{label}(((t_2\ \alpha_4)^*; l_2; \emptyset)$$

  We know $C, \mathrm{local}(t_1^*\ \ t^*), \mathrm{return}(t_2^*), \mathrm{label}(t_2^*) \vdash e^* : \epsilon \to t_2^*$ since it is a premise of $C \vdash \textbf{func } (t_1^* \to t_2^*) \textbf{ local } t^*\ e^* : ex^*\ t_1^* \to t_2^*$.
  By definition of $embed_C$, $C_2 = embed_C(C, \mathrm{local}(t_1^*\ t^*), \mathrm{return}(t_2^*), \mathrm{label}(t_2^*) \vdash e^* : \epsilon \to t_2^*)$.
  Then, we know that

$$C_2 \vdash embed_{e^*}(e^*)^{t_1^*\ t^*} : \epsilon; (t_1\ \alpha_1)^*\ (t\ \alpha)^*; (\emptyset, (t_1\ \alpha_1)^*, (t\ \alpha)^*); \emptyset \to (t_2\ \alpha_4)^*; l_2; \Gamma_2; \phi_2$$

  for some l_2, Γ_2, and ϕ_2, by Lemma Sound-Embedding-of-Instructions.
  The last case is somewhat tricky, as we allow the implementation of implication to be an under-approximation, and therefore cannot immediately claim that $\Gamma_2 \vdash \phi_2 \rightsquigarrow \emptyset[\alpha_2 \mapsto \alpha_4][\alpha_1 \mapsto \alpha_3]$. However, it is a reasonable requirement that any constraint set should imply the empty set, so we accept this as another requirement on the implementation of implication.

□

Embedding instructions replaces all type annotations used within the Wasm syntax with Wasm-precheck indexed type annotations, 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 7. 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 7. $\boxed{embed_e(e)^{t^*} = \text{e}}$

$$\begin{aligned}
embed_{e^*}(\textbf{block}\ (t_1^* \to t_2^*)\ e^*\ \textbf{end})^{t^*} &= \text{block}(t_1^* \to t_2^*)\ embed_{e^*}(e^*)^{t^*}\ \text{end}\\
embed_{e^*}(\textbf{loop}\ (t_1^* \to t_2^*)\ e^*\ \textbf{end})^{t^*} &= \text{loop}(t_1^* \to t_2^*)\ embed_{e^*}(e^*)^{t^*}\ \text{end}\\
embed_{e^*}(\textbf{if}\ (t_1^* \to t_2^*)\ e_1^*\ e_2^*\ \textbf{end})^{t^*} &= \text{if}(t_1^* \to t_2^*)\ embed_e(e_1^*)^{t^*}\ embed_e(e_2^*)^{t^*}\ \text{end}\\
embed_{e^*}(\textbf{call\_indirect}\ (t_1^* \to t_2^*))^{t^*} &= \text{call\_indirect}\ ((t_1\ \alpha_1)^*; \emptyset \to (t_2\ \alpha_2)^*; \emptyset)\\
embed_{e^*}(e)^{t^*} &= e, \text{otherwise}\\
embed_{e^*}(e^*)^{t^*} &= (embed_{e^*}(e)^{t^*})^*
\end{aligned}$$

Note that $(t_2\ \alpha_2)^*, (\hat{t}\ \alpha_3)^* \in \Gamma_2$ is generally implicitly proven by the structure of Γ_2.

Lemma 8. Sound-Embedding-of-Instructions
If $C \vdash e^* : t_1^* \to t_2^*$,
and $embed_C(C)_{\text{local}} = \hat{t}^*$,
and $(t_1\ \alpha_1)^*, (\hat{t}\ \alpha)^* \in \Gamma_1$,
then $\forall \Gamma_1, \phi_1.\exists, \Gamma_2, \phi_2.embed_C(C) \vdash embed_{e^*}(e^*)^{\hat{t}^*} : (t_1\ \alpha_1)^*; (\hat{t}\ \alpha)^*; \Gamma_1; \phi_1 \to (t_2\ \alpha_2)^*; (\hat{t}\ \alpha_3)^*; \Gamma_2; \phi_2$
and $(t_2\ \alpha_2)^*, (\hat{t}\ \alpha_3)^* \in \Gamma_2$.

Proof. We proceed by induction on the typing derivation.
Note: we use $\hat{t}^*$ to represent the types of local variables passed to $embed_{e^*}(e^*)$ to set them apart from other ts.

- Case: $C \vdash t.\textbf{const}\ c : \epsilon \to t$
  Trivially, $embed_C(C) \vdash t.\text{const}\ c : \epsilon; (\hat{t}\ \ \alpha)^*; \Gamma_1; \phi_1 \to (t\ \ \alpha_2); (\hat{t}\ \ \alpha)^*; \Gamma, (t\ \ \alpha_2); \phi_1, (= \alpha_2\ (t\ c))$, by Rule Const.
- Case: $C \vdash t.binop : t\ t \to t$
  Trivially, $embed_C(C) \vdash t.binop : (t\ \alpha_1)\ (t\ \alpha_2); (\hat{t}\ \alpha)^*; \Gamma_1; \phi_1 \to (t\ \alpha_3); (\hat{t}\ \alpha)^*; \Gamma_1, (t\ \alpha_3); \phi_1, (= \alpha_3\ (\|binop\|\ \alpha_1\ \alpha_2)$ by Rule Binop.
- Case: $C \vdash t.testop : t \to \textbf{i32}$
  Trivially, $embed_C(C) \vdash t.testop : (t\ \alpha_1); (\hat{t}\ \alpha)^*; \Gamma_1; \phi_1 \to (\text{i32}\ \alpha_2)(\hat{t}\ \alpha)^*; \Gamma_1, (\text{i32}\ \alpha_2); \phi_1, (= \alpha_2\ (\|testop\|\ \alpha_1)$ by Rule Testop.
- Case: $C \vdash t.relop : t\ t \to \textbf{i32}$
  Trivially, $embed_C(C) \vdash t.binop : (t\ \alpha_1)\ (t\ \alpha_2); (\hat{t}\ \alpha)^*; \Gamma_1; \phi_1 \to (\text{i32}\ \alpha_3); (\hat{t}\ \alpha)^*; \Gamma_1, (\text{i32}\ \alpha_3); \phi_1, (= \alpha_3\ (\|relop\|\ \alpha_1\ \alpha_2)$ by Rule Relop.
- Case: $C \vdash \textbf{unreachable} : t_1^* \to t_2^*$
  Trivially, $embed_C(C) \vdash \text{unreachable} : (t_1\ \ \alpha_1)^*; (t\ \ \alpha)^*; \Gamma_1; \phi_1 \to (t_2\ \ \alpha_2)^*; (\hat{t}\ \ \alpha)^*; \Gamma_2; \phi_2$ by Rule Unreachable.
- Case: $C \vdash \textbf{nop} : \epsilon \to \epsilon$
  Trivially, $embed_C(C) \vdash \text{nop} : \epsilon; (\hat{t}\ \alpha)^*; \Gamma_1; \phi_1 \to \epsilon; (\hat{t}\ \alpha)^*; \Gamma_1; \phi_1$ by Rule Nop.
- Case: $C \vdash \textbf{drop} : t \to \epsilon$
  Trivially, $embed_C(C) \vdash \text{nop} : (t\ \alpha_1); (\hat{t}\ \alpha)^*; \Gamma_1; \phi_1 \to \epsilon; (\hat{t}\ \alpha)^*; \Gamma_1; \phi_1$ by Rule Drop.
- Case: $C \vdash \textbf{select} : t\ t\ \textbf{i32} \to t$
  Trivially, $embed_C(C) \vdash t.binop : (t\ \alpha_1)\ (t\ \alpha_2); (\text{i32}\ \alpha_3); (\hat{t}\ \alpha)^*; \Gamma_1; \phi_1$
  $$\to (t\ \alpha_4); (\hat{t}\ \alpha)^*; \Gamma_1, (t\ \alpha_4); \phi_1, (\text{if}\ (=\ \alpha_3\ (\text{i32}\ 0))) \\ (=\ \alpha_4\ \alpha_2) \\ (=\ \alpha_4\ \alpha_1))$$
  by Rule Select.
- Case: $C \vdash \textbf{block}\ \ (t_1^* \to t_2^*)\ e^*\ \textbf{end} : t_1^* \to t_2^*$
  We want to show that $embed_C(C) \vdash \text{block}\ (t_1^* \to t_2^*)\ embed_{e^*}(e^*)^{t^*}\ \text{end} : (t_1\ \alpha_1)^*; l_1; \Gamma_1; \phi_1 \to (t_2\ \alpha_2)^*; l_2; \Gamma_2; \phi_2$.

To do so, we must show that $C' \vdash embed_{e^*}(e^*)^{\hat{t}_*} : (t_1\ \alpha_1)^*; l_1; \Gamma_1; \phi_1 \rightarrow (t_2\ \alpha_2)^*; l_2; \Gamma_2; \phi_2$, where $C' = embed_C(C), \text{label}((t_2\ \alpha_2)^*; l_2; \phi_2)$.
We have $C, \text{label}(t_2^*) \vdash e^* : t_1^* \rightarrow t_2^*$, as it is a premise of $C \vdash$ **block** $(t_1^* \rightarrow t_2^*)$ **end** $: t_1^* \rightarrow t_2^*$.
Further, we know $\hat{t}^* = embed_C(C)_{\text{local}}$ as it is a premise of the lemma we are trying to prove, and therefore $C'_{\text{local}} = \hat{t}^*$
Now we can invoke the inductive hypothesis on e^*, since $C' = embed_C(C, \text{label}(t_2^*))$, to get

$$C' \vdash embed_{e^*}(e^*)^{\hat{t}_*} : (t_1\ \alpha_1)^*; (\hat{t}\ \alpha_{l1})^*; (\emptyset, (t_1\ \alpha_1)^*, (\hat{t}\ \alpha_{l1})^*); \emptyset \rightarrow (t_2\ \alpha_2)^*; (\hat{t}\ \alpha_2)^*; \Gamma_2; \phi_3$$

The other premise, $\Gamma_2 \vdash \phi_3 \rightsquigarrow \phi_2$ is tricky. It is tricky because we allow the implementation of $\implies$ to be an under-approximation, so we cannot necessarily claim this immediately. However, it is a reasonable requirement of the implementation that any constraint set should imply the empty constraint set. Thus, we simply pick ϕ_2 to be the empty constraint set $\emptyset$, and therefore any ϕ_3 necessarily implies ϕ_2.

- Case: $C \vdash$ **loop** $(t_1^* \rightarrow t_2^*)\ e^*$ **end** $: t_1^* \rightarrow t_2^*$
We want to show that $embed_C(C) \vdash$ loop $(t_1^* \rightarrow t_2^*)\ embed_{e^*}(e^*)^{t^*}$ end $: (t_1\ \alpha_1)^*; l_1; \Gamma_1; \phi_1 \rightarrow (t_2\ \alpha_2)^*; l_2; \Gamma_2; \phi_2$.
To do so, we must show that $C' \vdash embed_{e^*}(e^*)^{\hat{t}_*} : (t_1\ \alpha_1)^*; l_1; \Gamma_1; \phi_3 \rightarrow (t_2\ \alpha_2)^*; l_2; \Gamma_2; \phi_2$, where $C' = embed_C(C), \text{label}((t_1\ \alpha_1)^*; l_1; \phi_3)$, and $\Gamma_1 \vdash \phi_1 \rightsquigarrow \phi_3$.
We have $C, \text{label}(t_1^*) \vdash e^* : t_1^* \rightarrow t_2^*$, as it is a premise of $C \vdash$ **loop** $(t_1^* \rightarrow t_2^*)$ **end** $: t_1^* \rightarrow t_2^*$.
Further, we know $\hat{t}^* = embed_C(C)_{\text{local}}$ as it is a premise of the lemma we are trying to prove, and therefore $C'_{\text{local}} = \hat{t}^*$
Now we can invoke the inductive hypothesis on e^*, since $C' = embed_C(C, \text{label}(t_1^*))$, to get

$$C' \vdash embed_{e^*}(e^*)^{\hat{t}_*} : (t_1\ \alpha_1)^*; (\hat{t}\ \alpha_{l1})^*; (\emptyset, (t_1\ \alpha_1)^*, (\hat{t}\ \alpha_{l1})^*); \emptyset \rightarrow (t_2\ \alpha_2)^*; (\hat{t}\ \alpha_2)^*; \Gamma_2; \phi_4$$

The other premise, $\Gamma_2 \vdash \phi_4 \rightsquigarrow \phi_2$, follows as described for the **block** case.

- Case: $C \vdash$ **if** $(t_1^* \rightarrow t_2^*)\ e_1^*\ e_2^*$ **end** $: t_1^* \rightarrow t_2^*$
We want to show that $embed_C(C) \vdash$ if $(t_1^* \rightarrow t_2^*)\ embed_{e^*}(e_1^*)^{t^*}\ embed_{e^*}(e_2^*)^{t^*}$ end $: (t_1\ \alpha_1)^*\ (\text{i32}\ \alpha); l_1; \Gamma_1; \phi_1 \rightarrow (t_2\ \alpha_2)^*; l_2; \Gamma_2; \phi_2$.
To do so, we must show that

$$C' \vdash embed_{e^*}(e_1^*)^{\hat{t}_*} : (t_1\ \alpha_1)^*; l_1; \Gamma_1; \phi_1, \neg(=\ \alpha(\text{i32}\ 0)) \rightarrow (t_2\ \alpha_2)^*; l_2; \Gamma_2; \phi_3$$

and

$$C' \vdash embed_{e^*}(e_2^*)^{\hat{t}_*} : (t_1\ \alpha_1)^*; l_1; \Gamma_1; \phi_1, (=\ \alpha(\text{i32}\ 0)) \rightarrow (t_2\ \alpha_2)^*; l_2; \Gamma_2; \phi_4$$

where $\Gamma_2 \vdash \phi_3 \rightsquigarrow \phi_2, \Gamma_2 \vdash \phi_4 \rightsquigarrow \phi_2$, and $C' = embed_C(C), \text{label}((t_2\ \alpha_2)^*; l_2; \Gamma_2; \phi_2)$.
We have $C, \text{label}(t_2^*) \vdash e_1^* : t_1^* \rightarrow t_2^*$ and $C, \text{label}(t_2^*) \vdash e_2^* : t_1^* \rightarrow t_2^*$ as they are premises of $C \vdash$ **if** $(t_1^* \rightarrow t_2^*) e_1^*\ e_2^*$ **end** $: t_1^* \rightarrow t_2^*$.
Further, we know $\hat{t}^* = embed_C(C)_{\text{local}}$ as it is a premise of the lemma we are trying to prove, and therefore $C'_{\text{local}} = \hat{t}^*$
Now we can invoke the inductive hypothesis on e_1^* and e_2^*, since $C' = embed_C(C, \text{label}(t_1^*))$, to get

$$C' \vdash embed_{e^*}(e_1^*)^{\hat{t}_*} : (t_1\ \alpha_1)^*; l_1; \Gamma_1; \phi_1 \rightarrow (t_2\ \alpha_2)^*; l_2; \Gamma_2; \phi_3$$

and

$$C' \vdash embed_{e_2^*}(e_2^*)^{\hat{t}_*} : (t_1\ \alpha_1)^*; (\hat{t}\ \alpha_{l1})^*; \Gamma_1; \phi_1 \rightarrow (t_2\ \alpha_2)^*; l_2; \Gamma_2; \phi_4$$

The other two premises, $\Gamma_2 \vdash \phi_3 \rightsquigarrow \phi_2, \Gamma_2 \vdash \phi_4 \rightsquigarrow \phi_2$, follow as described for the **block** case.

- $C \vdash$ **br** $i : t^* \; t_1^* \rightarrow t_2^*$
We want to show that $embed_C(C) \vdash \text{br } i : (t \; \alpha)^* \; (t_1 \; \alpha_1)^*; (\hat{t} \; \alpha_{l1})^*; \Gamma_1; \phi_1 \rightarrow ti_2; (\hat{t} \; \alpha_4)^*; \emptyset; \emptyset$
To do so, we must show that $embed_C(C)_{\text{label}}(i) = (t_1 \; \alpha_2)^*; (\hat{t} \; \alpha_{l2}); \phi_3$, where $\Gamma_1 \vdash \phi_1 \rightsquigarrow \phi_3[\alpha_2 \mapsto \alpha_1][\alpha_{l2} \mapsto \alpha_{l1}]$, for some α_2^* and α_{l2}^*.
We know $C_{\text{label}}(i) = t_1^*$ because it is a premise of $C \vdash$ **br** $i : t^* \; t_1^* \rightarrow t_2^*$.
Then, $embed_C(C)_{\text{label}}(i) = (t_1 \; \alpha_2)^*; (\hat{t} \; \alpha_{l2}); \emptyset$, by the definition of $embed_C$.
The other premise, $\Gamma_1 \vdash \phi_1 \rightsquigarrow \emptyset[\alpha_2 \mapsto \alpha_1][\alpha_{l2} \mapsto \alpha_{l1}]$, follows as described for the **block** case.
- $C \vdash$ **br_if** $i : t_1^* \; \text{i32} \rightarrow t_2^*$
We want to show that
$embed_C(C) \vdash \text{br\_if } i : (t_1 \; \alpha_1)^* \; (\text{i32} \; \alpha); (\hat{t} \; \alpha_3)^*; \Gamma_1; \phi_1 \rightarrow (t_1 \; \alpha_1)^*; (\hat{t} \; \alpha_3); \Gamma_1; \phi_1, (= \; \alpha \; (\text{i32} \; 0))$
To do so, we must show that $embed_C(C)_{\text{label}}(i) = (t_1 \; \alpha_2)^*; (\hat{t} \; \alpha_{l2}); \phi_3$, where $\Gamma_1 \vdash \phi_1, \neg(= \alpha \; (\text{i32} \; 0)) \rightsquigarrow \phi_3[\alpha_2 \mapsto \alpha_1][\alpha_{l2} \mapsto \alpha_{l1}]$, for some α_2^* and α_{l2}^*.
We know $C_{\text{label}}(i) = t^*$ because it is a premise of $C \vdash$ **br_if** $i : t_1^* \; \text{i32} \rightarrow t_2^*$.
Then, $embed_C(C)_{\text{label}}(i) = (t_1 \; \alpha_2)^*; (\hat{t} \; \alpha_{l2}); \emptyset$, by the definition of $embed_C$.
The other premise, $\Gamma_1 \vdash \phi_1, \neg(= \; \alpha \; (\text{i32} \; 0)) \rightsquigarrow \emptyset[\alpha_2 \mapsto \alpha_1][\alpha_{l2} \mapsto \alpha_{l1}]$, follows as described for the **block** case.
- $C \vdash$ **br_table** $i^+ : t_0^* \; t_1^*$ **i32** $\rightarrow t_2^*$
We want to show that $embed_C(C) \vdash \text{br\_table } i^+ : (t_0 \; \alpha_0)^* \; (t_1 \; \alpha_1)^* \; (\text{i32} \; \alpha); (\hat{t} \; \alpha_{l1})^*; \Gamma_1; \phi_1 \rightarrow \epsilon; \epsilon; \emptyset; \emptyset$
To do so, we must show that $embed_C(C)_{\text{label}}(i) = (t_1 \; \alpha_2)^*; (\hat{t} \; \alpha_{l2}); \phi_3$, where $\Gamma_1 \vdash \phi_1 \rightsquigarrow \phi_3[\alpha_2 \mapsto \alpha_1][\alpha_{l2} \mapsto \alpha_{l1}]^+$, for some α_2^* and α_{l2}^*.
We know that $(C_{\text{label}}(i) = t_1^*)^+$ because it is a premise of $C \vdash$ **br_table** $i^+ : t_0^* \; t_1^*$ **i32** $\rightarrow t_2^*$.
Then, $embed_C(C)_{\text{label}}(i) = ((t_1 \; \alpha_2)^*; (\hat{t} \; \alpha_{l2}); \emptyset)^+$, by the definition of $embed_C$.
The other premise, $(\Gamma_1 \vdash \phi_1 \rightsquigarrow \emptyset[\alpha_2 \mapsto \alpha_1][\alpha_{l2} \mapsto \alpha_{l1}])^+$, follows as described for the **block** case.
- $C \vdash$ **return** $: t^* \; t_1^* \rightarrow t_2^*$
We want to show that $embed_C(C) \vdash \text{return} i : (t \; \alpha)^* \; (t_1 \; \alpha_1)^*; (\hat{t} \; \alpha_{l1})^*; \Gamma_1; \phi_1 \rightarrow ti_2; (\hat{t} \; \alpha_4)^*; \emptyset; \emptyset$
To do so, we must show that $embed_C(C)_{\text{return}} = (t_1 \; \alpha_2)^*; \phi_3$, where $\Gamma_1 \vdash \phi_1 \rightsquigarrow \phi_3[\alpha_2 \mapsto \alpha_1]$, for some α_2^*.
We know $C_{\text{return}} = t_1^*$ because it is a premise of $C \vdash$ **return***i* $: t^* \; t_1^* \rightarrow t_2^*$.
Then, $embed_C(C)_{\text{return}} = (t_1 \; \alpha_2)^*; ; \emptyset$, by the definition of $embed_C$.
The other premise, $\Gamma_1 \vdash \phi_1 \rightsquigarrow \emptyset[\alpha_2 \mapsto \alpha_1]$, follows as described for the **block** case.
- $C \vdash$ **call_indirect** $(t_1^* \rightarrow t_2^*) : t_1^* \rightarrow t_2^*$
We want to show that

$$\begin{aligned} embed_C(C) \vdash \; &\text{call\_indirect } ((t_1 \; \alpha_3)^*; \emptyset \rightarrow (t_2 \; \alpha_4)^*; \emptyset) \\ &: (t_1 \; \alpha_1)^*; (\hat{t} \; \alpha_3)^*; \Gamma_1; \phi_1 \rightarrow (t_2 \; \alpha_2)^*; (\hat{t} \; \alpha_3)^*; \Gamma_1, (t_2 \; \alpha_4)^*; \phi_1 \end{aligned}$$

To do so, we must show that $embed_C(C)_{\text{table}} = (n, tfi^n)$, and that $\Gamma_1 \vdash \phi_1 \rightsquigarrow \emptyset[\alpha_1 \mapsto \alpha_2]$.
We know $C_{\text{table}} = n$, as it is a premise of $C \vdash$ **call_indirect** $(t_1^* \rightarrow t_2^*) : t_1^* \rightarrow t_2^*$.
Then, $embed_C(C)_{\text{table}} = (n, tfi^n)$, by definition of $embed_C$.
The other premise, $\Gamma_1 \vdash \phi_1 \rightsquigarrow \emptyset[\alpha_1 \mapsto \alpha_2$, follows as described for the **block** case.
- $C \vdash$ **get_local** $i : \epsilon \rightarrow t$
We want to show that $embed_C(C) \vdash \text{get\_local } i : \epsilon; (\hat{t} \; \alpha_1)^*; \Gamma_1; \phi_1 \rightarrow (t \; \alpha); (\hat{t} \; \alpha_1)^*; \Gamma_1; \phi_1$
To do so, we must show that $embed_C(C)_{\text{local}}(i) = t$, and that $(t \; \alpha) = ((\hat{t} \; \alpha_1)^*)(i)$.
$(t \; \alpha) = ((\hat{t} \; \alpha_1)^*)(i)$ is trivially correct by construction (we simply choose a t and α such that $(t \; \alpha) = ((\hat{t} \; \alpha_1)^*)(i)$.

Then, since $embed_C(C)_{\text{local}} = \hat{t}^*$ is a premise of this lemma, we have that $embed_C(C)_{\text{local}}(i) = t$.

- $C \vdash$ **set_local** $i : t \rightarrow \epsilon$
  We want to show that $embed_C(C) \vdash$ set_local $i : (t\ \alpha); (\hat{t}\ \alpha_1)^*; \Gamma_1; \phi_1 \rightarrow \epsilon; (\hat{t}\ \alpha_2)^*; \Gamma_1; \phi_1$
  To do so, we must show that $embed_C(C)_{\text{local}}(i) = t$, and that $(\hat{t}\ \alpha_2)^* = ((\hat{t}\ \alpha_1)^*)[i := (t\ \alpha)]$.
  $(\hat{t}\ \alpha_2)^* = ((\hat{t}\ \alpha_1)^*)[i := (t\ \alpha)]$ is trivially correct by construction (we simply choose a $(\hat{t}\ \alpha_2)^*$ such that $(\hat{t}\ \alpha_2)^* = ((\hat{t}\ \alpha_1)^*)[i := (t\ \alpha)]$.
  Then, since $embed_C(C)_{\text{local}} = \hat{t}^*$ is a premise of this lemma, we have that $embed_C(C)_{\text{local}}(i) = t$.
- $C \vdash$ **tee_local** $i : t \rightarrow t$
  We want to show that $embed_C(C) \vdash$ set_local $i : (t\ \alpha); (\hat{t}\ \alpha_1)^*; \Gamma_1; \phi_1 \rightarrow (t\ \alpha); (\hat{t}\ \alpha_2)^*; \Gamma_1; \phi_1$
  To do so, we must show that $embed_C(C)_{\text{local}}(i) = t$, and that $(\hat{t}\ \alpha_2)^* = ((\hat{t}\ \alpha_1)^*)[i := (t\ \alpha)]$.
  $(\hat{t}\ \alpha_2)^* = ((\hat{t}\ \alpha_1)^*)[i := (t\ \alpha)]$ is trivially correct by construction (we simply choose a $(\hat{t}\ \alpha_2)^*$ such that $(\hat{t}\ \alpha_2)^* = ((\hat{t}\ \alpha_1)^*)[i := (t\ \alpha)]$.
  Then, since $embed_C(C)_{\text{local}} = \hat{t}^*$ is a premise of this lemma, we have that $embed_C(C)_{\text{local}}(i) = t$.
- Case: $C \vdash$ **get_global** $i : \epsilon \rightarrow t$
  We want to show that $embed_C(C) \vdash$ **get_global** $i : \epsilon; (\hat{t}\ \alpha)^*; \Gamma_1; \phi_1 \rightarrow (t\ \alpha_2); (\hat{t}\ \alpha)^*; \Gamma, (t\ \alpha_2); \phi_1$.
  To do so, we must show that $embed_C(C)_{\text{global}}(i) = \text{mut}^?\ t$.
  We know $C_{\text{global}}(i) = \text{mut}^?\ t$, as it is a premise of $C \vdash$ **get_global** $i : \epsilon \rightarrow t$.
  Then, $embed_C(C)_{\text{global}}(i) = \text{mut}^?\ t$, by definition of $embed_C$.
- Case: $C \vdash$ **set_global** $i : t \rightarrow \epsilon$
  We want to show that $embed_C(C) \vdash$ **set_global** $i : (t\ \alpha_1); (\hat{t}\ \alpha)^*; \Gamma_1; \phi_1 \rightarrow \epsilon; (\hat{t}\ \alpha)^*; \Gamma_1; \phi_1$.
  To do so, we must show that $embed_C(C)_{\text{global}}(i) = \text{mut}^?\ t$.
  We know $C_{\text{global}}(i) = \text{mut}^?\ t$, as it is a premise of $C \vdash$ **get_global** $i : \epsilon \rightarrow t$.
  Then, $embed_C(C)_{\text{global}}(i) = \text{mut}^?\ t$, by definition of $embed_C$.
- Case: $C \vdash t$.**load** $(tp\_sx)^?\ align\ o :$ **i32** $\rightarrow t$
  We want to show that
  $embed_C(C) \vdash$ **get_global** $i :$ (**i32** $()\ \alpha); (\hat{t}\ \alpha)^*; \Gamma_1; \phi_1 \rightarrow (t\ \alpha_2); (\hat{t}\ \alpha)^*; \Gamma, (t\ \alpha_2); \phi_1$.
  To do so, we must show that $embed_C(C)_{\text{memory}} = n$ and $2^{align} \leq (|tp| <)^? t$.
  We know $C_{\text{memory}} = n$ and $2^{align} \leq (|tp| <)^? t$ as they are premises of $C \vdash t$.**load** $(tp\_sx)^?\ align\ o :$ **i32** $\rightarrow t$.
  Then, $embed_C(C)_{\text{memory}} = n$, by definition of $embed_C$.
- Case: $C \vdash t$.**store** $tp^?\ align\ o :$ **i32** $t \rightarrow \epsilon$
  We want to show that $embed_C(C) \vdash$ **set_global** $i :$ (**i32** $\alpha_1)\ (t\ \alpha_2); (\hat{t}\ \alpha)^*; \Gamma_1; \phi_1 \rightarrow \epsilon; (\hat{t}\ \alpha)^*; \Gamma_1; \phi_1$.
  To do so, we must show that $embed_C(C)_{\text{memory}} = n$ and $2^{align} \leq (|tp| <)^? t$.
  We know $C_{\text{memory}} = n$ and $2^{align} \leq (|tp| <)^? t$ as they are premises of $C \vdash t$.**store** $tp^?\ align\ o :$ **i32** $t \rightarrow \epsilon$.
  Then, $embed_C(C)_{\text{memory}} = n$, by definition of $embed_C$.
- Case: $C \vdash$ **current_memory** $: \epsilon \rightarrow$ **i32**
  We want to show that
  $embed_C(C) \vdash$ **get_global** $i : \epsilon; (\hat{t}\ \alpha)^*; \Gamma_1; \phi_1 \rightarrow$ (**i32** $()\ \alpha); (\hat{t}\ \alpha)^*; \Gamma,$ (**i32** $()\ \alpha); \phi_1$.
  To do so, we must show that $embed_C(C)_{\text{memory}} = n$.
  We know $C_{\text{memory}} = n$ as it is a premise of $C \vdash$ **current_memory** $: \epsilon \rightarrow$ **i32**.
  Then, $embed_C(C)_{\text{memory}} = n$, by definition of $embed_C$.
- Case: $C \vdash$ **grow_memory** $:$ **i32** $\rightarrow$ **i32**

We want to show that
$embed_C(C) \vdash \textbf{get\_global}\ i : (\textbf{i32}\ \alpha_1);\ (\hat{t}\ \alpha)^*;\ \Gamma_1;\ \phi_1 \rightarrow (\textbf{i32}\ ()\ \alpha_2);\ (\hat{t}\ \alpha)^*;\ \Gamma, (\textbf{i32}\ ()\ \alpha_2);\ \phi_1$.
To do so, we must show that $embed_C(C)_{\text{memory}} = n$.
We know $C_{\text{memory}} = n$ as it is a premise of $C \vdash \textbf{current\_memory}\ : \epsilon \rightarrow \textbf{i32}$.
Then, $embed_C(C)_{\text{memory}} = n$, by definition of $embed_C$.

- Case: $C \vdash \epsilon : \epsilon \rightarrow \epsilon$
Trivially, $embed_C(C) \vdash \epsilon : \epsilon;\ (\hat{t}\ \alpha)^*;\ \Gamma_1;\ \phi_1 \rightarrow \epsilon;\ (\hat{t}\ \alpha)^*;\ \Gamma_1;\ \phi_1$ by Rule Empty.
- Case: $C \vdash e_1^*\ e_2 : t_1^* \rightarrow t_3^*$
We want to show that $embed_C(C) \vdash e_1^*\ e_2 : (t_1\ \alpha_1)^*;\ (\hat{t}\ \alpha)^*;\ \Gamma_1;\ \phi_1 \rightarrow (t_3\ \alpha_3)^*;\ (\hat{t}\ \alpha_4)^*;\ \Gamma_3;\ \phi_3$.
To do so, we must show that

$$embed_C(C) \vdash embed_{e^*}(e_1^*)^{\hat{t}^*} : (t_1\ \alpha_1)^*;\ (\hat{t}\ \alpha)^*;\ \Gamma_1;\ \phi_1 \rightarrow (t_2\ \alpha_2)^*;\ (\hat{t}\ \alpha_5)^*;\ \Gamma_2;\ \phi_2$$

and

$$embed_C(C) \vdash embed_{e^*}(e_2)^{\hat{t}^*} : (t_2\ \alpha_2)^*;\ (\hat{t}\ \alpha_5)^*;\ \Gamma_2;\ \phi_2 \rightarrow (t_3\ \alpha_3)^*;\ (\hat{t}\ \alpha_4)^*;\ \Gamma_3;\ \phi_3$$

We have $C \vdash e_1^* : t_1^* \rightarrow t_2^*$, and $C \vdash e_2 : t_2^* \rightarrow t_3^*$, as they are premises of $C \vdash e_1^*\ e_2 : t_1^* \rightarrow t_3^*$.
Then, by the inductive hypothesis, we have

$$embed_C(C) \vdash embed_{e^*}(e_1^*)^{\hat{t}^*} : (t_1\ \alpha_1)^*;\ (\hat{t}\ \alpha)^*;\ \Gamma_1;\ \phi_1 \rightarrow (t_2\ \alpha_2)^*;\ (\hat{t}\ \alpha_5)^*;\ \Gamma_2;\ \phi_2$$

and

$$embed_C(C) \vdash embed_{e^*}(e_2)^{\hat{t}^*} : (t_2\ \alpha_2)^*;\ (\hat{t}\ \alpha_5)^*;\ \Gamma_2;\ \phi_2 \rightarrow (t_3\ \alpha_3)^*;\ (\hat{t}\ \alpha_4)^*;\ \Gamma_3;\ \phi_3$$

- Case: $C \vdash e^* : t^*\ t_1^* \rightarrow t^*\ t_2^*$
We want to show that

$$embed_C(C) \vdash e^* : (t\ \alpha)^*\ (t_1\ \alpha_1)^*;\ (\hat{t}\ \alpha_3)^*;\ \Gamma_1;\ \phi_1 \rightarrow (t\ \alpha)^*\ (t_2\ \alpha_2)^*;\ (\hat{t}\ \alpha_5)^*;\ \Gamma_2;\ \phi_2$$

.
To do so, we must show that

$$embed_C(C) \vdash embed_{e^*}(e_1^*)^{\hat{t}^*} : (t_1\ \alpha_1)^*;\ (\hat{t}\ \alpha_3)^*;\ \Gamma_1;\ \phi_1 \rightarrow (t_2\ \alpha_2)^*;\ (\hat{t}\ \alpha_5)^*;\ \Gamma_2;\ \phi_2$$

We have $C \vdash e^* : t_1^* \rightarrow t_2^*$, as it is a premis of $C \vdash e^* : t^*\ t_1^* \rightarrow t^*\ t_2^*$.
Then, by the inductive hypothesis, we have

$$embed_C(C) \vdash embed_{e^*}(e_1^*)^{\hat{t}^*} : (t_1\ \alpha_1)^*;\ (\hat{t}\ \alpha_3)^*;\ \Gamma_1;\ \phi_1 \rightarrow (t_2\ \alpha_2)^*;\ (\hat{t}\ \alpha_5)^*;\ \Gamma_2;\ \phi_2$$

□

These are not the only differences in the surface syntax between Wasm and Wasm-precheck: we also introduced four new instructions (the ✓-tagged instructions). The definition of embedding we have introduced has been entirely syntactic, but that will not work for replacing non-✓-tagged instructions with ✓-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 ✓-tagged instructions, though we do posit the existence of a trivial embedding that would provide ✓-tagged instructions. One could, for example, check at every div, call_indirect , load , and store whether the ✓-tagged version of the instruction is well typed, and only if it is well typed replace the instruction with the ✓-tagged version. However, a more sophisticated static analysis could provide more precise type annotations and therefore potentially allow even more check eliminations.

B.2 Erasing Wasm-precheck to Wasm

We provide an erasure function for Wasm-precheck that transforms Wasm-precheck programs into Wasm programs by discarding the extra information from the Wasm-precheck type system and replacing ✓-tagged instructions with their non-tagged counterparts. Erasure is used in the type safety proof. 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-precheck (run time) program produces a well-typed Wasm (run time) program.

As with the presentation of the embedding, we typeset Wasm-precheck 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-precheck as in Wasm.

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

$$\begin{array}{rcl} erase_{module}(\mathsf{module}\ f^*\ glob^*\ tab^?\ mem^?) & = & \mathbf{module}\ erase_f(f)^* \\ & & erase_g(glob)^* \\ & & erase_t(tab)^? \\ & & mem^? \end{array}$$

Erasing a table definition $\mathsf{table}\ n\ i^n$ does nothing, since a table definition has the same syntax in Wasm-precheck and in Wasm. However, erasing an imported table declaration $\mathsf{table}\ (n, tfi^n)\ im$ 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 9. $\boxed{erase_t(tab) = \mathbf{tab}}$

$$\begin{array}{rcl} (ex^*)\ \mathsf{table}\ n\ i^n & = & (ex^*)\ \mathbf{table}\ n\ i^n \\ (ex^*)\ \mathsf{table}\ (n, tfi^n)\ im & = & (ex^*)\ \mathbf{table}\ n\ im \end{array}$$

Erasing a global definition erases the instruction sequence used to initialize the global variable. However, erasing an imported global declaration does nothing. We do not use or care about the exports, since they are unchanged and only used for linking, so we omit them.

Definition 10. $\boxed{erase_g(glob) = \mathbf{glob}}$

$$\begin{array}{rcl} (ex^*)\ \mathsf{global}\ tg\ e^* & = & (ex^*)\ \mathbf{global}\ tg\ erase_{e^*}(e^*) \\ (ex^*)\ \mathsf{global}\ tg\ im & = & (ex^*)\ \mathbf{global}\ tg\ im \end{array}$$

To erase a function definition f, we erase both the type declaration $ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$ and the body e^*. We can also erase an imported function by erasing the declared type tfi.

Definition 11. $\boxed{erase_f(f) = \mathbf{f}}$

$$\begin{array}{rcl} erase(\mathsf{func}\ ((t_1\ \alpha_1)^*; l_1; \Gamma_1; \phi_1 \rightarrow (t_2\ \alpha_2)^*; l_2; \Gamma_1; \phi_2)\ \mathsf{local}\ t^*\ e^*) & = & \mathbf{func}\ (t_1^* \rightarrow t_2^*)\ \mathbf{local}\ t^*\ erase_{e^*}(e^*) \\ erase_f(\mathsf{func}\ ((t_1\ \alpha_1)^*; l_1; \Gamma_1; \phi_1 \rightarrow (t_2\ \alpha_2)^*; l_2; \Gamma_1; \phi_2)\ im) & = & \mathbf{func}\ (t_1^* \rightarrow t_2^*)\ im \end{array}$$

We show that erasing a Wasm-precheck 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 9. Sound-Function-Typing-Erasure

If $C \vdash \text{func} \;\; (t_1 \;\; \alpha_1)^*; l_1; \Gamma_1; \phi_1 \;\longrightarrow\; (t_2 \;\; \alpha_2); l_2; \Gamma_2; \phi_2) \;\; \text{local} \;\; t^* \;\; e^* \;:\; ex^* \;\; (t_1 \;\; \alpha_1)^*; l_1; \Gamma_1; \phi_1 \;\longrightarrow\; (t_2 \;\alpha_2); l_2; \Gamma_2; \phi_2$

then $erase_C(C) \;\vdash \mathbf{func} \;(t_1^* \rightarrow t_2^*) \;\text{local}\; t^* \; erase_{e^*}(e^*)) : ex^* \; t_1^* \rightarrow t_2^*$

Proof. We must show that $erase_C(C), \text{local}(t_1^* \; t^*), \text{label}(t_2^*), \text{return}(t_2^*) \vdash erase_{e^*}(e^*) : t_1^* \rightarrow t_2^*$ 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.

$$C, \text{local}(t_1^* \; t^*), \text{label}((t_2 \; \alpha_2)^*; l_2; \Gamma_2; \phi_2), \text{return}((t_2 \; \alpha_2)^*; l_2; \Gamma_2; \phi_2) \vdash e^* : (t_1 \; \alpha_1)^*; l_1; \Gamma_1; \phi_1 \longrightarrow (t_2 \; \alpha_2); l_2; \Gamma_2; \phi_2$$

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

$$erase_C(C), \text{local}(t_1^* \; t^*), \text{label}(t_2^*), \text{return}(t_2^*) \vdash erase_{e^*}(e^*) : t_1^* \longrightarrow t_2^*$$

□

Erasing an indexed type function keeps only the primitive Wasm types (t_1^* and t_2^*) from the indexed types representing the stack ($(t_1 \; \alpha_1)^*$ and $(t_2 \; \alpha_2)^*$), and discards everything else.

Definition 12. $\boxed{erase_{tfi}(tfi) = \mathbf{tf}}$

$erase_{tfi}((t_1 \; \alpha_1)^*; l_1; \Gamma_1; \phi_1 \longrightarrow (t_2 \; \alpha_2)^*; l_2; \Gamma_2 \phi_2) = t_1^* \longrightarrow t_2^*$

Erasing instructions involves erasing the indexed function types for indirect function calls, which includes it as part of their syntax. We must also remove the ✓tag from ✓-tagged instructions to turn them into instructions that exist in Wasm.

Definition 13. $\boxed{erase_{e^*}(e) = \mathbf{e}}$

$$\begin{array}{rcl}
erase_{e^*}(\text{block } (t_1^n \rightarrow t_2^n) \; e^* \text{ end}) & = & \mathbf{block} \;\; (t_1^n \rightarrow t_2^n) \; erase[e^*]e^* \;\; \mathbf{end} \\
erase_{e^*}(\text{loop } (t_1^n \rightarrow t_2^n) \; e^* \text{ end}) & = & \mathbf{loop} \;\; (t_1^n \rightarrow t_2^n) \; erase[e^*]e^* \;\; \mathbf{end} \\
erase_{e^*}(\text{if } (t_1^n \rightarrow t_2^n) \; e_1^* \; e_2^* \text{ end}) & = & \mathbf{if} \;\; (t_1^n \rightarrow t_2^n) \; erase[e^*]e_1^* \; erase[e^*]e_2^* \;\; \mathbf{end} \\
erase_{e^*}(\text{label}_n \; \{e_0^*\} \; e^* \text{ end}) & = & \mathbf{label}_n \; \{erase_{e^*}(e_0^*)\} \\
 & & \qquad erase_{e^*}(e^*) \\
 & & \;\mathbf{end} \\
erase_{e^*}(\text{local}_n \; \{i; v^*\} \; e^* \text{ end}) & = & \mathbf{local}_n \; \{i; v^*\} \\
 & & \qquad erase_{e^*}(e^*) \\
 & & \;\mathbf{end} \\
erase_{e^*}(\text{call\_indirect } tfi) & = & \mathbf{call\_indirect} \; erase_{tfi}(tfi) \\
erase_{e^*}(\text{call\_indirect✓ } tfi) & = & t.\mathbf{call\_indirect} \; erase_{tfi}(tfi) \\
erase_{e^*}(t.\text{div✓}) & = & t.\mathbf{div} \\
erase_{e^*}(t.\text{store✓ } tp^? \; align \; o) & = & t.\mathbf{store} \; tp^? \; align \; o \\
erase_{e^*}(t.\text{load✓ } (tp\_sx)^? \; align \; o) & = & t.\mathbf{load} \; (tp\_sx)^? \; align \; o \\
erase_{e^*}(e) & = & e, \text{otherwise} \\
erase_{e^*}(e^*) & = & erase_{e^*}(e)^*
\end{array}$$

Erasing Typing Constructs. Here, we prove that erasing a Wasm-precheck static typing derivation is sound with respect to Wasm's type system. This means that erasure on the Wasm-precheck static typing judgment is sound with respect to Wasm's type system. Specifically, a Wasm-precheck 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 10. Sound-Static-Typing-Erasure
If $C \vdash e^* : (t_1\ \alpha_1)^*; l_1; \Gamma_1; \phi_1 \longrightarrow (t_2\ \alpha_2)^*; l_2; \Gamma_2; \phi_2$,
then $erase_C(C) \vdash erase_{e^*}(e^*) : t_1^* \longrightarrow t_2^*$

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: $C \vdash t.binop : (t\ \alpha_1)\ (t\ \alpha_2); l_1; \Gamma_1; \phi_1 \longrightarrow (t\ \alpha_3); l_1; \Gamma_1, (t\ \alpha_3); \phi_1, (= a_3\ (t.binop\ a_1\ a_2))$
  We want to show that $erase_C(C)\ \vdash erase_{e^*}(t.binop) : t\ t \longrightarrow t$
  Then, by the definition of $erase_e$, we want to show that $erase_C(C) \vdash t.binop : t\ t \longrightarrow t$ is valid in Wasm.
  Trivially, we have $erase_C(C) \vdash t.binop : t\ t \longrightarrow t$.
- Case: $C \vdash \text{block}\ (t_1^n \longrightarrow t_2^n)e^*\ \text{end} : (t_1\ \alpha_1)^*; l_1; \Gamma_1; \phi_1 \longrightarrow (t_2\ \alpha_2)^*; l_2; \Gamma_2; \phi_2$
  We want to show that $erase_C(C) \vdash \textbf{block}\ \ (t_1^* \longrightarrow t_2^*)\ erase_{e^*}(e^*)\ \textbf{end} : (t_1^* \longrightarrow t_2^*)$.
  This proof is slightly more involved, since the derivation for this rule includes a premise that
  $$C, \text{label}((t_2\ \alpha_2)^*; l_2; \Gamma_2; \phi_2) \vdash e^* : (t_1\ \alpha_1)^*; l_1; \Gamma_1; \phi_1 \longrightarrow (t_2\ \alpha_2)^*; l_2; \Gamma_2; \phi_2$$
  By the inductive hypothesis for the well-typedness of e^*, we have that $erase_C(C, \text{label}((t_2\ \alpha_2)^*; l_2; \Gamma_2; \phi_2))$ $erase_{e^*}(e^*) : t_1^* \longrightarrow t_2^*$
  Then we have $erase_C(C), \text{label}(t_2^*) \vdash erase_{e^*}(e^*) : t_1^* \longrightarrow t_2^*$ by definition of $erase_C$.
- Case: $C \vdash \text{br}\ \ i : (t_1\ \alpha_1)^*\ (t\ a)^*; l_1; \Gamma_1; \phi_1 \longrightarrow (t_2\ \alpha_2)^*; l_2; \Gamma_2; \phi_2$
  We want to show that $erase_C(C) \vdash \textbf{br}\ i : t_1^*\ t^* \longrightarrow t_2^*$
  We have to reason about $erase_C(C)$ because the typing judgment relies on the label stack from C.
  From $C \vdash \text{br}\ \ i : (t_1\ \alpha_1)^*\ (t\ a)^*; l_1; \Gamma_1; \phi_1 \longrightarrow (t_2\ \alpha_2)^*; l_2; \Gamma_2; \phi_2$, we have that $C_{\text{label}}(i) = (t\ a)^*; l_1; \Gamma_3; \phi_3$, where $\Gamma_1 \vdash \phi_1 \Longrightarrow \phi_3$, since it is a premise.
  Then $erase_C(C)_{\text{label}}(i) = t^*$, by the definition of $erase_C$.
  Thus, $erase_C(C) \vdash \textbf{br}\ i : t_1^* \longrightarrow t_2^*$.
- Case: $C \vdash \text{call}\ \ i : (t_1\ \alpha_1)^*; l_1; \Gamma_1; \phi_1 \longrightarrow (t_2\ \alpha_2)^*; l_2; \Gamma_2; \phi_2$
  We want to show that $erase_C(C) \vdash \textbf{call}\ i : t_1^* \longrightarrow t_2^*$
  We again have to reason about $erase_C(C)$ because the typing judgment relies on the function type from C.
  From $C \vdash \text{call}\ \ i : (t_1\ \alpha_1)^*; l_1; \Gamma_1; \phi_1 \longrightarrow (t_2\ \alpha_2)^*; l_2; \Gamma_2; \phi_2$, we have that $C_{\text{func}}(i) = (t_1\ \alpha_1)^*; l_1; \Gamma_3; \phi_3 \longrightarrow (t_2\ \alpha_2)^*; l_4; \Gamma_4; \phi_4$.
  Then, $erase_C(C)_{\text{func}}(i) = t_1^* \longrightarrow t_2^*$, by the definition of $erase_C$.
  Thus, $erase_C(C) \vdash \textbf{call}\ i : t_1^* \longrightarrow t_2^*$.
- Case: $C \vdash \text{set\_local}\ \ i : (t\ a); l_1; \Gamma_1; \phi_1 \longrightarrow \epsilon; l_1[i := a]; \Gamma_1; \phi_1$
  We want to show that $erase_C(C) \vdash \textbf{set\_local}\ i : t \longrightarrow \epsilon$
  We again have to reason about $erase_C(C)$ because the typing judgment relies on the local variable types from C.

From $C \vdash \text{set\_local}\ \ i : (t\ a); l_1; \Gamma_1; \phi_1 \rightarrow \epsilon; l_1[i := a]; \Gamma_1; \phi_1$, 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 $erase_C$.
Thus, $erase_C(C) \vdash \textbf{set\_local}\ i : t \rightarrow \epsilon$.

- Case: $C \vdash e_1^*\ e_2 : (t_1\ \alpha_1)^*; l_1; \Gamma_1; \phi_1 \rightarrow (t_2\ \alpha_2); l_2; \Gamma_2; \phi_2$
We want to show that $erase_C(C) \vdash erase_{()}[e^*](e_1^*\ e_2) : t_1^* \rightarrow t_2^*$
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 subsequences together to get a well-typed sequence.
We know that $C \vdash e_1^* : (t_1\ \alpha_1)^*; l_1; \Gamma_1; \phi_1 \rightarrow (t_3\ \alpha_3); l_3; \Gamma_3; \phi_3$ and that $C \vdash e_2 : (t_3\ \alpha_3)^*; l_3; \Gamma_3; \phi_3 \rightarrow (t_2\ \alpha_2); l_2; \Gamma_2; \phi_2$ because they are premises of Rule Composition which we have assumed to hold.
$erase_C(C) \vdash erase_{e^*}(e_1^*) : t_1^* \rightarrow t_3^*$ by the inductive hypothesis on e_1^*, and $erase_C(C) \vdash erase_{e^*}(e_2) : t_3^* \rightarrow t_2^*$, by the inductive hypothesis on e_2.
Thus, $erase_C(C) \vdash erase_{e^*}(e_1^*)\ erase_{e^*}(e_2) : t_1^* \rightarrow t_2^*$.

□

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 $((t_1\ \alpha_1)^*; l_1; \Gamma_1; \phi_1)^*$ and the return stack $((t_2\ \alpha_2)^*; l_2; \Gamma_2; \phi_2)^?$. 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 14. $\boxed{erase_C(C) = \mathrm{C}}$

$$\begin{array}{lcl} erase_C(\{\text{func } tfi^*, \text{ global } tg^*, & = & \{\text{func } erase_{tfi}(tfi)^*, \\ \quad \text{table } (n, tfi_2^*)^?, & & \ \text{global } tg^*, \text{ table } n^?, \\ \quad \text{memory } m^?, \text{ local } t^*, & & \ \text{memory } m^?, \text{local } t^*, \\ \quad \text{label } ((t_1\ \alpha_1)^*; l_1; \Gamma_1; \phi_1)^*, & & \ \text{label } (t_1^*)^*, \text{ return } (t_2^*)^?\} \\ \quad \text{return } ((t_2\ \alpha_2)^*; l_2; \Gamma_2; \phi_2)^?\}) & & \end{array}$$

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-precheck 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 5. Sound-Program-Typing-Erasure
If $\vdash_i s; v^*\ e^* : (t_2\ \alpha_2)^*; l_2; \Gamma_2; \phi_2$,
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 $erase_S(S); \vdash_i erase_{e^*}(e^*) : \epsilon \rightarrow t_2^*$.

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

Then, $\vdash erase_s(s) : erase_S(S')$ by Lemma Sound-Store-Erasure.

We also have $S;\ \vdash_i v^*; e^* : \epsilon; l_1; \Gamma_1; \phi_1 \longrightarrow (t_2\ \alpha_2)^*; l_2; \Gamma_2; \phi_2$ as a premise of Rule Program.

In which case we have $erase_S(S);\ \vdash_i v^*; erase_{e^*}(e^*) : t_1^* \longrightarrow t_2^*$ 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 $((t_2\ \alpha_2)^*; l_2; \Gamma_2; \phi_2)^?$ 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 11. Sound-Code-Typing-Erasure
If $S;\ \vdash_i v^*; e^* : \epsilon; l_1; \Gamma_1; \phi_1 \longrightarrow (t_2\ \alpha_2)^*; l_2; \Gamma_2; \phi_2$,
then $erase_S(S);\ \vdash_i v^*; erase_{e^*}(e^*) : epsilon \longrightarrow t_2^*$

Proof. We must show that $(\vdash v : t_v)^*$ and $erase_S(S); erase_C(S_{\text{inst}}(i)), \text{local } t_v^* \vdash erase_{e^*}(e^*) : epsilon \longrightarrow t_2^*$

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

We also have $S; S_{\text{inst}}(i), \text{local } t_v^* \vdash e^* : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$, for some Γ_1,ϕ_1, and ϕ_2, by inversion on the Code judgment.

Then, by Lemma Sound-Admin-Typing-Erasure, we have that $erase_S(S); erase_C(()S_{\text{inst}}(i), \text{local } t_v^*) \vdash erase_{e^*}(e^*) : \epsilon \longrightarrow t_2^*$ □

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 12. Sound-Admin-Typing-Erasure
If $S; C \vdash e^* : (t_1\ \alpha_1)^*; l_1; \Gamma_1; \phi_1 \longrightarrow (t_2\ \alpha_2)^*; l_2; \Gamma_2; \phi_2$,
then $erase_S(S); erase_C(C) \vdash erase_{e^*}(e^*) : t_1^* \longrightarrow t_2^*$

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.

- $S; C \vdash \text{label}_n\{e_0^*\}\ e^*\ \text{end} : \epsilon; l_1; \Gamma_1; \phi_1 \longrightarrow (t_2\ \alpha_2)^n; l_2; \Gamma_2; \phi_2$
  We must show that $erase_S(S); erase_C(C) \vdash erase_{e^*}(e_0^*) : t_3^* \longrightarrow t_2^n$ and $erase_S(S); erase_C(C, \text{label}((t_3\ \alpha_3)^*;$ $erase_{e^*}(e^*) : \epsilon \longrightarrow t_3^*$ as they are the premises of typechecking a label block in Wasm.
  We have that $S; C \vdash e_0^* : (t_3\ \alpha_3)^*; l_3; \Gamma_3; \phi_3 \longrightarrow (t_2\ \alpha_2)^n; l_2; \Gamma_2; \phi_2$ 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 $erase_S(S); erase_C(C) \vdash erase_{e^*}(e_0^*) : t_3^* \longrightarrow t_2^n$
  $S; C, \text{label}((t_3\ \alpha_3)^*; l_3; \Gamma_3; \phi_3)) \vdash e^* : \epsilon; l_1; \Gamma_1; \phi_1 \longrightarrow (t_3\ \alpha_3)^*; l_3; \Gamma_3; \phi_3$, 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

$$erase_S(S); erase_C(C, \text{label}((t_3\ \alpha_3)^*; l_3; \Gamma_3; \phi_3)) \vdash erase_{e^*}(e^*) : \epsilon \longrightarrow t_3^*$$

- $S; C \vdash \mathsf{local}_n\{i; v^*\}\ e^*\ \mathsf{end} : \epsilon; l_1; \Gamma_1; \phi_1 \longrightarrow (t_2\ \alpha_2)^n; l_2; \Gamma_2; \phi_2$
  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 $erase_S(S); erase_C(S_{\text{inst}}(i), \text{local } t_v^*,\ \text{return}((t_2\ \alpha_2)^n; l_2; \Gamma_2; \phi_2)) \vdash$ $erase_{e^*}(e^*) : \epsilon \longrightarrow t_2^n$
  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 $S; S_{\text{inst}}(i), \text{local } t_v^*,\ \text{return}((t_2\ \alpha_2)^n; l_2; \Gamma_2; \phi_2)) \vdash e^*\ \mathsf{end} : \epsilon; l_1; \Gamma_1; \phi_1 \longrightarrow (t_2\ \alpha_2)^n; l_2; \Gamma_2;$
  Therefore, we have that $erase_S(S); erase_C(S_{\text{inst}}(i), \text{local } t_v^*,\ \text{return}((t_2\ \ \alpha_2)^n; l_2; \Gamma_2; \phi_2))\ \vdash$ $erase_{e^*}(e^*) : \epsilon \longrightarrow t_2^n$, by the inductive hypothesis of the well-typedness of e^*

□

We must prove safe erasure about the store s for use in Theorem 5. 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 15. $\boxed{erase_s(s) = \mathrm{s}}$

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

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

Lemma 13. 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,
$$erase_S(S) = \{\text{func } erase_{tfi}(tfi)^*,\ \text{global } tg^*,\ \text{table } n,\ \text{memory } n^?, \ldots\}$$
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 16. $\boxed{erase_{inst}(inst) = \text{inst}}$

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

We now prove that if a Wasm-precheck 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-precheck and Wasm). This will be useful for proving that a well-typed Wasm-precheck 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-precheck and Wasm).

Lemma 14. 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,

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

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 17. $\boxed{erase_S(S) = \mathrm{S}}$

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

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

Lemma 15. 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. □

C TYPE SAFETY PROOF

Before we jump into the type safety proof, we will first introduce several of the reasoning principles.

Lemma Well-Formedness reasons about the index variables in an instruction type. It says that every index variable used to represent a value on the stack or the value of a local variable must be present in the index environment Γ, and that the constraint set ϕ cannot refer to variables not in Γ.

Lemma 16. Well-Formedness
If $C \vdash e^* : ti_1^*; ti_3^*; \Gamma_1; \phi_1 \rightarrow ti_2^*; ti_4^*; \Gamma_2; \phi_2$,
(or $S; C \vdash e^* : ti_1^*; ti_3^*; \Gamma_1; \phi_1 \rightarrow ti_2^*; ti_4^*; \Gamma_2; \phi_2$)
then $(ti_1 \in \Gamma_1)^*$, $(ti_3 \in \Gamma_1)^*$, and $domain(\phi_1) \subset \Gamma_1$,
and $(ti_2 \in \Gamma_2)^*$, $(ti_4 \in \Gamma_4)^*$, and $domain(\phi_2) \subset \Gamma_2$,
and $\Gamma_2 \subseteq \Gamma_1$

Proof. Informally, no typing rule can produce a malformed instruction type, so all instruction types with a valid derivation must be well formed. Every rule that introduces fresh index variables in the stack, local index store, or constraint set ϕ also declares that index variable into Γ, unless the variable is already on the stack/in the local environment, in which case we can assume it is already in Γ. Similarly, functions must declare their local variables in Γ when typechecking their body.
Thus, this follows from induction over the typing judgment. □

(TODO: This and values-any-context should be able to replace threading-constraints in most if not all places)

Lemma 17. Strengthening
If $C \vdash e^* : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$,
and $\Gamma_1 \vdash [\Rightarrow \phi_3][\phi_1]$
then, $C \vdash e^* : ti_1^*; l_1; \Gamma_1; \phi_3 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_4$, for some ϕ_4
where $\Gamma_2 \vdash [\Rightarrow \phi_4][\phi_2]$

Proof. *Proof Sketch*: Intuitively, this proof follows from the fact that the only time ϕ_1 appears in a premise of a typing rule is on the left hand side of an implication assertion. Thus, since ϕ_3 implies ϕ_1, any ϕ implied by ϕ_1 is also implied by ϕ_3. Similarly, ϕ_2 is generally constructed syntactically by adding onto ϕ_1, so the where clause holds trivially. For the block rules, where ϕ_2 is a joinpoint of multiple paths, and not syntactically generated, each path will have stronger constraints, still implying the old path constraints, and therefore still implying ϕ_2.

The proof is by straightforward induction over typing derivations. □

Lemma Threading-Constraints is crucial for the inductive part of the proof; it allows us to add more index variables, and constraints on those variables, to the pre and post condition of an instruction type. The use case of this lemma is that when a program is evaluated inside a hole, there is no additional type information, but then we plug the reduced version of the program back into the hole and have to compose it with other instructions which may have more type information.

(TODO: Only when everything in Gamma is fresh!)

Lemma 18. Threading-Constraints

If $S; C \vdash e^* : ti_1^*; ti_3^*; \Gamma_1; \phi_1 \rightarrow ti_2^*; ti_4^*; \Gamma_2; \phi_2$,
and $domain(\phi) \subset \Gamma$
then $S; C \vdash e^* : ti_1^*; ti_3^*; \Gamma_1 \cup \Gamma; \phi_1 \cup \phi \rightarrow ti_2^*; ti_4^*; \Gamma_2 \cup \Gamma; \phi_2 \cup \phi$,

Proof. The idea is that if we have a constraint set and add more constraints to it, then the new constraint set must be stronger than the old one and imply the old one. In a little more detail, if $\Gamma \vdash \phi_1 \Rightarrow \phi_2$, then $\forall P.\Gamma \vdash \phi_1, P \Rightarrow \phi_2$, so if $\Gamma \vdash \phi_1 \Rightarrow \phi_2$, then $\forall \phi_3.\Gamma \vdash \phi_1, \phi_3 \Rightarrow \phi_2$. In effect, we are lifting common logic laws into the run-time type system.

Additionally, we can add in fresh index variable declarations to the index environments, because by Lemma Well-formedness those variable won't be referred to by the derivation for e^*.

The proof works by induction over typing derviations. □

Lemma Inversion-On-Instruction-Typing tells us what typing rules can apply to a given Wasm-precheck 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 $S; C \vdash t.\text{const } c : ti_1^*; l_1; \phi_1 \rightarrow ti_2^*; l_2; \phi_2$, 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 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 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.

Lemma 19. Inversion-On-Instruction-Typing

(1) If $S; C \vdash t.\text{const } c : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$, then $ti_2^* = ti_1^* \ (t\ \alpha)$, $l_1 = l_2$, $\Gamma_2 = \Gamma_1, (t\ \alpha)$, and $\phi_2 = \phi_1, (= \alpha\ (t\ c))$, for some $\alpha \notin \Gamma_1$.
(2) If $S; C \vdash t.binop : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$, then $ti_1^* = ti^*\ (t\ \alpha_1)\ (t\ \alpha_2)$, $ti_2^* = ti^*\ (t\ \alpha_3)$, $l_1 = l_2$, $\Gamma_2 = \Gamma_1, (t\ \alpha_3)$, and $\phi_2 = \phi_1, (= \alpha_3\ (binop\ \alpha_1\ \alpha_2))$, for some $\alpha_3 \notin \Gamma_1$.
(3) If $S; C \vdash t.testop : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$, then $ti_1^* = ti^*\ (t\ \alpha_1)$, $ti_2^* = ti^*\ (\text{i32}\ \alpha_2)$, $l_1 = l_2$, $\Gamma_2 = \Gamma_1, (\text{i32}\ \alpha_2)$, and $\phi_2 = \phi_1, (= \alpha_2\ (testop\ \alpha_1))$, for some $\alpha_2 \notin \Gamma_1$.
(4) If $S; C \vdash t.relop : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$, then $ti_1^* = ti^*\ (t\ \alpha_1)\ (t\ \alpha_2)$, $ti_2^* = ti^*\ (\text{i32}\ \alpha_3)$, $l_1 = l_2$, $\Gamma_2 = \Gamma_1, (\text{i32}\ \alpha_3)$, and $\phi_2 = \phi_1, (\text{i32}\ \alpha_3), (= \alpha_3\ (relop\ \alpha_1\ \alpha_2))$, for some $\alpha_3 \notin \Gamma_1$.
(5) If $S; C \vdash \text{nop} : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$, then $ti_1^* = ti_2^*$, $l_1 = l_2$, $\Gamma_1 = \Gamma_2$, and $\phi_1 = \phi_2$.

(6) If $S; C \vdash \mathsf{drop} : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$, then $ti_1^* = ti_0^*\ (t\ \alpha)$, $ti_2^* = ti_0^*$, $l_1 = l_2$, $\Gamma_1 = \Gamma_2$, and $\phi_1 = \phi_2$.

(7) If $S; C \vdash \mathsf{select} : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$, then $ti_1^* = ti_0^*\ (t\ \alpha_1)\ (t\ \alpha_2)\ (\mathsf{i32}\ \alpha_3)$, $ti_2^* = ti_0^*; (t\ \alpha)$, $l_1 = l_2$, $\Gamma_2 = \Gamma_1, (t\ \alpha)$, and $\phi_2 = \phi_1, (t\ \alpha), (\mathrm{if}(=\alpha_3\ (\mathsf{i32}\ 0))\ (=\alpha_1\ \alpha)\ (=\alpha_2\ \alpha))$, for some $\alpha \notin \Gamma_1$.

(8) If $S; C \vdash \mathsf{block}\ ((t_1^n \rightarrow t_2^m))e^*\ \mathsf{end} : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$, then
- $ti_1^* = ti_0^*\ (t_1\ \alpha_1)^n$
- $ti_2^* = ti_0^*\ (t_2\ \alpha_2)^m$
- $S; C, \mathrm{label}\ ((t_2\ \alpha_2)^m; l_2; \phi_2) \vdash e^* : (t_1\ \alpha_1)^n; l_1; \Gamma_1; \phi_1 \longrightarrow (t_2\ \alpha_2)^m; l_2; \Gamma_2; \phi_3$,
- $\Gamma_2 \vdash \phi_3 \Rightarrow \phi_2$

(9) If $S; C \vdash \mathsf{loop}\ ((t_1\ \alpha_3)^*; l_3; \phi_3 \longrightarrow (t_2\ \alpha_4)^*; l_4; \phi_4)\ e^*\ \mathsf{end} : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$, then $ti_1^* = ti_0^*\ (t_1\ \alpha_1)^*$, $ti_2^* = ti_0^*\ (t_2\ \alpha_2)^*$, $l_1 = (t_l\ \alpha_{l1})^*$, $l_3 = (t_l\ \alpha_{l3})^*$, $l_2 = (t_l\ \alpha_{l2})^*$, $l_4 = (t_l\ \alpha_{l4})^*$, $\Gamma_2 = \Gamma_1, (t_2\ \alpha_2)^*, (t_l\ \alpha_{l2})^*$, $\phi_2 = \phi_1 \cup \phi_4'$, $\phi_3' = \phi_3[\alpha_3 \mapsto \alpha_1]^*[\alpha_{l3} \mapsto \alpha_{l1}]^*$, $\phi_4' = \phi_4[\alpha_4 \mapsto \alpha_2]^*[\alpha_{l4} \mapsto \alpha_{l2}]^*$, $S; C, \mathrm{label}\ ((t_1\ \alpha_3)^*; l_3; \phi_3) \vdash e^* : (t_1\ \alpha_1)^*; l_1; \emptyset, (t_1\ \alpha_1)^*, l_1; \phi_3' \longrightarrow (t_2\ \alpha_2)^*; l_2; \Gamma_5; \phi_5$, $\Gamma_1 \vdash \phi_1 \Rightarrow \phi_3'$, and $\Gamma_5 \vdash \phi_5 \Rightarrow \phi_4'$.

(10) If $S; C \vdash \mathsf{if}\ ((t_1\ \alpha_3)^*; l_3; \phi_3 \longrightarrow (t_2\ \alpha_4)^*; l_4; \phi_4)\ e_1^*\ \mathsf{else}\ e_2^*\ \mathsf{end} : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$, then $ti_1^* = ti_0^*\ (t_1\ \alpha_1)^*\ (\mathsf{i32}\ \alpha)$, $ti_2^* = ti_0^*\ (t_2\ \alpha_4)^*$, $l_1 = (t_l\ \alpha_{l1})^*$, $l_3 = (t_l\ \alpha_{l3})^*$, $l_2 = (t_l\ \alpha_{l2})^*$, $l_4 = (t_l\ \alpha_{l4})^*$, $\Gamma_2 = \Gamma_1, (t_2\ \alpha_2)^* \cup (t_l\ \alpha_{l2})^*$, $\phi_2 = \phi_1 \cup \phi_4'$, $\phi_3' = \phi_3[\alpha_3 \mapsto \alpha_1]^*[\alpha_{l3} \mapsto \alpha_{l1}]^*$, $\phi_4' = \phi_4[\alpha_4 \mapsto \alpha_2]^*[\alpha_{l4} \mapsto \alpha_{l2}]^*$, $S; C, \mathrm{label}\ ((t_2\ \alpha_4)^*; l_4; \phi_4) \vdash e_1^* : (t_1\ \alpha_1)^*; l_1; \emptyset, (t_1\ \alpha_1)^*, l_1; \phi_3' \longrightarrow (t_2\ \alpha_2)^*; l_2; \Gamma_5; \phi_5, S; C, \mathrm{label}\ ((t_2\ \alpha_4)^*; l_4; \phi_4) \vdash e_2^* : (t_1\ \alpha_1)^*; l_1; \emptyset, (t_1\ \alpha_1)^*, l_1; \phi_3' \longrightarrow (t_2\ \alpha_2)^*; l_2; \Gamma_6; \phi_6$, $\Gamma_1 \vdash \phi_1 \Rightarrow \phi_3'$, $\Gamma_6 \vdash \phi_6 \Rightarrow \phi_4'$, and $\Gamma_5 \vdash \phi_5 \Rightarrow \phi_4'$.

(11) If $S; C \vdash \mathsf{label}_n\{e_0^*\}e^*\ \mathsf{end} : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$, then $ti_2^* = ti_1^*\ ti^*$, $S; C \vdash e_0^* : ti_3^*; l_3; \Gamma_3; \phi_3 \longrightarrow ti^*; l_2; \Gamma_4; \phi_4$, $S; C, \mathrm{label}\ (ti_3^*; l_3; \phi_3) \vdash e^* : \epsilon; l_1; \Gamma_1; \phi_1 \longrightarrow ti^*; l_2; \Gamma_5; \phi_5$, $ti_3^*, l_3 \notin \Gamma_1$, $\Gamma_4 \vdash \phi_4 \Rightarrow \phi_2$, and $\Gamma_5 \vdash \phi_5 \Rightarrow \phi_2$.

(12) If $S; C \vdash \mathsf{br}\ i : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$, then $ti_1^* = ti_0^*\ (t_1\ \alpha_1)^*$ and $l_1 = (t_{l1}\ \alpha_{l1})^*$, where $C_{\mathrm{label}}(i) = (t_1\ \alpha_2); (t_{l1}\ \alpha_{l2})^*; \phi_3$ and $\Gamma_1 \vdash \phi_1 \Rightarrow \phi_3[\alpha_2 \mapsto \alpha_1][\alpha_{l2} \mapsto \alpha_{l1}]$.

(13) If $S; C \vdash \mathsf{br\_if}\ i : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$, then $ti_1^* = ti_2^*\ (\mathsf{i32}\ \alpha)$, $l_2 = l_1$, $\Gamma_1 = \Gamma_2$, $\phi_2 = \phi_1, (=\alpha\ (\mathsf{i32}\ 0))$, $ti_1^* = (t_1\ \alpha_1)^*\ (\mathsf{i32}\ \alpha)$ and $l_1 = (t_{l1}\ \alpha_{l1})^*$, where $C_{\mathrm{label}}(i) = (t_1\ \alpha_2); (t_{l1}\ \alpha_{l2})^*; \phi_3$ and $\Gamma_1 \vdash \phi_1, \neg(=\alpha\ (\mathsf{i32}\ 0)) \Rightarrow \phi_3[\alpha_2 \mapsto \alpha_1][\alpha_{l2} \mapsto \alpha_{l1}]$.

(14) If $S; C \vdash \mathsf{br\_table}\ i^+ : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$, then $ti_1^* = ti_0^*\ (t_1\ \alpha_1)^*\ (\mathsf{i32}\ \alpha)$ and $l_1 = (t_{l1}\ \alpha_{l1})^*$, where $(C_{\mathrm{label}}(i) = (t_1\ \alpha_i); (t_{l1}\ \alpha_{li})^*; \phi_i)^+$, and $(\Gamma_1 \vdash \phi_1 \Rightarrow \phi_i[\alpha_i \mapsto \alpha_1][\alpha_{li} \mapsto \alpha_{l1}])^+$.

(15) If $S; C \vdash \mathsf{call}\ i : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$, then $ti_1^* = (t_1\ \alpha_1)^*$, $ti_2^* = (t_2\ \alpha_5)$, $l_1 = l_2$, $\Gamma_2 = \Gamma_1 \cup \Gamma_4$, $\phi_2 = \phi_1 \cup \phi_4[\alpha_4 \mapsto \alpha_5]^*$, where $\alpha_5^* \notin \Gamma_1$, $C_{\mathrm{func}}(i) = (t_1\ \alpha_3)^*; \phi_3 \longrightarrow (t_2\ \alpha_4)^*; \phi_4$, and $\Gamma_1 \vdash \phi_1 \Rightarrow \phi_3[\alpha_3 \mapsto \alpha_1]$.

(16) If $S; C \vdash \mathsf{call\_indirect}\ ((t_1\ \alpha_3)^*; \phi_3 \longrightarrow (t_2\ \alpha_4)^*; \phi_4) : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$, then $ti_1^* = (t_1\ \alpha_1)^*$, $ti_2^* = (t_2\ \alpha_5)$, $l_1 = l_2$, $\Gamma_2 = \Gamma_1 \cup \Gamma_4$, $\phi_2 = \phi_1 \cup \phi_4[\alpha_4 \mapsto \alpha_5]^*$, where $\alpha_5^* \notin \Gamma_1$, $C_{\mathrm{table}} = (j, (ti_6^*; \phi_6 \longrightarrow ti_7^*; \phi_7)^*)$, and $\Gamma_1 \vdash \phi_1 \Rightarrow \phi_3[\alpha_3 \mapsto \alpha_1]$.

(17) If $S; C \vdash \mathsf{call}\ cl : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$, then $ti_1^* = (t_1\ \alpha_1)^*$, $ti_2^* = (t_2\ \alpha_5)$, $l_1 = l_2$, $\Gamma_2 = \Gamma_1 \cup \Gamma_4$, $\phi_2 = \phi_1 \cup \phi_4[\alpha_4 \mapsto \alpha_5]^*$, where $\alpha_5^* \notin \Gamma_1$, $S \vdash cl : (t_1\ \alpha_3)^*; \phi_3 \longrightarrow (t_2\ \alpha_4)^*; \phi_4$, and $\Gamma_1 \vdash \phi_1 \Rightarrow \phi_3[\alpha_3 \mapsto \alpha_1]$.

(18) If $S; C \vdash \mathsf{local}_n\{i; v_l^*\}\ e^*\ \mathsf{end} : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$, then $ti_2^* = ti_1^*\ ti^n$, $l_1 = l_2$, $\Gamma_2 = \Gamma_1 \cup \Gamma_3$, $\phi_2 = \phi_1 \cup \phi_3$, and $S; (ti^n; l_3; \Gamma_3; \phi_3) \vdash_i v_l^*; e^* : ti^n; l_3; \Gamma_3; \phi_3$.

(19) If $S; C \vdash \mathsf{return} : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$, then $ti_1^* = ti_0^*\ (t_1\ \alpha_1)^*$, where $C_{\mathrm{return}} = (t_1\ \alpha_2); \phi_3$ and $\Gamma_1 \vdash \phi_1 \Rightarrow \phi_3[\alpha_2 \mapsto \alpha_1]$.

(20) If $S; C \vdash \mathsf{get\_local}\ i : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$, then $ti_2^* = ti_1^*\ (t\ \alpha)$, $l_1 = l_2$, $\Gamma_1 = \Gamma_2$, $\phi_1 = \phi_2$, $C_{\mathrm{local}}(i) = t$, and $l_1(i) = (t\ \alpha)$.

(21) If $S; C \vdash \text{set\_local } i : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$, then $ti_1^* = ti_2^*\ (t\ \alpha)$, $\Gamma_1 = \Gamma_2$, $\phi_1 = \phi_2$, $C_{\text{local}}(i) = t$, and $l_2 = l_1[i := (t\ \alpha)]$.
(22) If $S; C \vdash \text{tee\_local } i : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$, then $ti_1^* = ti_2^* = ti^*\ (t\ \alpha)$, $\Gamma_1 = \Gamma_2$, $\phi_1 = \phi_2$, $C_{\text{local}}(i) = t$, and $l_2 = l_1[i := (t\ \alpha)]$.
(23) If $S; C \vdash \text{get\_global } i : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$, then $ti_2^* = ti_1^*\ (t\ \alpha)$, $l_1 = l_2$, $\Gamma_2 = \Gamma_1, (t\ \alpha)$, $\phi_1 = \phi_2$, $C_{\text{global}}(i) = \text{mut}^?\ t$, and $\alpha \notin \Gamma_1$.
(24) If $S; C \vdash \text{set\_global } i : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$, then $ti_1^* = ti_2^*\ (t\ \alpha)$, $l_1 = l_2$, $\Gamma_1 = \Gamma_2$, $\phi_1 = \phi_2$, and $C_{\text{global}}(i) = \text{mut}\ t$.
(25) If $S; C \vdash t.\text{load}\ (tp\_sx)^?\ align\ o : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$, then $ti_1^* = ti^*\ (\text{i32}\ \alpha_1)$, $ti_2^* = ti^*\ (t\ \alpha_2)$, $l_1 = l_2$, $\Gamma_2 = \Gamma_1, (t\ \alpha_2)$, $\phi_1 = \phi_2$, $C_{\text{memory}} = n$, $2^{align} \le (|tp| <)^?|t|$, and $\alpha_2 \notin \Gamma_1$.
(26) If $S; C \vdash t.\text{store}\ tp^?\ align\ o : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$, then $ti_1^* = ti_2^*\ (\text{i32}\ \alpha_1)\ (t\ \alpha_2)$, $l_1 = l_2$, $\Gamma_1 = \Gamma_2$, $\phi_1 = \phi_2$, $C_{\text{memory}} = n$, and $2^{align} \le (|tp| <)^?|t|$.
(27) If $S; C \vdash \text{current\_memory} : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$,
then $ti_2^* = ti_1^*\ (\text{i32}\ \alpha)$, $l_1 = l_2$, $\Gamma_2 = \Gamma_1, (\text{i32}\ \alpha)$, $\phi_1 = \phi_2$, $C_{\text{memory}} = n$, and $\alpha \notin \Gamma_1$.
(28) If $S; C \vdash \text{grow\_memory} : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$, then $ti_1^* = ti^*\ (\text{i32}\ \alpha_1)$, $ti_2^* = ti^*\ (\text{i32}\ \alpha_2)$, $l_1 = l_2$, $\Gamma_2 = \Gamma_1, (\text{i32}\ \alpha_2)$, $\phi_1 = \phi_2$, $C_{\text{memory}} = n$, and $\alpha_2 \notin \Gamma_1$.
(29) If $S; C \vdash e_1^*\ e_2 : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_3^*; l_3; \Gamma_3; \phi_3$, then $S; C \vdash e_1^* : ti_0^*\ ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_1'^*; l_1'; \Gamma_1'; \phi_1'$, and $S; C \vdash e_2 : ti_1'^*; l_1'; \Gamma_1'; \phi_1' \longrightarrow ti_0^*\ ti_2^*; l_2; \Gamma_2; \phi_2$.

Proof. Follows from straightforward induction over typing derivations. □

(TODO: Shiny new corollaries so these principles have names and are explicitly stated somewhere.)

Corollary 6. *Type-of-Values*
If $S; C \vdash (t.\text{const}\ c)^n : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$, then $ti_2^* = ti_1^*(t\ \alpha)^*$, $l_2 = l_1$, $\Gamma_2^* = \Gamma_1, (t\ \alpha)^*$, $\phi_2 = \phi_1, (= \alpha (t\ c))^*$.

Proof. Follows from straightforward induction over the typing derivations and Lemma Inversion-On-Instruction-Typing. □

Corollary 7. *Values-Any-Context*
If $S; C \vdash (t.\text{const}\ c)^n : ti^*; l_1; \Gamma; \phi \longrightarrow ti_1^*\ (t\ \alpha)^*; l_2; \Gamma, (t\ \alpha)^*; \phi, (=\ \alpha\ (t\ c))$ then $\hat{S}; \hat{C} \vdash (t.\text{const}\ c)^n : \hat{ti}^*; l_1; \hat{\Gamma}; \phi_1 \longrightarrow ti_1^*\ (t\ \alpha)^*; l_2; \hat{\Gamma}, (t\ \alpha)^*; \hat{\phi}, (=\ \alpha\ (t\ c))$ for all $\hat{S}$, $\hat{C}$, $\hat{\Gamma}$, and $\hat{\phi}$.

Proof. Follows from straightforward induction over the typing derivations, Lemma Inversion-On-Instruction-Typing, Rule Const, and Rule Composition. □

Corollary 8. *Sequence-Composition*
If $S; C \vdash e_1^* : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$, and $S; C \vdash e_2^* : ti_2^*; l_2; \Gamma_2; \phi_2 \longrightarrow ti_3^*; l_3; \Gamma_3; \phi_3$.

Proof. Follows from straightforward induction over the typing derivations, Lemma Inversion-On-Instruction-Typing, and Rule Composition. □

Corollary 9. *Sequence-Decomposition*
If $S; C \vdash e_1^*\ e_2^* : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$, and $S; C \vdash e_1^* : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_3^*; l_3; \Gamma_3; \phi_3$, then $S; C \vdash e_2^* : ti_3^*; l_3; \Gamma_3; \phi_3 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$

Proof. Follows from straightforward induction over the typing derivations, Lemma Inversion-On-Instruction-Typing, and Rule Composition. □

Corollary 10. *Consts-In-Br*
If $S; C \vdash e_1^* : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$, and $S; C \vdash e_2^* : ti_2^*; l_2; \Gamma_2; \phi_2 \longrightarrow ti_3^*; l_3; \Gamma_3; \phi_3$.

Proof. Follows from straightforward induction over the typing derivations, Lemma Inversion-On-Instruction-Typing, and Rule Composition. □

Lemma 20. Erasure-Same-Semantics
If $\vdash s; v^*; e^* : ti_2^*; l_2; \Gamma_2; \phi_2$,
then $erase_s(s); v^*; erase_{e^*}(e^*) \hookrightarrow_i erase_s(s'); v'^*; erase_{e^*}(e'^*)$ for some s' and e'^*.

Proof. The intuition for this is that the reduction semantics are syntactically the same except for the representation of types. Therefore, we can relate the semantics of a Wasm-precheck program to the Wasm semantics by erasing the Wasm-precheck program to a Wasm program, reducing under the Wasm semantics, and then adding back the Wasm-precheck types.
The proof follows then by induction over the typing derivation. □

(TODO: Update text) The next lemma, Lemma Values-Br-In-Context, shows that if a sequence of constants, v^n, has a certain postcondition within a nested context, L^k, then it has the same postcondition outside of that context with the precondition of the context extended with equality constraints on fresh variables. We use this rule for branching and returning when we have some values v^n inside a reduction context L^k.

Lemma 21. Values-Br-In-Context
If $S; C \vdash L^k[(t.\mathsf{const}\ c)^n \mathsf{br}\ j] : ti_1^*; l_1; \Gamma_1; phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$
where $C_{\text{label}}(j-k) = (ti_3^n; l_3; \phi_3)$
where ti_3^n and l_3 are entirely fresh (no variable in ti_3^n or l_3 appears in Γ_1 or Γ_2), which we can safely assume through alpha-renaming,
then $S; C' \vdash (t.\mathsf{const}\ c)^n : ti_1^*; l_1; \Gamma_1, (t_0\ \alpha_0)^*; \phi_1, (=\ \alpha_0\ (t_0\ c_0))^* \rightarrow ti_1^*\ ti_3^n; l_1; \Gamma_1, (t_0\ \alpha_0)^*, ti_3^n; \phi_1, (=\ \alpha_0\ (t_0\ c_0))^*, (=\ \alpha_3\ (t\ c))^*$
where $\Gamma_1, (t_0\ \alpha_0)^*, ti_3^n \vdash \phi_1, (=\ \alpha_0\ (t_0\ c_0))^*, (=\ \alpha_3\ (t\ c))^* \Rightarrow \phi_3[l_3 \mapsto l_1]$
where $ti_3^n = (t\ \alpha_3)^n$

Proof. By induction on k.

- Base case: $k = 0$

(1) Expanding L^0, we have $S; C \vdash (t_0.\mathsf{const}\ c_0)^*\ (t.\mathsf{const}\ c)^n\ <br>\ j\ e^* : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$ for some $(t_0.\mathsf{const}\ c_0)^*$ and e^*

(2) By Lemma Sequence-Decomposition on 1, we know that the following hold for some ti_4^*, l_4, Γ_4, and ϕ_4

  (a) $S; C \vdash (t_0.\mathsf{const}\ c_0)^* : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_4^*; l_4; \Gamma_4; \phi_4$

  (b) $S; C \vdash (t.\mathsf{const}\ c)^n\ <br>\ j\ e^* : ti_4^*; l_4; \Gamma_4; \phi_4 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$

(3) By Lemma Type-Of-Values on 2a, we know that $ti_4^* = ti_1^*\ (t_0\ \alpha_0)^*$, $l_4 = l_1$, $\Gamma_4 = \Gamma_1, (t_0\ \alpha_0)^*$, and $\phi_4 = \phi_1, (=\ \alpha_0\ (t_0\ c_0))^*$

(4) Therefore, $S; C \vdash (t.\mathsf{const}\ c)^n\ <br>\ j\ e^* : ti_1^*\ (t_0\ \alpha_0)^*; l_1; \Gamma_1, (t_1\ \alpha_0)^*; \phi_1, (=\ \alpha_0\ (t_0\ c_0))^* \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$

(5) By Lemma Sequence-Decomposition on 4, we have $S; C \vdash (t.\mathsf{const}\ c)^n\ <\ br\ >\ j : ti_1^*\ (t_1\ \alpha_1)^*; l_1; \Gamma_1, (t_0\ \alpha_0)^*; \phi_1, (=\ \alpha_0\ (t_0\ c_0))^* \rightarrow ti_5^*; l_5; \Gamma_5; \phi_5$ for some ti_5^*, l_5, Γ_5, and ϕ_5.

(6) By Lemma Inversion-On-Instruction-Typing on $S; C \vdash (t.\mathsf{const}\ c)^n\ <br>\ j : ti_1^*\ (t_0\ \alpha_0)^*; l_1; \Gamma_1, (t_0\ \alpha_0$
$\alpha_0\ (t_0\ c_0))^* \rightarrow ti_5^*; l_5; \Gamma_5; \phi_5$, we have that

  (a) $S; C \vdash (t.\mathsf{const}\ c)^n : ti_1^*\ (t_0\ \alpha_0)^*; l_1; \Gamma_1, (t_0\ \alpha_0)^*; \phi_1, (=\ \alpha_0\ (t_0\ c_0))^* \rightarrow ti_6^*; l_6; \Gamma_6; \phi_6$ for some ti_6^*, l_6, Γ_6, and ϕ_6

  (b) $S; C \vdash \mathsf{br}\ j : ti_6^*; l_6; \Gamma_6; \phi_6 \rightarrow ti_5^*; l_5; \Gamma_5; \phi_5$

(7) By Lemma Type-Of-Values on 4a, we know that $ti_6^* = ti_1^*\ (t_0\ \alpha_0)^*\ (t\ \alpha_6)^n$, $l_6 = l_1$, $\Gamma_6 = \Gamma_1, (t_0\ \alpha_0)^*(t\ \alpha_6)^n$, and $\phi_6 = \phi_1, (=\ \alpha_0\ (t_0\ c_0))^*, (=\ \alpha_6\ (t\ c))^*$

(8) By Lemma Inversion-On-Instruction-Typing on 4b, we know that $(t = t_3)^n$, where $ti_3^n = ((t_3\ \alpha_3))^n$, $l_1 = (t_{l3}\ \alpha_{l1})^*$, where $l_3 = (t_{l3}\ \alpha_{l3})^*$ and $\Gamma_0, (t_0\ \alpha_0)^* \vdash \phi_1, (=\ \alpha_0\ (t_0\ c_0))^*, (=\ \alpha_6\ (t\ c))^* \Rightarrow \phi_3[\alpha_3 \mapsto \alpha_6][\alpha_{l3} \mapsto \alpha_{l6}]$

(9) Then, by combining Lemma Values-Any-Context, 4a, 5, and 6, and since ti_3^n are fresh, we have that $S; C' \vdash (t.\mathsf{const}\ c)^n : ti_1^*\ (t_0\ \alpha_0)^*; l_3; \Gamma_1, (t_0\ \alpha_0)^*; \phi_1, (=\ \alpha_0\ (t_0\ c_0))^* \rightarrow ti_1^*\ (t_1\ \alpha_1)^*\ ti_3^n; l_3; \Gamma_1, (t_1\ \alpha_1)^*; \phi_1, (=\ \alpha_1\ (t_1\ c_1))^*, (=\ \alpha_3\ (t\ c))^*$

(10) Finally, by 6, and because ti_3^n is fresh, $\Gamma_1, (t_1\ \alpha_1)^*, ti_3^n \vdash \phi_1, (=\ \alpha_1\ (t_1\ c_1))^*, (=\ \alpha_3\ (t\ c))^* \Rightarrow \phi_3[\alpha_{l3} \mapsto \alpha_{l6}]$ (here we are essentially moving the renaming $[\alpha_3 \mapsto \alpha_6]$ to the left-hand-side of the implication, which is safe because we control what the names are at this point)

- Inductive case: $k > 0$

  This proof case is simpler than above as we only have to reason about br in the base case, so in the inductive case the inductive hypothesis handles it for us.

(1) Expanding L^k, we have $S; C \vdash (t_0.\mathsf{const}\ c_0)^*\mathsf{label}\{e_0^*\}\ L^{k-1}[(t.\mathsf{const}\ c)^n\ < br > j]\ \mathsf{end}e^* : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$ for some $(t_0.\mathsf{const}\ c_0)^*$, e_0^*, and e^*

(2) By Lemma Sequence-Decomposition on 1, we know that the following hold for some ti_4^*, l_4, Γ_4, and ϕ_4
   (a) $S; C \vdash (t_0.\mathsf{const}\ c_0)^* : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_4^*; l_4; \Gamma_4; \phi_4$
   (b) $S; C \vdash \mathsf{label}\{e_0^*\}\ L^{k-1}[(t.\mathsf{const}\ c)^n\ < br > j]\ \mathsf{end}e^* : ti_4^*; l_4; \Gamma_4; \phi_4 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$

(3) By Lemma Type-Of-Values on 2a, we know that $ti_4^* = ti_1^*\ (t_0\ \alpha_0)^*$, $l_4 = l_1$, $\Gamma_4 = \Gamma_1, (t_0\ \alpha_0)^*$, and $\phi_4 = \phi_1, (=\ \alpha_0\ (t_0\ c_0))^*$

(4) Therefore, $S; C \vdash \mathsf{label}\{e_0^*\}\ L^{k-1}[(t.\mathsf{const}\ c)^n\ < br > j]\ \mathsf{end}e^* : ti_1^*\ (t_0\ \alpha_0)^*; l_1; \Gamma_1, (t_1\ \alpha_0)^*; \phi_1, (=\ \alpha_0\ (t_0\ c_0))^* \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$

(5) By Lemma Sequence-Decomposition on 4, we have $S; C \vdash \mathsf{label}\{e_0^*\}\ L^{k-1}[(t.\mathsf{const}\ c)^n\ < br > j]\ \mathsf{end}e^* : ti_1^*\ (t_0\ \alpha_0)^*; l_1; \Gamma_1, (t_0\ \alpha_0)^*; \phi_1, (=\ \alpha_0\ (t_0\ c_0))^* \rightarrow ti_5^*; l_5; \Gamma_5; \phi_5$ for some ti_5^*, l_5, Γ_5, and ϕ_5.

(6) By Lemma Inversion-On-Instruction-Typing on 5, we have that $S; C, \mathrm{label}(ti^*; l; \phi) \vdash L^{k-1}[(t.\mathsf{const}\ c)^n\ < br > j]\ \mathsf{end}e^* : ti_1^*\ (t_0\ \alpha_0)^*; l_1; \Gamma_1, (t_0\ \alpha_0)^*; \phi_1, (=\ \alpha_0\ (t_0\ c_0))^* \rightarrow ti_5^*; l_5; \Gamma_5; \phi_5$

(7) Then, by the inductive hypothesis on 6, we have that $S; C, \mathrm{label}(ti^*; l; \phi) \vdash (t.\mathsf{const}\ c)^n : ti_1^*\ (t_0\ \alpha_0)^*; l_3; \Gamma_1, (t_0\ \alpha_0)^*, (t'\ \alpha')^*; \phi_1, (=\ \alpha_0\ (t_0\ c_0))^*, (=\ \alpha'\ (t'\ c'))^* \rightarrow ti_1^*\ (t_0\ \alpha_0)^*\ ti_3^n; l_3; \Gamma_1, (t_0\ \alpha_0)$
$\alpha_0\ (t_0\ c_0))^*, (=\ \alpha'\ (t'\ c'))^*, (=\ \alpha_3\ (t\ c))^*$, where $\Gamma_1, (t_0\ \alpha_0)^*, (t'\ \alpha')^*, ti_3^n \vdash \phi_1, (=\ \alpha_0\ (t_0\ c_0))^*, (=\ \alpha'\ (t'\ c'))^*, (=\ \alpha_3\ (t\ c))^* \Rightarrow \phi_3[l_3 \mapsto l_1]$, for some t'^*, c'^*, and α'^*.

(8) Finally, by Lemma Values-Any-Context on 7, we have $S; C' \vdash (t.\mathsf{const}\ c)^n : ti_1^*\ (t_0\ \alpha_0)^*; l_1; \Gamma_1, (t_0\ \alpha_0)^*, ($
$\alpha_0\ (t_0\ c_0))^*, (=\ \alpha\ (t\ c))^* \rightarrow ti_1^*\ (t_0\ \alpha_0)^*\ ti_3^n; l_1; \Gamma_1, (t_0\ \alpha_0)^*, (t'\ \alpha')^*, ti_3^n; \phi_1, (=\ \alpha_0\ (t_0\ c_0))^*, (=\ \alpha'\ (t'\ c'))^*, (=\ \alpha_3\ (t\ c))^*$

□

Lemma 22. Values-Return-In-Context
If $S; C \vdash L^k[(t.\mathsf{const}\ c)^n\mathsf{return}] : ti_1^*; l_1; \Gamma_1; phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$
where $C_{\mathrm{return}} = (ti_3^n; \phi_3)$
where ti_3^n are entirely fresh (no variable in ti_3^n appears in Γ_1 or Γ_2), which we can safely assume through alpha-renaming,
then $S; C' \vdash (t.\mathsf{const}\ c)^n : ti_1^*; l_1; \Gamma_1, (t_0\ \alpha_0)^*; \phi_1, (=\ \alpha_0\ (t_0\ c_0))^* \rightarrow ti_1^*\ ti_3^n; l_1; \Gamma_1, (t_0\ \alpha_0)^*, ti_3^n; \phi_3$

Proof. Similar to Lemma Values-Br-In-Context. □

Note: while we parameterize Wasm-precheck over implication, since we start with an under-approximation of implication, we can assume that implication is sound.

C.1 Subject Reduction Lemmas and Proofs

Lemma 23 (Subject Reduction for Programs). If $\vdash s; v^*; e^* : ti^*; l; \Gamma; \phi$,
and $s; v^*; e^* \hookrightarrow s'; v'^*; e'^*$,
then $\vdash s'; v'^*; e'^* : ti^*; l; \Gamma; \phi$

Proof.

(1) Because $\vdash_i s; v^*; e^* : ti^*; l; \Gamma; \phi$, we know that for some S
  (a) $\vdash s : S$
  (b) $S; \epsilon \vdash_i v^*; e^* : ti^*; l; \Gamma; \phi$
  by inversion on Rule Program.
(2) by inversion on Rule Code and 1b, for some α^*, t^*, c^*, we have that
  (a) $(\vdash v : (t_v\ \alpha_v); \phi_v)^*$
  (b) $S; S_{\text{inst}}(i), \text{local } (t_v^*) \vdash e^* : \epsilon; (t_v\ \alpha_v)^*; \emptyset, (t_v\ \alpha_v)^*, (t\ \alpha)^*; \phi_v^*, (= \alpha\ (t\ c))^* \longrightarrow ti^*; l; \Gamma; \phi_2$
  (c) $\Gamma \vdash \phi_2 \Rightarrow \phi$
(3) By Lemma 24 on 1a, 2a, 2b, and Lemma Well-Formedness, we have
  (a) $S; S_{\text{inst}}(i), \text{local } (t_v^*) \vdash e'^* : \epsilon; (t_v\ \alpha_v)^*; \emptyset, (t_v\ \alpha_v)^*, (t\ \alpha)^*; \phi_v^*, (= \alpha\ (t\ c))^* \longrightarrow ti^*; l; \Gamma; \phi_2$
  (b) $\vdash \epsilon; (t_v\ \alpha_v)^*; \emptyset, (t_v\ \alpha_v)^*, (t\ \alpha)^*; \phi_v^*, (= \alpha\ (t\ c))^* \longrightarrow ti^*; l; \Gamma; \phi_2$
  (c) $\vdash s' : S$ for some s'
  (d) $(\vdash v' : (t_v\ \alpha_v'); \phi_v')^*$, where $l_2 = (t_v\ \alpha_v')$ and $\phi_3 = \phi_1 \bigcup \phi_v'^*$, $\alpha^* \notin \Gamma_3$ for some v'^*
  (e) $\alpha^* \notin \Gamma_1$
  (f) $\Gamma_4 \vdash \phi_4 \Rightarrow \phi_2$
(4) By Rule Code, 3a, and 3d, $S; \epsilon \vdash_i v'^*; e'^* : ti^*; l; \Gamma; \phi$.
(5) by Rule Program, 4, and 3c, $\vdash_i s'; v'^*; e'^* : ti^*; l; \Gamma; \phi$

□

Lemma 24 (Subject Reduction for Instructions). If $S; C \vdash e^* : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$ and $s; v^*; e^* \hookrightarrow s'; v'^*; e'^*$ (we may omit s and v^* on rules that don't refer to them), where

Assumption 1) $\vdash ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$
Assumption 2) $\vdash s : S$, where $C = S_{\text{inst}}(i)$
Assumption 3) $(\vdash v : (t_v\ \alpha_v); \phi_v)^*$, where $l_1 = (t_v\ \alpha_v)^*$ and $\phi_v \subset \phi_1$

then $S; C \vdash e'^* : ti_1^*; l_3; \Gamma_1, (t\ \alpha)^*; \phi_3, (= \alpha\ (t\ c))^* \longrightarrow ti_2^*; l_2; \Gamma_4; \phi_4$ for some α^*, t^*, and c^* where

Subgoal 1) $\vdash ti_1^*; l_3; \Gamma_1, (t\ \alpha)^*; \phi_3, (= \alpha\ (t\ c))^* \longrightarrow ti_2^*; l_2; \Gamma_4; \phi_4$,
Subgoal 2) $\vdash s' : S$ (implicitly true and omitted when $s' = s$),
Subgoal 3) $(\vdash v' : (t_v\ \alpha_v'); \phi_v')^*$, $l_3 = (t_v'\ \alpha_v')^*$,
Subgoal 4) $\phi_3 = \phi_1 \bigcup \phi_v'^*$ for some v'^* (implicitly true and omitted when $v'^* = v^*$ and $l_1 = l_3$, or when $\phi_3 = \phi_1[\alpha_v \mapsto \alpha_v']$),
Subgoal 5) $\alpha^* \notin \Gamma_1$ (ensures α^* don't appear in the typing derivation for e'^*, according to Lemma Well-formedness),
Subgoal 6) and $\Gamma_4 \vdash \phi_4 \Rightarrow \phi_2$ (implicitly true and omitted when $\phi_4 = \phi_2$ and $\Gamma_2 \subseteq \Gamma_4$)

Proof. By case analysis on the reduction rules.

- $S; C \vdash L^0[\text{trap}] : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$
  $\wedge\ L^0[\text{trap}] \hookrightarrow \text{trap}$
  This case is trivial since trap accepts any precondition and postcondition. Thus, $S; C \vdash \text{trap} : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$ by Rule Trap, where: subgoal 1 follows from assumption 4;

subgoals 2, 4, and 6 trivially hold; subgoal 3 follows from assumption 2, and subgoal 5 holds since $\alpha^* = \epsilon$.

- $S; C \vdash (t.\text{const}\ c_1)\ (t.\text{const}\ c_2)\ t.binop : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$
  $\wedge\ (t.\text{const}\ c_1)\ (t.\text{const}\ c_2)\ t.binop \hookrightarrow t.\text{const}\ c$ where $c = binop(c_1, c_2)$
  We begin by reasoning about the type of the original instructions $(t.\text{const}\ c_1)\ (t.\text{const}\ c_2)\ t.binop$
  By Lemma Inversion-On-Instruction-Typing on $S; C \vdash (t.\text{const}\ c_1)\ (t.\text{const}\ c_2)\ t.binop : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$, we know that $\Gamma_2 = \Gamma_1, (t\ \alpha_1), (t\ \alpha_2), (t\ \alpha_3)$ where $\alpha_1, \alpha_2, \alpha_3 \notin \Gamma_1$, $ti_2^* = ti_1^*\ (t\ \alpha_3)$, $l_2 = l_1$, and $\phi_2 = \phi_1, (=\alpha_1\ (t\ c_1)), (=\alpha_2\ (t\ c_2)), (=\alpha_3\ (\|binop\|\ \alpha_1\ \alpha_2))$
  Now we know that we want to show that

  $$\begin{aligned}&S; C \vdash t.\text{const}\ c : ti_1^*; l_1; \Gamma_1, (t\ \alpha_1), (t\ \alpha_2); \phi_1, (=\alpha_1\ (t\ c_1)), (=\alpha_2\ (t\ c_2))\\ &\longrightarrow ti_1^*\ (t\ \alpha_3); l_1; \Gamma_1, (t\ \alpha_1), (t\ \alpha_2), (t\ \alpha_3); \phi_1, (=\alpha_1\ (t\ c_1)), (=\alpha_2\ (t\ c_2)), (=\alpha_3\ c)\end{aligned}$$

  where $(t\ \alpha_3); l_1 \subset \Gamma_1, (t\ \alpha_1), (t\ \alpha_2)$ and $\Gamma_1, (t\ \alpha_1), (t\ \alpha_2), (t\ \alpha_3) \vdash \phi_3 \Rightarrow (\phi_1, (=\alpha_1\ (t\ c_1)), (=\alpha_2\ (t\ c_2)), (=\alpha_3\ (\|binop\|\ \alpha_1\ \alpha_2)))$
  Now we show that $t.\text{const}\ c$ has the appropriate type.
  By Rule Const,

$$\begin{aligned}S; C \vdash t.\text{const}\ c : \epsilon; l_1; \Gamma_1, (t\ \alpha_1), (t\ \alpha_2); \phi_1, (=\alpha_1\ (t\ c_1)), (=\alpha_2\ (t\ c_2))\\ \longrightarrow (t\ \alpha_3); l_1; \Gamma_1, (t\ \alpha_1), (t\ \alpha_2), (t\ \alpha_3); \phi_1, (=\alpha_1\ (t\ c_1)), (=\alpha_2\ (t\ c_2)), (=\alpha_3\ c)\end{aligned}$$

  since $\alpha_3 \notin \Gamma_1, (=\alpha_1\ (t\ c_1)), (=\alpha_2\ (t\ c_2))$.
  Then, by Rule Stack-Poly,

$$\begin{aligned}S; C \vdash t.\text{const}\ c : ti_1^*; l_1; \Gamma_1, (t\ \alpha_1), (t\ \alpha_2); \phi_1, (=\alpha_1\ (t\ c_1)), (=\alpha_2\ (t\ c_2))\\ \longrightarrow ti_1^*\ (t\ \alpha_3); l_1; \Gamma_1, (t\ \alpha_1), (t\ \alpha_2), (t\ \alpha_3); \phi_1, (=\alpha_1\ (t\ c_1)), (=\alpha_2\ (t\ c_2)), (=\alpha_3\ c)\end{aligned}$$

  By assumption 1, $l_1 \subset \Gamma_1$, so $(t\ \alpha_3), l_1 \subset \Gamma_1, (=\alpha_1\ (t\ c_1)), (=\alpha_2\ (t\ c_2)), (t\ \alpha_3)$.
  Further, since $c = binop(c_1, c_2)$, then $\Gamma_1, (t\ \alpha_1), (t\ \alpha_2), (t\ \alpha_3) \vdash \phi_1, (=\alpha_1\ (t\ c_1)), (=\alpha_2\ (t\ c_2)), (=\alpha_3\ (t\ c)) \Rightarrow \phi_1, (=\alpha_1\ (t\ c_1)), (=\alpha_2\ (t\ c_2)), (=\alpha_3\ (\|binop\|\ \alpha_1\ \alpha_2))$.

- $S; C \vdash (t.\text{const}\ c_1)\ (t.\text{const}\ c_2)\ t.\text{div}\checkmark : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$
  $\wedge\ (t.\text{const}\ c_1)\ (t.\text{const}\ c_2)\ t.\text{div}\checkmark \hookrightarrow t.\text{const}\ c$ where $c = div(c_1, c_2)$
  Same as above.

- $C \vdash (t.\text{const}\ c_1)\ (t.\text{const}\ c_2)\ t.binop : \epsilon; l_1; \Gamma_1\phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$
  $\wedge\ (t.\text{const}\ c_1)\ (t.\text{const}\ c_2)\ t.binop \hookrightarrow \text{trap}$
  This case is trivial since trap accepts any precondition and postcondition. Thus, $S; C \vdash \text{trap} : \epsilon; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$ by Rule Trap.

- $S; C \vdash (t.\text{const}\ c_1)\ t.testop : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$
  $\wedge\ (\text{i32.const}\ c)\ t.testop \hookrightarrow \text{i32.const}\ c_2$ where $c_2 = testop(c)$
  We begin by reasoning about the type of the original instructions $(t.\text{const}\ c_1)\ t.testop$
  By Lemma Inversion-On-Instruction-Typing on $S; C \vdash (t.\text{const}\ c_1)\ t.testop : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$, we know that $ti_2^* = ti_1^*\ (\text{i32}\ \alpha_2)$, $\Gamma_2 = \Gamma_1, (t\ \alpha_1), (\text{i32}\ \alpha_2)$ where $\alpha_1, \alpha_2 \notin \Gamma_1$, $ti_2^* = (\text{i32}\ \alpha_2)$, $l_2 = l_1$, and $\phi_2 = \phi_1, (=\alpha_1\ (t\ c_1)), (=\alpha_2\ (testop\ \alpha_1))$
  Now we must show that

$$S; C \vdash \text{i32.const}\ c : ti_1^*; l_1; \Gamma_1, (t\ \alpha_1); \phi_1, (=\alpha_1\ (t\ c_1)) \longrightarrow ti_1^*\ (t\ \alpha_2); l_1; \Gamma_1, (t\ \alpha_1), (\text{i32}\ \alpha_2); \phi_3$$

  where $ti^*; l \subset \Gamma_3$ and $\Gamma_1, (t\ \alpha_1), (t\ \alpha_2) \vdash \phi_3 \Rightarrow \phi_1, (=\alpha_1\ (t\ c_1)), (=\alpha_2\ (testop\ \alpha_1))$
  We show that i32.const c has the appropriate type.

By Rule Const, $S; C \vdash \text{i32.const } c : \epsilon; l_1; \Gamma_1, (t\ \alpha_1); \phi_1 \rightarrow (\text{i32}\ \alpha_2); l_1; \Gamma_1, (t\ \alpha_1), (\text{i32}\ \alpha_2); \phi_1, (= \alpha_1\ (t\ c_1)), (= \alpha_2\ (t\ c))$.
Then, $S; C \vdash \text{i32.const } c : ti_1^*; l_1; \Gamma_1, (t\ \alpha_1); \phi_1 \rightarrow ti_1^*\ (\text{i32}\ \alpha_2); l_1; \Gamma_1, (t\ \alpha_1), (\text{i32}\ \alpha_2); \phi_1, (= \alpha_1\ (t\ c_1)), (= \alpha_2\ (t\ c))$ by Rule Stack-Poly.
We have $\Gamma_3 = \Gamma_1, (t\ \alpha_1), (\text{i32}\ \alpha_2)$, and by Lemma Well-formedness, $l_1 \subset \Gamma_1$, so $(\text{i32}\ \alpha_2), l_1 \subset \Gamma_1, (t\ \alpha_1), (\text{i32}\ \alpha_2)$.
Further, since $c_2 = testop(c_1)$, then $\Gamma_1, (t\ \alpha_1), (t\ \alpha_2) \vdash \phi_1, (= \alpha_1\ (t\ c_1)), (= \alpha_2\ (t\ c_2)) \Rightarrow \phi_1, (= \alpha_1\ (t\ c_1)), (= \alpha_2\ (\|testop\|\ (t\ c_2)))$.

- $S; C \vdash (t.\text{const } c_1)\ (t.\text{const } c_2)\ t.relop : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2\phi_2$
  $\wedge\ (\text{i32.const } c_1)\ (t.\text{const } c_2)\ t.relop \hookrightarrow t.\text{const } c$ where $c = relop(c_1, c_2)$
  This case is identical to the $(t.\text{const } c_1)\ (t.\text{const } c_2)\ t.binop \hookrightarrow t.\text{const } c$ case, except that *binop* is replaced with *relop*, and the result type is replaced with i32.

- $S; C \vdash \text{unreachable} : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$
  $\wedge$ unreachable $\hookrightarrow$ trap
  This case is trivial since trap accepts any precondition and postcondition. Thus, $S; C \vdash \text{trap} : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$ by *trap*.

- $S; C \vdash \text{nop} : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$
  $\wedge$ nop $\hookrightarrow \epsilon$
  By Lemma Inversion-On-Instruction-Typing on Rule Nop, we know that $ti_2^* = ti_1^*$, $l_2 = l_1$, $\Gamma_2 = \Gamma_1$ and $\phi_2 = \phi_1$.
  Then, we want to show that $S; C \vdash \epsilon : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_3; \phi_3$, where $ti^*; l \subset \Gamma_3$ and $\Gamma_3 \vdash \phi_3 \Rightarrow \phi_1$
  Then, $S; C \vdash \epsilon : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_1^*; l_1; \Gamma_1; \phi_1$ by Rule Empty and Rule Stack-Poly.

- $S; C \vdash (t.\text{const } c)\ \text{drop} : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$
  $\wedge\ (t.\text{const } c)\ \text{drop} \hookrightarrow \epsilon$
  By Lemma Inversion-On-Instruction-Typing on $S; C \vdash (t.\text{const } c)\ \text{drop} : \epsilon; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$, we know that $ti_2^* = ti_1^*$, $l_2 = l_1$, and $\Gamma_2 = \Gamma_1, (t\ \alpha)$. $\phi_2 = \phi_1, (= \alpha\ (t\ c))$.
  We want to show that $S; C \vdash \epsilon : \epsilon; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_3; \phi_3$.
  We have $\alpha \notin \Gamma_1$, as it is a premise of Rule Const.
  By Lemma Well-formedness, $l_1 \subset \Gamma_1$.
  By Rule Empty, $S; C \vdash \epsilon : \epsilon; l_1; \Gamma_1, (t\ \alpha); \phi_1(= \alpha\ (t\ c)) \rightarrow \epsilon; l_1; \Gamma_1, (t\ \alpha); \phi_1(= \alpha\ (t\ c))$.
  Thus, $S; C \vdash \epsilon : ti_1^*; l_1; \Gamma_1, (t\ \alpha); \phi_1(= \alpha\ (t\ c)) \rightarrow ti_1^*; l_1; \Gamma_1, (t\ \alpha); \phi_1(= \alpha\ (t\ c))$ since $ti_2^* = ti_1^*$, by Rule Stack-Poly.

- Case: $S; C \vdash (t.\text{const } c_1)\ (t.\text{const } c_2)\ (\text{i32.const } k+1)\ \text{select} : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$
  $\wedge\ (t.\text{const } c_1)\ (t.\text{const } c_2)\ (\text{i32.const } k+1)\ \text{select} \hookrightarrow (t.\text{const } c_1)$
  By Lemma Inversion-On-Instruction-Typing on $S; C \vdash (t.\text{const } c_1)\ (t.\text{const } c_2)\ (\text{i32.const } k+1)\ \text{select} : \epsilon; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$, we know that $ti_2^* = ti_1^*\ (\alpha_3\ )$, $l_2 = l_1$, $\Gamma_2 = \Gamma_1, (t\ \alpha_1), (t\ \alpha_2), (\text{i32}\ \alpha), (t\ \alpha_3)$ and $\phi_2 = \phi_1, (= \alpha_1\ (t\ c_1)), (= \alpha_2\ (t\ c_2)), (= \alpha\ (\text{i32}\ k+1)), (if\ (= \alpha\ (\text{i32}\ 0))\ (= \alpha_3\ \alpha_2)\ (= \alpha_3\ \alpha_1))$
  Then, we want to show that $S; C \vdash (t.\text{const } c_1) : \epsilon; l_1; \Gamma_1, (t\ \alpha_1), (t\ \alpha_2), (\text{i32}\ \alpha); \phi_1 \rightarrow (t\ \alpha_3); l_2; \Gamma_3; \phi_3$, where $(t\ \alpha_3), l_2 \subset \Gamma_3$ and

$$\Gamma_3 \vdash \phi_3 \Rightarrow \phi_1, (= \alpha_1\ (t\ c_1)), (= \alpha_2\ (t\ c_2)), (= \alpha\ (\text{i32}\ k+1)), (if\ (= \alpha\ (\text{i32}\ 0))\ (= \alpha_3\ \alpha_2)\ (= \alpha_3\ \alpha_1))$$

By Rule Const,

$$S; C \vdash (t.\mathsf{const}\ c_1) : \epsilon; l_1; \Gamma_1, (t\ \alpha_1), (t\ \alpha_2), (\mathsf{i32}\ \alpha); \phi_1, (=\alpha_1\ (t\ c_1)), (=\alpha_2\ (t\ c_2)), (=\alpha\ (\mathsf{i32}\ k+1)), (=\alpha_3\ (t\ c_1$$
$$\rightarrow (t\ \alpha_3); l_1; \Gamma_1, (\mathsf{i32}\ \alpha_2), (t\ \alpha_3); \phi_1, (=\alpha_1\ (t\ c_1)), (=\alpha_2\ (t\ c_2)), (=\alpha\ (\mathsf{i32}\ k+1)), (=\alpha_3\ (t\ c_1)), (=\alpha_1\ (t\ c_1))$$

since $\alpha_3 \notin \Gamma_1, (t\ \alpha_1), (t\ \alpha_2), (\mathsf{i32}\ \alpha)$.
We have $\Gamma_3 = \Gamma_1, (t\ \alpha_1), (t\ \alpha_2), (\mathsf{i32}\ \alpha), (t\ \alpha_3)$, and by Lemma Well-formedness, $l_1 \subset \Gamma_1$, so

$$(t\ \alpha_3), l_1 \subset \Gamma_1, (t\ \alpha_1), (t\ \alpha_2), (\mathsf{i32}\ \alpha), (t\ \alpha_3)$$

Recall that we know $\Gamma_3 \vdash \phi_3 \Rightarrow \phi_1, (=\alpha_1\ (t\ c_1)), (=\alpha_2\ (t\ c_2)), (=\alpha\ (\mathsf{i32}\ k+1)), (\mathit{if}\ (=\alpha\ (\mathsf{i32}\ 0))\ (=\alpha_3\ \alpha_2)\ (=\alpha_3\ \alpha_1))$
Finally,

$$S; C \vdash (t.\mathsf{const}\ c_1) : ti_1^*; l_1; \Gamma_1, (t\ \alpha_1), (t\ \alpha_2), (\mathsf{i32}\ \alpha); \phi_1, (=\alpha_1\ (t\ c_1)), (=\alpha_2\ (t\ c_2)), (=\alpha\ (\mathsf{i32}\ k+1)), (=\alpha_3\ (t$$
$$\rightarrow ti_1^*\ (t\ \alpha_3); l_1; \Gamma_1, (\mathsf{i32}\ \alpha_2), (t\ \alpha_3); \phi_1, (=\alpha_1\ (t\ c_1)), (=\alpha_2\ (t\ c_2)), (=\alpha\ (\mathsf{i32}\ k+1)), (=\alpha_3\ (t\ c_1)), (=\alpha_1\ (t\ c_1)$$

since $ti_2 = ti_1^*\ (t\ \alpha_3)$, by Rule Stack-Poly.

- Case: $S; C \vdash (t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ (\mathsf{i32.const}\ 0)\ \mathsf{select} : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$
  $\wedge\ (t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ (\mathsf{i32.const}\ 0)\ \mathsf{select} \hookrightarrow (t.\mathsf{const}\ c_2)$
  Same as above, except 0 instead of $k+1$, and $(=\alpha_3\ (t\ c_2))$ instead of $(=\alpha_3\ (t\ c_1))$.

- Case: $S; C \vdash (t.\mathsf{const}\ c)^n\ \mathsf{block}\ (t_1^n \rightarrow t_2^m)\ e^*\ \mathsf{end} : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$
  $\wedge\ (t.\mathsf{const}\ c)^n\ \mathsf{block}\ (t_1^n \rightarrow t_2^m)\ e^*\ \mathsf{end}$
  $\hookrightarrow \mathsf{label}_m\{\epsilon\}\ (t.\mathsf{const}\ c)^n\ e^*\ \mathsf{end}$
  We want to show that $S; C \vdash \mathsf{label}_m\{\epsilon\}\ (t.\mathsf{const}\ c)^n\ e^*\ \mathsf{end} : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$.
  (1) The following hold by Lemma Inversion-On-Instruction-Typing on $S; C \vdash (t.\mathsf{const}\ c)^n\ \mathsf{block}\ (t_1^n \rightarrow t_2^m)\ e^*\ \mathsf{end} : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$, for some $ti_1'^*$, l_1', Γ_1', ϕ_1', and ti_3^*
    (a) $S; C \vdash (t.\mathsf{const}\ c)^n\ : ti_3^*\ ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_1'^*; l_1'; \Gamma_1'; \phi_1'$
    (b) $S; C \vdash \mathsf{block}\ (t_1^n \rightarrow t_2^m)\ e^*\ \mathsf{end} : ti_1'^*; l_1'; \Gamma_1'; \phi_1' \rightarrow ti_3^*\ ti_2^*; l_2; \Gamma_2; \phi_2$
  (2) Then, by Lemma Inversion-On-Instruction-Typing on $S; C \vdash \mathsf{block}\ (t_1^n \rightarrow t_2^m)\ e^*\ \mathsf{end} : ti_1'^*; l_1'; \Gamma_1'; \phi_1' \rightarrow ti_3^*\ ti_2^*; l_2; \Gamma_2; \phi_2$, we have that
    (a) $ti_1'^* = ti_4^*\ (t_1\ \alpha_1)^n$, for some ti_4^*
    (b) $ti_2^* = ti_4^*\ (t_2\ \alpha_2)^m$
    (c) $S; C, \mathsf{label}\ ((t_2\ \alpha_2)^m; l_2; \phi_2) \vdash e^* \vdash e^* : (t_1\ \alpha_1)^n; l_1'; \Gamma_1'; \phi_1' \rightarrow (t_2\ \alpha_2)^m; l_2; \Gamma_2; \phi_3$
    (d) $\Gamma_2 \vdash \phi_3 \Rightarrow \phi_2$
  (3) By Rule Stack-Poly and 2c, we have that $S; C, \mathsf{label}\ ((t_2\ \alpha_2)^m; l_2; \phi_2) \vdash e^* \vdash e^* : ti_4^*\ (t_1\ \alpha_1)^n; l_1'; \Gamma_1'; \phi_1' \rightarrow ti_4^*\ (t_2\ \alpha_2)^m; l_2; \Gamma_2; \phi_3$
  (4) Then, by 2a and 2b, it follows that $S; C, \mathsf{label}\ ((t_2\ \alpha_2)^m; l_2; \phi_2) \vdash e^* \vdash e^* : ti_1'^*; l_1'; \Gamma_1'; \phi_1' \rightarrow ti_2^*; l_2; \Gamma_2; \phi_3$
  (5) By Lemma Values-Any-Context and 1a, we have that $S; C, \mathsf{label}\ ((t_2\ \alpha_2)^m; l_2; \phi_2) \vdash (t.\mathsf{const}\ c)^n\ : ti_3^*\ ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_1'^*; l_1'; \Gamma_1'; \phi_1'$

(6) Then, by Lemma Sequence-Composition, 4, and 5, we have that $S; C, \text{label } ((t_2\ \alpha_2)^m; l_2; \phi_2) \vdash (t.\text{const } c)^n\ e^* : ti_3^*\ ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$

(7) By Rule Empty, $S; C \vdash \epsilon : ti_2^*; l_2; \Gamma_2; \phi_2 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$.

(8) We have $\Gamma_2 \supseteq \Gamma_1, ti_2^*; l_2$ as a result of Lemma Well-Formedness

(9) Finally, by Rule Label, 6, 7, 8, and 2d, we have that $S; C \vdash \text{label}_m\{\epsilon\}\ (t.\text{const } c)^n\ e^*\ \text{end} : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$

- Case: $S; C \vdash (t.\text{const } c)^n\ \text{loop}\ (t_1^n \rightarrow t_2^m)\ e^*\ \text{end} : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$
$\wedge\ (t.\text{const } c)^n\ \text{loop}\ ti_3^n; l_3; \phi_3 \rightarrow ti_4^m; l_4; \phi_4\ e^*\ \text{end} \hookrightarrow \text{label}_n\{\text{loop}\ ti_3^n; l_3; \phi_3 \rightarrow ti_4^m; l_4; \phi_4\ e^*\ \text{end}\}\ (t.\text{co}$
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 want to show that $S; C \vdash \text{label}_m\{\text{loop}\ (t_1^n \rightarrow t_2^m)\ e^*\ \text{end}\}\ (t.\text{const } c)^n\ e^*\ \text{end} : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$.

(1) The following hold by Lemma Inversion-On-Instruction-Typing on $S; C \vdash (t.\text{const } c)^n\ \text{loop}\ (t_1^n \rightarrow t_2^m)\ e^*\ \text{end} : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$, for some $ti_1'^*$, l_1', Γ_1', ϕ_1', and ti_3^*
  (a) $S; C \vdash (t.\text{const } c)^n\ : ti_3^*\ ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_1'^*; l_1'; \Gamma_1'; \phi_1'$
  (b) $S; C \vdash \text{loop}\ (t_1^n \rightarrow t_2^m)\ e^*\ \text{end} : ti_1'^*; l_1'; \Gamma_1'; \phi_1' \rightarrow ti_3^*\ ti_2^*; l_2; \Gamma_2; \phi_2$

(2) Then, by Lemma Inversion-On-Instruction-Typing on $S; C \vdash \text{loop}\ (t_1^n \rightarrow t_2^m)\ e^*\ \text{end} : ti_1'^*; l_1'; \Gamma_1'; \phi_1' \rightarrow ti_3^*\ ti_2^*; l_2; \Gamma_2; \phi_2$, we have that
  (a) $ti_1'^* = ti_4^*\ (t_1\ \alpha_1)^n$, for some ti_4^*
  (b) $ti_2^* = ti_4^*\ (t_2\ \alpha_2)^m$
  (c) $S; C, \text{label } ((t_1\ \alpha_1)^m; l_1; \phi_1) \vdash e^* : (t_1\ \alpha_1)^n; l_1'; \Gamma_1'; \phi_1' \rightarrow (t_2\ \alpha_2)^m; l_2; \Gamma_2; \phi_3$
  (d) $\Gamma_2 \vdash \phi_3 \Rightarrow \phi_2$

(3) By Rule Stack-Poly and 2c, we have that $S; C, \text{label } ((t_1\ \alpha_1)^m; l_1; \phi_1) \vdash e^* : ti_4^*\ (t_1\ \alpha_1)^n; l_1'; \Gamma_1'; \phi_1' \rightarrow ti_4^*\ (t_2\ \alpha_2)^m; l_2; \Gamma_2; \phi_3$

(4) Then, by 2a and 2b, it follows that $S; C, \text{label } ((t_1\ \alpha_1)^m; l_1; \phi_1) \vdash e^* : ti_1'^*; l_1'; \Gamma_1'; \phi_1' \rightarrow ti_2^*; l_2; \Gamma_2; \phi_3$

(5) By Lemma Values-Any-Context and 1a, we have that $S; C, \text{label } ((t_1\ \alpha_1)^m; l_1; \phi_1) \vdash (t.\text{const } c)^n\ : ti_3^*\ ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_1'^*; l_1'; \Gamma_1'; \phi_1'$

(6) Then, by Lemma Sequence-Composition, 4, and 5, we have that $S; C, \text{label } ((t_1\ \alpha_1)^m; l_1\phi_1) \vdash (t.\text{const } c)^n\ e^* : ti_3^*\ ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$

(7) We have $\Gamma_1 \supseteq \Gamma_1, ti_1^*; l_1$ as a result of Lemma Well-Formedness

(8) Finally, by Rule Label, 6, 7, and 2d, we have that $S; C \vdash \text{label}_m\{\text{loop}\ (t_1^n \rightarrow t_2^m)\ e^*\ \text{end}\}\ (t.\text{const } c)^n\ e^*\ \text{e}$
$ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$

- Case: $S; C \vdash (\text{i32.const } 0)\ \text{if } (ti_3^n \to ti_4^m)\ e_1^*\ \text{else } e_2^*\ \text{end} : ti_1^*; l_1; \Gamma_1; \phi_1 \to ti_2^*; l_2; \Gamma_2; \phi_2$
$\wedge\ (\text{i32.const } 0)\ \text{if } (ti_3^n \to ti_4^m)\ e_1^*\ \text{else } e_2^*\ \text{end} \hookrightarrow \text{block } (ti_3^n \to ti_4^m)\ e_2^*\ \text{end}$
We want to show that

$$\text{block } (ti_3^n \to ti_4^m)\ e_1^*\ \text{end} : ti_1^*; l_1; \Gamma_1; \phi_1 \to ti_2^*; l_2; \Gamma_2; \phi_2$$

First, we reason about $ti_1^*; l_1; \Gamma_1; \phi_1 \to ti_2^*; l_2; \Gamma_2; \phi_2$.
(1) By Lemma Inversion-On-Instruction-Typing on $S; C \vdash (\text{i32.const } 0)\ \text{if } (ti_3^n \to ti_4^m)\ e_1^*\ \text{else } e_2^*\ \text{end} : ti_1^*; l_1; \Gamma_1; \phi_1 \to ti_2^*; l_2; \Gamma_2; \phi_2$, we have that
 (a) $S; C \vdash (t.\text{const } 0)^n\ : ti_1^*; l_1; \Gamma_1; \phi_1 \to ti_4^*; l_4; \Gamma_4; \phi_4$
 (b) $S; C \vdash \text{if } (ti_3^n \to ti_4^m)\ e_1^*\ \text{else } e_2^*\ \text{end} : ti_4^*; l_4; \Gamma_4; \phi_4 \to ti_2^*; l_2; \Gamma_2; \phi_2$
(2) By Lemma Type-Of-Values and 1a, we have that $ti_4^* = ti_1^*\ (\text{i32}\ \alpha)$, $l_4 = l_1$, $\Gamma_4 = \Gamma_1, (\text{i32}\ \alpha)$, and $\phi_4 = \phi_1, (=\ \alpha\ (\text{i32}\ \alpha))$
(3) Then, by Lemma Inversion-On-Instruction-Typing on $S; C \vdash \text{if } (ti_3^n \to ti_4^m)\ e_1^*\ \text{else } e_2^*\ \text{end} : ti_1^*\ (\text{i32}\ \alpha); l_1; \Gamma_1, (\text{i32}\ \alpha); \phi_1, (=\ \alpha\ (\text{i32}\ \alpha)) \to ti_2^*; l_2; \Gamma_2; \phi_2$, we have that $S; C, \text{label}(ti_2^m; l_2; \phi_2) \vdash e_2^* : ti_1^*; l_1; \Gamma_1, (\text{i32}\ \alpha); \phi_1, (=\ \alpha\ (\text{i32}\ \alpha)) \to ti_2^*; l_2; \Gamma_2; \phi_5$, where $\Gamma_2 \vdash \phi_5 \Rightarrow \phi_2$
(4) Then, by Rule Block, we have that $S, C \vdash \text{block } (ti_3^n \to ti_4^m)\ e_2^*\ \text{end} : ti_1^*; l_1; \Gamma_1, (\text{i32}\ \alpha); \phi_1, (=\ \alpha\ (\text{i32}\ \alpha)) \to ti_2^*; l_2; \Gamma_2; \phi_2$

- Case: $S; C \vdash (\text{i32.const } k+1)\ \text{if } (ti_3^n; l_3; \phi_3 \to ti_4^m; l_4; \phi_4)\ e_1^*$
 $\text{else } e_2^*$
 end
 $: ti_1^*; l_1; \phi_1 \to ti_2^*; l_2; \phi_2$
$\wedge\ (\text{i32.const } k+1)\ \text{if } ti_3^n; l_3; \phi_3 \to ti_4^m; l_4; \phi_4\ e_1^*\ \text{else } e_2^*\ \text{end}$
$\hookrightarrow \text{block } ti_3^n; l_3; \phi_3 \to ti_4^m; l_4; \phi_4\ e_1^*\ \text{end}$
This case is the same as above, except with e_2 instead of e_1 and $k+1$ instead of 0.

- Case: $S; C \vdash \text{label}_n\{e^*\}\ (t.\text{const } c)^n\ \text{end} : ti_1^*; l_1; \Gamma_1; \phi_1 \to ti_2^*; l_2; \Gamma_2; \phi_2$
$\wedge\ \text{label}_n\{e^*\}\ (t.\text{const } c)^*\ \text{end} \hookrightarrow (t.\text{const } c)^*$
By Lemma Inversion-On-Instruction-Typing on $S; C \vdash \text{label}_n\{e^*\}\ (t.\text{const } c)^n\ \text{end} : ti_1^*; l_1; \Gamma_1; \phi_1 \to ti_2^*; l_2; \Gamma_2; \phi_2$, we have that $ti_2^* = ti_1^*\ ti^n$.
We have $S; C \vdash (t.\text{const } c)^n\ : \epsilon; l_1; \Gamma_1; \phi_1 \to (t\ \alpha_3)^n; (t_{l2}\ \alpha_{l3})^*; \Gamma_3; \phi_3$, where $(t\ \alpha)^n = ti^*$, $(t_{l2}\ \alpha_{l2})^* = l_2$; $\Gamma_3 \vdash \phi_3 \Rightarrow \phi_2[\alpha \mapsto \alpha_3][\alpha_{l2} \mapsto \alpha_{l3}]$; and $ti^*\ l_2 \notin \Gamma_1$, as they are premises of Rule Label, which we have assumed to hold.
By Lemma Inversion-On-Instruction-Typing on $S; C \vdash (t.\text{const } c)^n : \epsilon; l_1; \Gamma_1; \phi_1 \to (t\ \alpha_3)^n; l_1; \Gamma_3; \phi_3$, we know $(t_{l2}\ \alpha_{l2})^* = l_2 = l_1$, $\Gamma_3 = \Gamma_1, (t\ \alpha_3)^n$, and $\phi_3 = \phi_1, (=\alpha_3\ (t\ c))^n$.
Since $ti^*\ l_2 \notin \Gamma_1$, we can get $S; C \vdash (t.\text{const } c)^*\ : \epsilon; l_1; \Gamma_1; \phi_1 \to (t\ \alpha)^*; l_1; \Gamma_1, (t\ \alpha)^*; \phi_1, (=\alpha\ (t\ c))^*$ by Rule Const.
Then, by Rule Const,

$$\begin{aligned}S; C \vdash (t.\text{const } c)^* : \epsilon; (t_{l2}\ \alpha_{l2})^*; \Gamma_1[\alpha_{l1} \mapsto \alpha_{l2}]; \phi_1[\alpha_{l1} \mapsto \alpha_{l2}]\\ \to (t\ \alpha)^*; (t_{l2}\ \alpha_{l2})^*; \Gamma_1[\alpha_{l1} \mapsto \alpha_{l2}], (t\ \alpha)^*; \phi_1[\alpha_{l1} \mapsto \alpha_{l2}], (=\alpha\ (t\ c))^*\end{aligned}$$

Therefore,

$$\begin{aligned}S; C \vdash (t.\text{const } c)^* : ti_1^*; (t_{l2}\ \alpha_{l2})^*; \Gamma_1[\alpha_{l1} \mapsto \alpha_{l2}]; \phi_1[\alpha_{l1} \mapsto \alpha_{l2}]\\ \to ti_2^*; (t_{l2}\ \alpha_{l2})^*; \Gamma_1[\alpha_{l1} \mapsto \alpha_{l2}], (t\ \alpha)^*; \phi_1[\alpha_{l1} \mapsto \alpha_{l2}], (=\alpha\ (t\ c))^*\end{aligned}$$

since $\Gamma_1[\alpha_{l1} \mapsto \alpha_{l2} \vdash, \Rightarrow (t\ \alpha)^*]\phi_1[\alpha_{l1} \mapsto \alpha_{l2}], (= \alpha\ (t\ c))^*\phi_2$ and $ti_2^* = ti_1^*\ (t\ \alpha)^*$.

- Case: $S; C \vdash \text{label}_n\{e^*\}\ \text{trap end} : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$
  $\wedge\ \text{label}_n\{e^*\}\ \text{trap end} \hookrightarrow \text{trap}$
  Trivially, $C \vdash \text{trap} : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$ by Rule Trap since trap accepts any precondition and postcondition.

- Case: $S; C \vdash \text{label}_n\{e^*\}\ L^j[(t.\text{const}\ c)^n\ (\text{br}\ j)]\ \text{end} : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$
  $\wedge\ \text{label}_n\{e^*\}\ L^j[(t.\text{const}\ c)^n\ (\text{br}\ j)] \hookrightarrow (t.\text{const}\ c)^n\ e^*$
  Intuitively, this proof works because the premise of Rule Br assumes that $C_{\text{label}}(i)$ is the precondition ($ti_3^n; l_3; \phi_3$, 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 assumption 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.
  First, we derive the type of $(t.\text{const}\ c)^n$ from the precondition of Rule Br.

(1) By Lemma Inversion-On-Instruction-Typing and $S; C \vdash \text{label}_n\{e^*\}\ L^j[(t.\text{const}\ c)^n\ (\text{br}\ j)]\ \text{end} : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$, we have
  (a) $S; C \vdash e^* : ti_3^n; l_3; \Gamma_3; \phi_3 \rightarrow ti_5^*; l_2; \Gamma_2; \phi_4$
  (b) $\Gamma_2 \vdash \phi_4 \Rightarrow \phi_2$
  (c) $\Gamma_3 \supset \Gamma_1, ti_3^n, l_3$
  (d) $S; C, \text{label}(ti_3^n; l_3; \phi_3) \vdash L^j[(t.\text{const}\ c)^n\ (\text{br}\ j)] : ti_0^*\ ti_4^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_0^*\ ti_5^*; l_2; \Gamma_2; \phi_5$, where $ti_1^* = ti_0^*\ ti_4^*$ and $ti_2^* = ti_0^*\ ti_5^*$ for some ti_0^*

(2) Then, by Lemma Values-Br-In-Context and 1d, we have that
  (a) $S; C' \vdash (t.\text{const}\ c)^n : ti_1^*; l_1; \Gamma_1, (t_0\ \alpha_0)^*; \phi_1, (= \alpha_0\ (t_0\ c_0))^* \rightarrow ti_1^*\ ti_3^n; l_1; \Gamma_1, (t_0\ \alpha_0)^*, ti_3^n; \phi_1, (= \alpha_0\ (t_0\ c_0))^*, (= \alpha_3\ (t\ c))^n$, for some α_0^*, t_0^*, and c_0^*, where $ti_3^n = (t\ \alpha_3)^n$
  (b) $\Gamma_1, (t_0\ \alpha_0)^*, ti_3^n \vdash \phi_1, (= \alpha_0\ (t_0\ c_0))^*, (= \alpha_3\ (t\ c))^n \Rightarrow \phi_3[l_3 \mapsto l_1]$
  (c) l_3 is fresh with respect to Γ_1 and Γ_2

(3) By assumption 3, we know $(\vdash v : (t_v\ \alpha_v); \phi_v)^*$, where $l_1 = (t_v\ \alpha_v)^*$ and $\phi_v \subset \phi_1$

(4) Trivially then, we have $(\vdash v : (t_v\ \alpha_{v3}); \phi_{v3})^*$, where $l_3 = (t_v\ \alpha_{v3})^*$

(5) Replacing ϕ_v with ϕ_{v3} in ϕ_1 to obtain $\hat{\phi}_1$, we have that $\Gamma_1, (t_0\ \alpha_0)^*, ti_3^n \vdash p\hat{h}i_1, (= \alpha_0\ (t_0\ c_0))^*, (= \alpha_3\ (t\ c))^n \Rightarrow \phi_3$

(6) We choose $(t\ \alpha)^*$ to be the set difference between Γ_3 and $\Gamma_1, (t_0\ \alpha_0)^*, l_3, ti_3^n$

(7) Then, by Lemma Values-Any-Context, and 2a, we have that $S; C \vdash (t.\text{const}\ c)^n : ti_0^*; l_3; \Gamma_1, (t\ \alpha)^*, l_3, (t_0$
  $\alpha_0\ (t_0\ c_0))^*, (= \alpha_3\ (t\ c))^n \rightarrow ti_0^*\ ti_3^n; l_3; \Gamma_3; p\hat{h}i_1, (= \alpha_0\ (t_0\ c_0))^*, (= \alpha_3\ (t\ c))^n$

(8) Then, by Lemma Strengthening, 5, and 1a, we have that $S; C \vdash e^* : ti_0^*\ ti_3^n; l_3; \Gamma_3; p\hat{h}i_1, (= \alpha_0\ (t_0\ c_0))^*, (= \alpha_3\ (t\ c))^n \rightarrow ti_2^*; l_2; \Gamma_2; \phi_6$, where $\Gamma_2 \vdash \phi_6 \Rightarrow \phi_2$

(9) Finally, by Lemma Sequence-Composition, we have that $S; C \vdash (t.\mathsf{const}\ c)^n\ e^* : ti_0^*; l_3; \Gamma_1, (t\ \alpha)^*, l_3, (t_0\ \alpha$
$\alpha_0\ (t_0\ c_0))^*, (=\ \alpha_3\ (t\ c))^n \rightarrow ti_2^*; l_2; \Gamma_2; \phi_6$

- Case: $S; C \vdash (\mathsf{i32.const}\ 0)\ (\mathsf{br\_if}\ j) : \epsilon; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$
$\wedge\ (\mathsf{i32.const}\ 0)\ (\mathsf{br\_if}\ j) \hookrightarrow \epsilon$
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: $S; C \vdash (\mathsf{i32.const}\ k+1)\ (\mathsf{br\_if}\ j) : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$
$\wedge\ (\mathsf{i32.const}\ k+1)\ (\mathsf{br\_if}\ j) \hookrightarrow \mathsf{br}\ j$
We want to show that $S; C \vdash \mathsf{br}\ j : ti_1^*; l_1; \Gamma_1, (\mathsf{i32}\ \alpha); \phi_1, (=\alpha\ (\mathsf{i32}\ k+1)) \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$.
We know $S; C \vdash \mathsf{br\_if}\ j : ti_1^*\ (\mathsf{i32}\ \alpha); l_1; \Gamma_1, (\mathsf{i32}\ \alpha); \phi_1, (=\alpha\ (\mathsf{i32}\ k+1)) \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$, where $\alpha \notin \Gamma_1$ by Lemma Inversion-On-Instruction-Typing on $S; C \vdash (\mathsf{i32.const}\ k{+}1)\ (\mathsf{br\_if}\ j) : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$.
Then we know $C_{\text{label}}(j) = (ti_3^*; l_3; \phi_3)$, where $ti_1^* = ti_0^*\ (t_1\ \alpha_1)^*$, $ti_3^* = (t_1\ \alpha_3)^*$, $l_1 = (t_l\ \alpha_{l1})^*$, $l_3 = (t_l\ \alpha_{l3})^*$, and $\Gamma_1, (\mathsf{i32}\ \alpha) \vdash \phi_1, \neg(=\alpha\ (\mathsf{i32}\ 0)) \Rightarrow \phi_3[\alpha_3 \mapsto \alpha_1]^*[\alpha_{l3} \mapsto \alpha_{l1}]^*$ by Lemma Inversion-On-Instruction-Typing on $S; C \vdash \mathsf{br\_if}\ j : ti_1^*\ (\mathsf{i32}\ \alpha); l_1; \Gamma_1; \phi_1, (=\alpha\ (\mathsf{i32}\ k+1)) \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$.
Then we have $S; C \vdash \mathsf{br}\ j : ti_0^*\ (t_1\ \alpha_1)^*; l_1; \Gamma_1, (\mathsf{i32}\ \alpha); \phi_1, (=\alpha\ (\mathsf{i32}\ k+1)) \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$ by Rule Br.

- Case: $S; C \vdash (\mathsf{i32.const}\ k)\ (\mathsf{br\_table}\ j_1^k\ j\ j_2^*) : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$
$\wedge\ (\mathsf{i32.const}\ k)\ (\mathsf{br\_table}\ j_1^k\ j\ j_2^*) \hookrightarrow \mathsf{br}\ j$
We want to show that $S; C \vdash \mathsf{br}\ j : ti_1^*; l_1; \Gamma_1, (\mathsf{i32}\ \alpha); \phi_1, (=\alpha\ (\mathsf{i32}\ k)) \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$.
This case is similar in structure to the $(\mathsf{i32.const}\ k+1)\ (\mathsf{br\_if}\ j)$ case.
We know $S; C \vdash \mathsf{br\_table}\ j_1^k\ j\ j_2^* : ti_1^*\ (\mathsf{i32}\ \alpha); l_1; \Gamma_1, (t\ \alpha); \phi_1, (=\alpha\ (\mathsf{i32}\ k)) \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$ by Lemma Inversion-On-Instruction-Typing on $S; C \vdash (\mathsf{i32.const}\ k)\ (\mathsf{br\_table}\ j_1^k\ j\ j_2^*) : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$.
Then we know $C_{\text{label}}(j) = (ti_3^*; l_1; \phi_3)$, where $ti_1^* = ti_0^*\ (t_1\ \alpha_1)^*$, $ti_3^* = (t_1\ \alpha_3)^*$, $l_1 = (t_l\ \alpha_{l1})^*$, $l_3 = (t_l\ \alpha_{l3})^*$, and $\Gamma_1, (\mathsf{i32}\ \alpha) \vdash \phi_1, (=\alpha\ (\mathsf{i32}\ k)) \Rightarrow \phi_3[\alpha_3 \mapsto \alpha_1]^*[\alpha_{l3} \mapsto \alpha_{l1}]^*$ by Lemma Inversion-On-Instruction-Typing on $S; C \vdash \mathsf{br\_table}\ j_1^k\ j\ j_2^* : ti_1^*\ (\mathsf{i32}\ \alpha); l_1; \Gamma_1, (\mathsf{i32}\ \alpha); \phi_1, (=\alpha\ (\mathsf{i32}\ k)) \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$.
Therefore we have $S; C \vdash \mathsf{br}\ j : ti_0^*\ (t_1\ \alpha_1)^*; l_1; \Gamma_1, (\mathsf{i32}\ \alpha); \phi_1, (=\alpha\ (\mathsf{i32}\ k)) \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$ by Rule Br.

- Case: $C \vdash (\mathsf{i32.const}\ k+n)\ (\mathsf{br\_table}\ j_1^k\ j) : ti_1^*; l_1; \phi_1 \rightarrow ti_2^*; l_2; \phi_2$
$\wedge\ (\mathsf{i32.const}\ k+n)\ (\mathsf{br\_table}\ j_1^k\ j) \hookrightarrow \mathsf{br}\ j$
Same as above.

- Case: $S; C \vdash \mathsf{call}\ j : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$
$\wedge\ s; v^*; \mathsf{call}\ j \hookrightarrow_i \mathsf{call}\ s; v^*; s_{\text{func}}(i, j)$
We want to show that $S; C \vdash \mathsf{call}\ s_{\text{func}}(i, j) : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$.
By Lemma Inversion-On-Instruction-Typing on $S; C \vdash \mathsf{call}\ j : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$, we know that $l_2 = l_1$, $ti_1^* = ti_0^*\ (t_3\ \alpha_3)^*$, $ti_2^* = ti_0^*\ (t_4\ \alpha_4)^*$, $\Gamma_1 \vdash \phi_1 \Rightarrow \phi_3[\alpha_5 \mapsto \alpha_3]$, $\phi_2 = \phi_1 \cup \phi_4[\alpha_5 \mapsto \alpha_3][\alpha_6 \mapsto \alpha_4]$, and $\Gamma_2 = \Gamma_1, (t_4\ \alpha_4)^*$ where $(t_3\ \alpha_5)^*; \phi_3 \rightarrow (t_4\ \alpha_6)^*; \phi_4 = C_{\text{func}}(j)$, and $(t_4\ \alpha_4)^* \notin \Gamma_1$.
Thus, we want to show that $S; C \vdash \mathsf{call}\ s_{\text{func}}(i, j) : ti_0^*\ (t_3\ \alpha_3)^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_0^*\ (t_4\ \alpha_4)^*; l_2; \Gamma_1, (t_3\ \alpha_3)^*; \phi_1 \cup$
$\phi_4[\alpha_5 \mapsto \alpha_3][\alpha_6 \mapsto \alpha_4]$.

Then we know $S \vdash s_{\text{func}}(i, j) : (t_3\ \alpha_5)^*; \phi_3 \rightarrow (t_4\ \alpha_6)^*; \phi_4$ because it is a premise of $S \vdash s_{\text{inst}}(i) : C$, which is a premise of $\vdash s : S$.
Therefore, $S; C \vdash \text{call } s_{\text{func}}(i, j) : (t_3\ \alpha_3)^*; l_1; \Gamma_1; \phi_1 \rightarrow (t_4\ \alpha_4)^*; l_1; \Gamma_1, (t_4\ \alpha_4)^*; \phi_1 \cup \phi_4[\alpha_5 \mapsto \alpha_3][\alpha_6 \mapsto \alpha_4]$ by Rule Call-Cl.
Thus, $S; C \vdash \text{call } s_{\text{func}}(i, j) : ti_0^*\ (t_3\ \alpha_3)^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_0^*\ (t_4\ \alpha_4)^*; l_2; \Gamma_1, (t_4\ \alpha_4)^*; \phi_1 \cup \phi_4[\alpha_5 \mapsto \alpha_3][\alpha_6 \mapsto \alpha_4]$ by Rule Stack-Poly and since $l_1 = l_2$.

- Case: $S; C \vdash (\text{i32.const } j)\ \text{call\_indirect } ((t_3\ \alpha_5)^*; \phi_3 \rightarrow (t_4\ \alpha_6)^*; \phi_4) : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$
  $\wedge\ s; (\text{i32.const } j)\ \text{call\_indirect } ((t_3\ \alpha_5)^*; \phi_3 \rightarrow (t_4\ \alpha_6)^*; \phi_4) \hookrightarrow_i \text{call } s_{\text{tab}}(i, j)$
  where $s_{\text{tab}}(i, j)_{\text{code}} = (\text{func } (t_3\ \alpha_5)^*; \phi_3 \rightarrow (t_4\ \alpha_6)^*; \phi_4 \text{ local } t^*\ e^*)$
  We want to show that $\text{call } s_{\text{tab}}(i, j) : ti_1^*; l_1; \Gamma_1, (\text{i32}\ \alpha); \phi_1, (= \alpha\ (\text{i32}\ j)) \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$.
  The extra index variable is added to the precondition since the instruction that creates it is reduced away.
  We know that $S; C \vdash \text{call\_indirect } ((t_3\ \alpha_5)^*; \phi_3 \rightarrow (t_4\ \alpha_6)^*; \phi_4) : ti_1^*\ (\text{i32}\ \alpha); l_1; \Gamma_1, (\text{i32}\ \alpha); \phi_1, (= \alpha\ (\text{i32}\ j)) \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$ by Lemma Inversion-On-Instruction-Typing on $S; C \vdash (\text{i32.const } j)$ call_indir
  $(t_4\ \alpha_6)^*; \phi_4) : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$.
  By Lemma Inversion-On-Instruction-Typing on $S; C \vdash \text{call\_indirect } ((t_3\ \alpha_5)^*; \phi_3 \rightarrow (t_4\ \alpha_6)^*; \phi_4) : ti_1^*\ (\text{i32}\ \alpha); l_1; \Gamma_1, (\text{i32}\ \alpha); \phi_1, (= \alpha\ (\text{i32}\ j)) \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$, we know that $ti_1^* = ti_0^*\ (t_3\ \alpha_3)^*$, and $ti_2^* = ti_0^*\ (t_4\ \alpha_4)^*$ for some ti_0^*, $l_1 = l_2$, $\Gamma_1 \vdash \phi_1 \Rightarrow \phi_3[\alpha_5 \mapsto \alpha_3]$, $\Gamma_2 = \Gamma_1, (\text{i32}\ \alpha), (t_4\ \alpha_4)^*$, and $\phi_2 = \phi_1, (= \alpha\ (\text{i32}\ j)) \cup \phi_4[\alpha_5 \mapsto \alpha_3][\alpha_6 \mapsto \alpha_4]$.
  We know $S \vdash s_{\text{tab}}(i, j) : (t_3\ \alpha_5)^*; \phi_3 \rightarrow (t_4\ \alpha_6)^*; \phi_4$ since it is a premise of $\vdash s : S$ which we have assumed to hold.
  Then, $S; C \vdash \text{call } s_{\text{tab}}(i, j) : (t_3\ \alpha_3)^*; l_1; \Gamma_1, (\text{i32}\ \alpha); \phi_1, (= \alpha\ (\text{i32}\ j)) \rightarrow (t_4\ \alpha_4)^*; l_1; \Gamma_1, (\text{i32}\ \alpha), (t_4\ \alpha_4)^*; \phi_1$
  $\alpha\ (\text{i32}\ j)) \cup \phi_4[\alpha_5 \mapsto \alpha_3][\alpha_6 \mapsto \alpha_4]$ by Rule Call-Cl.
  Therefore, $S; C \vdash \text{call } s_{\text{tab}}(i, j) : ti_0^*\ (t_3\ \alpha_3)^*; l_1; \Gamma_1, (\text{i32}\ \alpha); \phi_1, (= \alpha\ (\text{i32}\ j)) \rightarrow ti_0^*\ (t_4\ \alpha_4)^*; l_1; \Gamma_1, (\text{i32}\ \alpha),$
  $\alpha\ (\text{i32}\ j)) \cup \phi_4[\alpha_5 \mapsto \alpha_3][\alpha_6 \mapsto \alpha_4]$ by Rule Stack-Poly.

- Case: $S; C \vdash (\text{i32.const } j)\ \text{call\_indirect}✓\ (ti_3^*; l_3; \Gamma_3; \phi_3 \rightarrow ti_4^*; l_4; \Gamma_4; \phi_4) : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$
  $\wedge\ s; (\text{i32.const } j)\ \text{call\_indirect}✓\ ti_3^*; l_3; \Gamma_3; \phi_3 \rightarrow ti_4^*; l_4; \Gamma_4; \phi_4 \hookrightarrow_i \text{call } s_{\text{tab}}(i, j)$
  where $s_{\text{tab}}(i, j)_{\text{code}} = (\text{func } ti_3^*; \epsilon; \Gamma_3; \phi_3 \rightarrow ti_4^*; \epsilon; \Gamma_4; \phi_4 \text{ local } t^*\ e^*)$
  Same as above.

- Case: $S; C \vdash (\text{i32.const } j)\ \text{call\_indirect } tfi : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$
  $\wedge\ s; (\text{i32.const } j)\ \text{call\_indirect } tfi \hookrightarrow_i \text{trap}$.
  Trivially, $S; C \vdash \text{trap} : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$ by Rule Trap.

- Case: $S; C \vdash v^n\ \text{call } cl : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$
  $\wedge\ s; v^n\ \text{call } cl \hookrightarrow_i \text{local}_m\ \{i; v^n\ (t.\text{const } 0)^k\}$
```
                                   block tfi
                                         e*
                                   end
                             end
```
  where $cl_{\text{code}} = \text{func } (ti_3^n; \phi_3 \rightarrow ti_4^m; \phi_4) \text{ local } t^k\ e^*$ and $cl_{\text{inst}} = i$
  Let $tfi = \epsilon; ti_3^n\ (t\ \alpha)^k; \phi_3 \rightarrow (t_4\ \alpha_5)^m; l_4; \phi_4[\alpha_4 \mapsto \alpha_5]$
  We want to show that

$$S; C \vdash \text{local}_m\ \{i; v^n\ (t.\text{const } 0)^k\}\ (\text{block } (\epsilon; ti_3^n\ (t_2\ \alpha_2)^n; \phi_3 \rightarrow ti_4^m; l_4; \phi_4)\ e^*\ \text{end})\ \text{end} : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l$$

By Lemma Inversion-On-Instruction-Typing on Rule Composition, Rule Const, and Rule Call-Cl, we know $l_2 = l_1$, $ti_2^* = ti_1^*\ (t_4\ \alpha_5)^m$, $\Gamma_1 \vdash \phi_1, (t_2\ \alpha_2), (\text{eq}\ \alpha_2\ (t_2\ c)) \Rightarrow \phi_3[\alpha_3 \mapsto \alpha_2]$, $\phi_2 = \phi_1, (=\alpha_2\ (t_2\ c))^n \cup \phi_4[\alpha_4 \mapsto \alpha_5][\alpha_3 \mapsto \alpha_2]$, $\Gamma_2 = \Gamma_1, (t_2\ \alpha_2)^n, (t_4\ \alpha_5)^m$, $(t_4\ \alpha_5)^m \notin \Gamma_1$ and $S \vdash cl : ti_3^n; \phi_3 \rightarrow ti_4^m; \phi_4$, where $(t_4\ \alpha_4)^m = ti_4^m$.
We also know that

$$S; C \vdash (t_2.\text{const}\ c)^n : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_1^*\ (t_2\ \alpha_2)^n; l_1; \Gamma_1, (t_2\ \alpha_2)^n; \phi_1, (=\alpha_2\ (t_2\ c))^n$$

where $v^n = (t_2.\text{const}\ c)^n$, and

$$\begin{aligned} S; C \vdash \text{call}\ cl : ti_1^*\ (t_2\ \alpha_2)^n; l_1; \Gamma_1, (t_2\ \alpha_2)^n; \phi_1, (=\alpha_2\ (t_2\ c))^n \\ \rightarrow (t_4\ \alpha_5)^m\ ti_2^m; l_1; \Gamma_1, (t_2\ \alpha_2)^n, (t_4\ \alpha_5)^m; \phi_1, (=\alpha_2\ (t_2\ c))^n \cup \phi_4[\alpha_4 \mapsto \alpha_5][\alpha_3 \mapsto \alpha_2] \end{aligned}$$

by Rule Inversion-On-Instruction-Typing on Rule Composition.
We have $C \vdash \text{func}\ ti_3^n; \phi_3 \rightarrow ti_4^m; \phi_4\ \text{local}\ t^k\ e^* : ti_3^n; \phi_3 \rightarrow ti_4^m; \phi_4$ because it is a premise of $S \vdash cl : ti_3^n; \phi_3 \rightarrow ti_4^m; \phi_4$.
Then,

$$\begin{aligned} &S; C, \text{local}\ t_2^n\ t^k, && \vdash e^* : \epsilon; (t_2\ \alpha_2)^n\ (t\ \alpha)^k; \emptyset, (t_2\ \alpha_2)^n\ (t\ \alpha)^k; (\phi_3, (=\alpha\ (t\ 0))^k)[\alpha_3 \mapsto \alpha_2] \\ &\quad \text{label}\ (ti_4^m; l_4; \phi_4), && \quad \rightarrow (t_4\ \alpha_6)^m; l_4; \Gamma_6; \phi_6 \\ &\quad \text{return}\ ((t_4\ \alpha_5)^m; \phi_4[\alpha_4 \mapsto \alpha_5]) && \end{aligned}$$

where $\Gamma_6 \vdash \phi_6 \Rightarrow \phi_4[\alpha_4 \mapsto \alpha_6]$ because it is a premise of the above derivation.
We can now reconstruct the type after reduction.

$$\begin{aligned} &S; C, \text{local}\ t_2^n\ t^k, && \vdash \text{block}\ tfi\ e^*\ \text{end} : (\epsilon; ti_3^n\ (t_2\ \alpha_2)^n; \Gamma_1; \phi_3 \\ &\quad \text{return}\ ((t_4\ \alpha_5)^m; \phi_4[\alpha_4 \mapsto \alpha_5]) && \quad \rightarrow (t_4\ \alpha_5)^m; l_4; \Gamma_1, (t_4\ \alpha_5)^m, l_4; \phi_1 \cup \phi_4[\alpha_4 \mapsto \alpha_5][\alpha_3 \end{aligned}$$

by Rule Block, since $(t_4\ \alpha_5)^m \notin \Gamma_1$.
$\vdash v : (t_2\ \alpha_2); \emptyset, (t_2\ \alpha_2), (\text{eq}\ \alpha_2\ (t_2\ c)))^n$ by Rule Admin-Const, and $(\vdash (t\text{const}\ 0) : (t\ \alpha); \emptyset, (t\ \alpha), (\text{eq}\ a\ (t\ 0$
by Rule Admin-Const.
Then, $S; (ti_4^m; \phi_4) \vdash v^n\ (t\text{const}\ 0)^k; \text{block}\ tfi\ e^*\ \text{end} : (t_4\ \alpha_5)^m; l_4; \Gamma_1, (t_4\ \alpha_5)^m, l_4; \phi_1 \cup \phi_4[\alpha_4 \mapsto \alpha_5][\alpha_3 \mapsto \alpha_2])$ by Rule Code.
$S; C \vdash \text{local}_m\{j; v^n\ (t.\text{const}\ 0)^k\}\ \text{block}\ tfi\ e^*\ \text{end end} : ti_1^*; l_1; \Gamma_1, (t_2\ \alpha_2)^n; \phi_1, (=\alpha_2\ (t_2\ c))^n\epsilon; l_1; \Gamma_1, (t_2$
$\alpha_2\ (t_2\ c))^n \rightarrow (t_4\ \alpha_5)^m; l_4; \Gamma_1, (t_4\ \alpha_5)^m; \phi_1 \cup \phi_4[\alpha_4 \mapsto \alpha_5][\alpha_3 \mapsto \alpha_2])$ by Rule Local.
Finally,

$$\begin{aligned} &S; C \vdash \text{local}_m\{j; v^n\ (t.\text{const}\ 0)^k\}\ \text{block}\ tfi\ e^*\ \text{end end} \\ &\quad :\ ti_1^*; l_1; \Gamma_1, (t_2\ \alpha_2)^n; \phi_1, (=\alpha_2\ (t_2\ c))^n\epsilon; l_1; \Gamma_1, (t_2\ \alpha_2)^n; \phi_1, (=\alpha_2\ (t_2\ c))^n \\ &\quad \rightarrow ti_1^*\ (t_4\ \alpha_5)^m; l_4; \Gamma_1, (t_4\ \alpha_5)^m; \phi_1 \cup \phi_4[\alpha_4 \mapsto \alpha_5][\alpha_3 \mapsto \alpha_2]) \end{aligned}$$

by Rule Stack-Poly.

- Case: $S; C \vdash \text{local}_n\{i; v_l^*\}\ v^n\ \text{end} : \epsilon; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$
  $\wedge\ \text{local}_n\{i; v_l^*\}\ v^n\ \text{end} \hookrightarrow_j\ v^n$
  We want to show that $v^n : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma'; \phi'$, where $\Gamma' \vdash \phi' \Rightarrow \phi_2$.
  First we derive the type of v^n from the precondition of the local.
  We have $S; (ti^n; \phi_3) \vdash_i v_l^*; v^n : (t\ \alpha)^n; l_3; \Gamma_3; \phi_3$, as it is a premise of Rule Local that we have assumed to hold.
  By Lemma Inversion-On-Instruction-Typing on Rule Local, $ti_2^* = ti_1^*\ (t\ \alpha)^n$, $l_1 = l_2$, $\Gamma_3 = \Gamma_1, ti^n$, and $\phi_2 = \phi_1 \cup \phi_3$.
  $(\vdash v_l : ti_l; \phi_l)^*$ and $S; C_l \vdash v^n : \epsilon : ti_l^*; \emptyset, ti_l^*; \phi_l^* \rightarrow (t\ \alpha)^n; l_3; \Gamma_3; \phi_3$ because they are premises of Rule Code which we have assumed to hold.

$ti_l = (t_l\ \alpha_l)^*$, and $\phi_l^* = \emptyset, (= \alpha_l\ (t_l\ c_l))^*$ because they are premises of Rule Admin-Const which we have assumed to hold.
By Lemma Inversion-On-Instruction-Typing on Rule Const, $\Gamma_3 = ti_l^*, (t\ \alpha)^n$, $\phi_3 = \phi_l^*, (= \alpha\ (t\ c))^n$, where $v^n = t.\mathsf{const}\ c$.
$S; C \vdash v^n : \epsilon; l_1; \Gamma_1; \phi_1 \rightarrow (t\ \alpha)^n; l_1; \Gamma_1, (t\ \alpha)^n; \phi_1, (= \alpha\ (t\ c))^n$ by Rule Const and Rule Composition.
By Lemma Well-formedness, $l_1 \subset \Gamma_1$, so $(t\ \alpha)^n, l_1 \subset \Gamma_1, (t\ \alpha)^n$.
Trivially, $\Gamma_1, \Gamma_v, (t_l\ \alpha_l)^* \vdash \phi_1, (= \alpha\ (t\ c))^n, (= \alpha_l\ (t_l\ c_l))^* \Rightarrow \phi_2$ since $\phi_2 = \phi_1 \cup \phi_3$ and $\phi_3 = \emptyset, (= \alpha_l\ (t_l\ c_l))^*, (= \alpha\ (t\ c))^n$.
Finally, $S; C \vdash v^n : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_1^*\ (t\ \alpha)^n; l_1; \Gamma_1, (t\ \alpha)^n; \phi_1, (= \alpha\ (t\ c))^n$, by Rule Stack-Poly.

- Case: $S; C \vdash \mathsf{local}_n\{i; v_l^*\}\ \mathsf{trap\ end} : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$
$\wedge\ \mathsf{local}_n\{i; v_l^*\}\ \mathsf{trap\ end} \hookrightarrow\ \mathsf{trap}$
Trivially, $S; C \vdash \mathsf{trap} : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$ by Rule Trap.

- Case: $S; C \vdash \mathsf{local}_n\{i; v_l^*\}\ L^k[(t.\mathsf{const}\ c)^n\ \mathsf{return}]\ \mathsf{end} : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$
$\wedge\ \mathsf{local}_n\{i; v_l^*\}\ L^k[(t.\mathsf{const}\ c)^n\ \mathsf{return}]\ \mathsf{end} \hookrightarrow_j\ (t.\mathsf{const}\ c)^n$
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.
$ti_2^* = ti_1^*\ (t\ \alpha)^n, l_1 = l_2, \Gamma_2 = \Gamma_1 \cup \Gamma_3, \phi_2 = \phi_1 \cup \phi_3, S; ((t\ \alpha)^n; \phi_3) \vdash_i v_l^*; L^k[(t.\mathsf{const}\ c)^n\ \mathsf{return}] : ti_3^n; l_3; \Gamma_3; \phi_3$ by Lemma Inversion-On-Instruction-Typing on $S; C \vdash \mathsf{local}_n\{i; v_l^*\}\ L^k[(t.\mathsf{const}\ c)^n\ \mathsf{return}]$ $ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$.
$(\vdash v_l : ti_l; \phi_l)^*$ and $S; C_l \vdash L^k[(t.\mathsf{const}\ c)^n\ \mathsf{return}] : \epsilon; ti_l^*; \emptyset, ti_l^*; \phi_l^* \rightarrow (t\ \alpha)^n; l_3; \Gamma_3; \phi_3$, where $C_l = C, \mathrm{local}\ t^*, \mathrm{return}\ ((t\ \alpha)^n; \phi_3)$, by inversion on the Code judgment.
$ti_l^* = (t_l\ \alpha_l)^*$ because it is a premise of Rule Admin-Const which we have assumed to hold.
By Lemma Inversion-On-Instruction-Typing on Rule Composition and Rule Return, $S; C_l \vdash (t.\mathsf{const}\ c)^n : ti_4^*; l_4; \Gamma_4; \phi_4 \rightarrow ti_4^*\ (t\ \alpha_5)^n; l_3; \Gamma_5; \phi_5, S; C_l \vdash \mathsf{return} : ti_4^*\ (t_3\ \alpha_5)^n; l_3; \Gamma_5; \phi_5 \rightarrow ti_\emptyset^*; l_\emptyset; \Gamma_\emptyset; \phi_\emptyset, ti_3^n \notin \Gamma_1$ and $\Gamma_5 \vdash \phi_5 \Rightarrow \phi_3[\alpha_3 \rightarrow \alpha_5]$, where $(t\ \alpha_3) = ti_3^n$.
By Lemma Inversion-On-Instruction-Typing on $S; C_l \vdash (t.\mathsf{const}\ c)^n : ti_4^*; l_4; \Gamma_4; \phi_4 \rightarrow ti_4^*\ (t_3\ \alpha_5)^n; l_3; \Gamma_5; \phi_5$, we have $l_4 = l_3$, $\Gamma_5 = \Gamma_4, (t\ \alpha)^n$, $\phi_5 = \phi_4, (= \alpha_5\ (t\ c))^n$, and $S; C_l \vdash (t.\mathsf{const}\ c)^n : \epsilon; l_4; \Gamma_4; \phi_4 \rightarrow (t_3\ \alpha_5)^n; l_3; \Gamma_5; \phi_5$.
Then, $S; C \vdash (t.\mathsf{const}\ c)^n : \epsilon; l_1; \Gamma_1, (t_1\ \alpha_1)^*; \phi_1, (= \alpha_\emptyset\ (t_\emptyset\ c_\emptyset))^* \rightarrow (t_3\ \alpha_5)^n; l_3; \Gamma_5; \phi_5$ by Lemma Lift-Consts.
We have $S; C \vdash (t.\mathsf{const}\ c)^n : ti_1^*; l_1; \Gamma_1, (t_1\ \alpha_1)^*; \phi_1, (= \alpha_\emptyset\ (t_\emptyset\ c_\emptyset))^* \rightarrow ti_1^*\ (t_3\ \alpha_5)^n; l_2; \Gamma_5; \phi_5$ by Rule Stack-Poly.
Further, $S; C \vdash (t.\mathsf{const}\ c)^n : ti_1^*; l_1; \Gamma_1, (t_1\ \alpha_1)^*; \phi_1, (= \alpha_\emptyset\ (t_\emptyset\ c_\emptyset))^* \rightarrow (t_3\ \alpha_5)^n; l_3; \Gamma_5 \cup \Gamma_1; \phi_5 \cup \phi_1$ by Lemma Threading-Constraints, since $\Gamma_1, (t_1\ \alpha_1)^* \cup \Gamma_1 = \Gamma_1, (t_1\ \alpha_1)^*$, and $\phi_1, (= \alpha_\emptyset\ (t_\emptyset\ c_\emptyset))^* \cup \phi_1 = \phi_1, (= \alpha_\emptyset\ (t_\emptyset\ c_\emptyset))^*$.
We have $\Gamma_5 \vdash \phi_5 \Rightarrow \phi_3[\alpha_3 \mapsto \alpha_5]$, so $\Gamma_5 \cup \Gamma_1 \vdash \phi_5 \cup \phi_1 \Rightarrow (\phi_3[\alpha_3 \mapsto \alpha_5]) \cup \phi_1$.
Then, since $(t_3\ \alpha_3)^n \notin \Gamma_1$ are fresh, we can perform renaming to get $S; C \vdash (t.\mathsf{const}\ c)^n : ti_1^*; l_1; \Gamma_1, (t_1\ \alpha_1)^*; \phi_1, (= \alpha_\emptyset\ (t_\emptyset\ c_\emptyset))^* \rightarrow ti_1^*\ (t_3\ \alpha_3)^n; l_3; \Gamma_5 \cup \Gamma_1; \phi_5 \cup \phi_1$

- Case: $S; C \vdash v\ (\mathsf{tee\_local}\ j) : \epsilon; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$
$\wedge\ v\ (\mathsf{tee\_local}\ j) \hookrightarrow v\ v\ (\mathsf{set\_local}\ j)$
Note that the reduction of tee_local 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.
As usual, we start by figuring out what $\epsilon; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$ looks like.

By Lemma Inversion-On-Instruction-Typing on Rule Composition, we know that $S; C \vdash v : \epsilon; l_1; \Gamma_1; \phi_1 \rightarrow ti_3^*; l_3; \Gamma_3; \phi_3$, and $S; C \vdash \text{tee\_local } j : ti_3^*; l_3; \Gamma_3; \phi_3 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$.
By Lemma Inversion-On-Instruction-Typing on Rule Tee-Local, we also know that $ti_3^* = ti^* \ (t\ \alpha)$, $ti_2^* = ti^* \ (t\ \alpha_2)$, $l_2 = l_3[j := (t\ \alpha)]$, $\Gamma_2 = \Gamma_3, (t\ \alpha_2)$, and $\phi_2 = \phi_3, (= \alpha\ \alpha_2)$.
Then, by Lemma Inversion-On-Instruction-Typing on Rule Const, $t.\text{const } c = v$, $ti^* = \epsilon$, $l_1 = l_3, \Gamma_3 = \Gamma_1, (t\ \alpha)$, and $\phi_3 = \phi_1, (= \alpha\ (t\ c))$.
Now, we can show that $v\ v\ (\text{set\_local } j) : \epsilon; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_1, (= \alpha\ (t\ c)), (= \alpha\ \alpha_2)$ and $\Gamma_2 \vdash \phi_1, (= \alpha\ (t\ c)), (= \alpha\ \alpha_2) \implies \phi_2$.
By Rule Const and Rule Composition, $S; C \vdash v\ v : \epsilon; l_1; \Gamma_1; \phi_1 \rightarrow (t\ \alpha_2)\ (t\ \alpha); l_1; \Gamma_1, (t\ \alpha_2), (t\ \alpha); \phi_1, (= \alpha_2\ (t\ c)), (= \alpha\ (t\ c))$.
By Rule Set-Local, $S; C \vdash \text{set\_local } j : (t\ \alpha); l_1; \Gamma_1; \phi_2 \rightarrow \epsilon; l_1[j := (t\ \alpha)]; \Gamma_1, (t\ \alpha_2), (t\ \alpha); \phi_1, (= \alpha_2\ (t\ c)), (= \alpha\ (t\ c))$.
By Rule Composition, $S; C \vdash v\ v\ (\text{set\_local } j) : \epsilon; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_1[j := (t\ \alpha)]; \Gamma_1, (t\ \alpha_2), (t\ \alpha); \phi_1, (= \alpha_2\ (t\ c)), (= \alpha\ (t\ c))$.
By Lemma Well-formedness, $l_1 \subset \Gamma_1$, so $(t\ \alpha_2), l_1[j := \alpha] \subset \Gamma_1, (t\ \alpha), (t\ \alpha_2)$.
Finally, $\Gamma_1, (t\ \alpha), (t\ \alpha_2) \vdash \phi_1, (= \alpha_2\ (t\ c)), (= \alpha\ (t\ c)) \Rightarrow \phi_1, (= \alpha\ (t\ c)), (= \alpha\ \alpha_2)$ trivially.

- Case: $S; C \vdash \text{get\_local } j : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$
$\wedge\ s; v_1^j\ (t.\text{const } c)\ v_2^k; \text{get\_local } j \hookrightarrow s; v_1^j\ (t.\text{const } c)\ v_2^k; (t.\text{const } c)$
We know $l_1 = (t_1\ \alpha_1)^j\ (t\ \alpha)\ (t_2\ \alpha_2)^j$, where $\vdash t.\text{const } c : (t\ \alpha); \emptyset, (= \alpha\ (t\ c))$ and $(= \alpha\ (t\ c)) \in \phi_1$ as it is one of our assumptions.
Then, $S; C \vdash \text{get\_local } j : \epsilon; l_1; \Gamma_1; \phi_1 \rightarrow (t\ \alpha_2); l_1; \Gamma_1, (t\ \alpha_2); \phi_1, (= \alpha_2\ \alpha)$, $ti_2^* = ti_1^*\ (t\ \alpha_2)$, $l_1 = l_2, \Gamma_2 = \Gamma_1, (t\ \alpha_2)$, and $\phi_2 = \phi_1, (= \alpha_2\ \alpha)$ by Lemma Inversion-On-Instruction-Typing on $S; C \vdash \text{get\_local } j : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$.
We have $S; C \vdash (t.\text{const } c) : \epsilon; l_1; \Gamma_1; \phi_1 \rightarrow (t\ \alpha_2); l_1; \Gamma_1, (t\ \alpha_2); \phi_1, (= \alpha_2\ (t\ c))$ by Rule Const.
Then, $S; C \vdash (t.\text{const } c) : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_1^*\ (t\ \alpha_2); l_1; \Gamma_1, (t\ \alpha_2); \phi_1, (= \alpha_2\ (t\ c))$ by Rule Stack-Poly.
Finally, since $(= \alpha\ (t\ c)) \in \phi_1, \Gamma_1, (t\ \alpha_2) \vdash \phi_1, (= \alpha_2\ (t\ c)) \Rightarrow \phi_1, (= \alpha_2\ \alpha)$.

- Case: $S; C \vdash (t'.\text{const } c')\ \text{set\_local } j : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$
$\wedge\ s; v_1^j\ (t.\text{const } c)\ v_2^k; (t.\text{const } c')\ \text{set\_local } j \hookrightarrow s; v_1^j\ (t.\text{const } c')\ v_2^k; \epsilon$
We know $l_1 = (t_1\ \alpha_1)^j\ (t\ \alpha)\ (t_2\ \alpha_2)^k$, where $\vdash v_1^j\ (t.\text{const } c)\ v_2^k : (t\ \alpha); (= \alpha\ (t\ c))$ and $(= \alpha\ (t\ c)) \in \phi_1$ as it is one of our assumptions.
Then, $S; C \vdash (t'.\text{const } c') : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_3^*; l_3; \Gamma_3; \phi_3$, $S; C \vdash \text{set\_local } j : ti_3^*; l_3; \Gamma_3; \phi_3 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$ by Lemma Inversion-On-Instruction-Typing on $S; C \vdash (t'.\text{const } c')\ \text{set\_local } j : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$.
Then, $C_{\text{local}}(j) = t$, $ti_3^* = ti_2^*\ (t\ a')$, $\Gamma_3 = \Gamma_2$, $\phi_3 = \phi_1$, and $l_2 = l_3[j := (t\ \alpha')]$ by Lemma Inversion-On-Instruction-Typing on
$S; C \vdash \text{set\_local } j : ti_3^*; l_3; \Gamma_3; \phi_3 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$.
Further, $ti_3^* = ti_1^*\ (t'\ \alpha')$, $l_1 = l_3$, $\Gamma_3 = \Gamma_1, (t'\ \alpha')$, $\phi_3 = \phi_1, (= \alpha'\ (t'\ c'))$, by Lemma Inversion-On-Instruction-Typing on
$S; C \vdash (t'.\text{const } c') : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_3^*; l_3; \Gamma_3; \phi_3$.
Since $ti_3^* = ti_1^*\ (t'\ \alpha')$ and $ti_3^* = ti_1^*(t\ \alpha')$, $t' = t$.
Then, $S; C \vdash \epsilon : \epsilon; l_2; \Gamma_2; \phi_2 \rightarrow \epsilon; l_2; \Gamma_2; \phi_2$ by Rule Empty.
Further, $S; C \vdash \epsilon : ti_1^*; l_2; \Gamma_2; \phi_2 \rightarrow ti_1^*; l_2; \Gamma_2; \phi_2$ by Rule Stack-Poly.

Note that since ϕ_1 contains the equality constraints for all the locals except for index j, the only novel case is for the modified local.
Then $\vdash (t.\text{const } c') : (t\ \alpha'); \emptyset, (= \alpha'\ (t\ c'))$ by Rule Admin-Const.
Finally, since $\phi_2 = \phi_1, (= \alpha'\ (t\ c')), \phi_2 = \phi_1 \bigcup (\emptyset, (= \alpha'\ (t\ c')))$.

- Case: $S; C \vdash \text{get\_global } j : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$
$\wedge\ s; v^*; \text{get\_global } j \hookrightarrow_i s; v^*; s_{\text{glob}}(i, j)$
We know $ti_2^* = ti_1^*\ (t\ \alpha)$, $l_1 = l_2$, $C_{\text{global}}(j) = \text{mut}^? t$, $\Gamma_2 = \Gamma_1, (t\ \alpha)$, and $\phi_1 = \phi_2$ by Lemma Inversion-On-Instruction-Typing on
$S; C \vdash \text{get\_global } j : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$.
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 $C_{\text{global}}(j) = \text{mut}^? t$, then $\vdash s_{\text{glob}}(i, j) : (t\ \alpha_\emptyset); \phi_\emptyset$ 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.
Then $S; C \vdash (t.\text{const } c) : \epsilon; l_1; \Gamma_1; \phi_1 \longrightarrow (t\ \alpha); l_1; \Gamma_1, (t\ \alpha); \phi_1, (= \alpha\ (t\ c))$, where $t.\text{const } c = s_{\text{glob}}(i, j)$, by Rule Const.
Further $S; C \vdash (t.\text{const } c) : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_1^*\ (t\ \alpha); l_1; \Gamma_1, (t\ \alpha); \phi_1, (= \alpha\ (t\ c))$ by Rule Stack-Poly.
Finally, $\Gamma_1, (t\ \alpha) \vdash \phi_1, (= \alpha\ (t\ c)) \Longrightarrow \phi_1$ trivially.

- Case: $S; C \vdash (t.\text{const } c)\ (\text{set\_global } j) : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$
$\wedge\ s; v^*; (t.\text{const } c)\ (\text{set\_global } j) \hookrightarrow_i s'; v^*; \epsilon$, where $s' = s$ with $\text{glob}(i, j) = (t.\text{const } c)$
We know $S; C \vdash (t.\text{const } c) : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_3^*; l_3; \Gamma_3; \phi_3, S; C \vdash \text{set\_global } j : ti_3^*; l_3; \Gamma_3; \phi_3 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$, by Lemma Inversion-On-Instruction-Typing on $S; C \vdash (t.\text{const } c)\ (\text{set\_global } j) : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$.
Then, $ti_3^* = ti_1^*\ (t\ \alpha)$, $l_1 = l_3$, $\Gamma_3 = \Gamma_1, (t\ \alpha)$, $\phi_3 = \phi_1, (= \alpha\ (t\ c))$, and $\alpha \notin \Gamma_1$, by Lemma Inversion-On-Instruction-Typing on $S; C \vdash (t.\text{const } c) : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_3^*; l_3; \Gamma_3; \phi_3$.
Then, $ti_3^* = ti_2^*\ (t\ \alpha)$, $l_3 = l_2$, $\Gamma_2 = \Gamma_3$, $\phi_2 = \phi_3$, $C_{\text{global}} = \text{mut } t$, by Lemma Inversion-On-Instruction-Typing on $S; C \vdash \text{set\_global } j : ti_3^*; l_3; \Gamma_3; \phi_3 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$.
Now we construct a type for ϵ.
We have $S; C \vdash \epsilon : \epsilon; l_1; \Gamma_1, (t\ \alpha); \phi_1, (= \alpha\ (t\ c)) \longrightarrow \epsilon; l_1; \Gamma_1, (t\ \alpha); \phi_1, (= \alpha\ (t\ c))$ by Rule Empty.
Then, $S; C \vdash \epsilon : ti_1^*; l_1; \Gamma_1, (t\ \alpha); \phi_1, (= \alpha\ (t\ c)) \longrightarrow ti_2^*; l_1; \Gamma_1, (t\ \alpha); \phi_1, (= \alpha\ (t\ c))$ by Rule Stack-Poly.
Now we must ensure that the new store s' is well typed: $\vdash s' : S$.
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 $C_{\text{global}}(j) = \text{mut } t$ and $s_{\text{glob}}(i, j) = (t.\text{const } c')$, where $\vdash (t.\text{const } c') : (t\ \alpha_\emptyset); \emptyset, (= \alpha_\emptyset\ (t\ c'))$ because it is a premise of Rule Instance that we have assumed to hold.
We know $\vdash$: $(t.\text{const } c) : (t\ \alpha); \emptyset, (= \alpha\ (t\ c))$ by Rule Admin-Const.
Therefore $\vdash s' : S$ by Rule Store.

- Case: $S; C \vdash (\text{i32.const } k)\ (t.\text{load } a\ o) : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$
$\wedge\ s; v^*; (\text{i32.const } k)\ (t.\text{load } a\ o) \hookrightarrow_i s; v^*; t.\text{const const}_t(b^*)$, where $s_{\text{mem}}(i, k + o, |t|) = b^*$
We know $ti_2^* = ti_1^*\ (t\ \alpha)$, $l_1 = l_2$, $\Gamma_2 = \Gamma_1, (t\ \alpha_k), (t\ \alpha)$, $\phi_2 = \phi_1, (= \alpha_k\ (\text{i32}\ k))$, and $\alpha_k \notin \Gamma_1$, by Lemma Inversion-On-Instruction-Typing on $S; C \vdash (\text{i32.const } k)\ (t.\text{load } a\ o) : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$.

Let $c = \text{const}_t(b^*)$. Although note that the actual value of c is irrelevant for the rest of the proof case.
Now we can restruct a type for $t.\text{const}\ c$.
We have $S; C \vdash t.\text{const}\ c : \epsilon; l_1; \Gamma_1, (\text{i32}\ \alpha_k); \phi_1, (= \alpha_k\ (\text{i32}\ k)) \longrightarrow (t\ \alpha); l_1; \Gamma_1, (\text{i32}\ \alpha_k), (t\ \alpha); \phi_1, (=$ $\alpha_k\ (\text{i32}\ k)), (= \alpha\ (t\ c))$ by Rule Const.
Then, $S; C \vdash t.\text{const}\ c : ti_1^*; l_1; \Gamma_1, (\text{i32}\ \alpha_k); \phi_1, (= \alpha_k\ (\text{i32}\ k)) \longrightarrow ti_1^*\ (t\ \alpha); l_1; \Gamma_1, (\text{i32}\ \alpha_k), (t\ \alpha); \phi_1, (=$ $\alpha_k\ (\text{i32}\ k)), (= \alpha\ (t\ c))$ by Rule Stack-Poly.
Finally, $\Gamma_1, (t\ \alpha_k), (t\ \alpha) \vdash \phi_1, (= \alpha_k\ (\text{i32}\ k)), (= \alpha\ (t\ c)) \Rightarrow \phi_1, (= \alpha_k\ (\text{i32}\ k))$ trivially.

- Case: $S; C \vdash (\text{i32.const}\ k)\ (t.\text{load}\checkmark\ a\ o) : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$
  $\wedge\ s; v^*; (\text{i32.const}\ k)\ (t.\text{load}\checkmark\ tp\_sx^?\ a\ o) \hookrightarrow_i s; t.\text{const}\ \text{const}_t(b^*)$, where $s_{\text{mem}}(i, k{+}o, |t|) =$ b^*
  Proceeds the same as the above case, the reduction of $\text{load}\checkmark$ is equivalent to a successful load.

- Case: $S; C \vdash (\text{i32.const}\ k)\ (t.\text{load}\ tp\_sx\ a\ o) : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$
  $\wedge\ s; v^*; (\text{i32.const}\ k)\ (t.\text{load}\ tp\_sx\ a\ o) \hookrightarrow_i s; t.\text{const}\ \text{const}_t^{sx}(b^*)$, where $s_{\text{mem}}(i, k{+}o, |tp|) =$ b^*
  This case is the same as the above two cases except that tp_sx is present in the load instruction.
  Similar to above case, except with $|tp|$ replacing $|t|$ and $\text{const}_t^{sx}(b^*)$ instead of $\text{const}_t(b^*)$.

- Case: $S; C \vdash (\text{i32.const}\ k)\ (t.\text{load}\checkmark\ tp\_sx\ a\ o) : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$
  $\wedge\ s; v^*; (\text{i32.const}\ k)\ (t.\text{load}\checkmark\ tp\_sx^?\ a\ o) \hookrightarrow_i s; t.\text{const}\ \text{const}_t(b^*)$, where $s_{\text{mem}}(i, k{+}o, |tp|) =$ b^*
  Proceeds the same as the above case.

- Case: $S; C \vdash (\text{i32.const}\ k)\ (t.\text{load}\ tp\_sx^?\ a\ o) : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$
  $\wedge\ s; v^*; (\text{i32.const}\ k)\ (t.\text{load}\ tp\_sx^?\ a\ o) \hookrightarrow_i s; v^*; \text{trap}$
  Trivially, we have $S; C \vdash \text{trap} : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$ by Rule Trap.

- Case: $S; C \vdash (\text{i32.const}\ k)\ (t.\text{const}\ c)\ (t.\text{store}\ a\ o) : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$
  $\wedge\ s; (\text{i32.const}\ k)\ (t.\text{const}\ c)\ (t.\text{store}\ a\ o) \hookrightarrow_i s'; \epsilon$, where $s' = s$ with $\text{mem}(i, k + o, |t|) =$ $\text{bits}_t^{|t|}(c)$
  We know $ti_1^* = ti_2^*$, $l_1 = l_2$, $\Gamma_2 = \Gamma_1, (\text{i32}\ \alpha_k), (t\ \alpha_c)$, $\phi_2 = \phi_1, (= \alpha_k\ (\text{i32}\ k)), (= \alpha_c\ (t\ c))$, $C_{\text{memory}} = n$, and $\alpha_k, \alpha_c \notin \Gamma_1$ by Lemma Inversion-On-Instruction-Typing on $S; C \vdash (\text{i32.const}\ k)\ (t.\text{const}$ $ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$.
  Now we construct a type for ϵ.
  We have $S; C \vdash \epsilon : \epsilon; l_1; \Gamma_1, (\text{i32}\ \alpha_k), (t\ \alpha_c); \phi_1, (= \alpha_k\ (\text{i32}\ k)), (= \alpha_c\ (t\ c)) \longrightarrow \epsilon; l_1; \Gamma_1, (\text{i32}\ \alpha_k), (t\ \alpha_c); \phi_1,$ $\alpha_k\ (\text{i32}\ k)), (= \alpha_c\ (t\ c))$ by Rule Empty.
  Now we must ensure that the new store s' is well typed: $\vdash s' : S$.
  Recall $\vdash s : S$ and $C_{\text{memory}} = n$, 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 $|s'_{\text{mem}}(i)| = |s_{\text{mem}}(i)|$, and therefore $n \leq |s'_{\text{mem}}(i)|$, so $s' : S$ by Rule Store.

- Case: $S; C \vdash (\text{i32.const}\ k)\ (t.\text{const}\ c)\ (t.\text{store}\checkmark\ a\ o) : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$
$\wedge\ s; (\text{i32.const}\ k)\ (t.\text{const}\ c)\ (t.\text{store}\checkmark\ a\ o) \hookrightarrow_i s'; \epsilon$, where $s' = s$ with $\text{mem}(i, k + o, |t|) = \text{bits}_t^{|t|}(c)$
Proceeds the same as the above case, the reduction of store$\checkmark$ is equivalent to a successful store.

- Case: $S; C \vdash (\text{i32.const}\ k)\ (t.\text{const}\ c)\ (t.\text{store}\ tp\ a\ o) : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$
$\wedge\ s; v^*; (\text{i32.const}\ k)\ (t.\text{const}\ c)\ (t.\text{store}\ tp\ a\ o) \hookrightarrow_i s'; v^*; \epsilon$, where $s' = s$ with $\text{mem}(i, k + o, |tp|) = \text{bits}_t^{|tp|}(c)$
This case is the same as the above two cases except that tp is present in the store instruction. Similar to above case, except with $|tp|$ replacing $|t|$ and $\text{const}_t^{sx}(b^*)$ instead of $\text{const}_t(b^*)$.

- Case: $S; C \vdash (\text{i32.const}\ k)\ (t.\text{const}\ c)\ (t.\text{store}\checkmark\ tp\ a\ o) : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$
$\wedge\ s; v^*; (\text{i32.const}\ k)\ (t.\text{const}\ c)\ (t.\text{store}\checkmark\ tp\ a\ o) \hookrightarrow_i s'; v^*; \epsilon$, where $s' = s$ with $\text{mem}(i, k + o, |tp|) = \text{bits}_t^{|tp|}(c)$
Proceeds the same as above.

- Case: $S; C \vdash (\text{i32.const}\ k)\ (t.\text{const}\ c)\ (t.\text{store}\ tp^?\ a\ o) : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$
$\wedge\ s; v^*; (\text{i32.const}\ k)\ (t.\text{const}\ c)\ (t.\text{store}\ tp^?\ a\ o) \hookrightarrow_i s; v^*; \text{trap}$
Trivially, we have $S; C \vdash \text{trap} : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$ by Rule Trap.

- Case: $S; C \vdash \text{current\_memory} : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$
$\wedge\ s; v^*; \text{current\_memory} \hookrightarrow_i s; v^*; \text{i32.const}\ |s_{\text{mem}}(i, *)|/64\text{Ki}$
We know $ti_2^* = ti_1^*\ (\text{i32}\ \alpha)$, $l_1 = l_2$, $\Gamma_2 = \Gamma_1, (\text{i32}\ \alpha)$, $\phi_2 = \phi_1$, and $\alpha \notin \Gamma_1$ by Lemma Inversion-On-Instruction-Typing on $S; C \vdash \text{current\_memory} : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$.
Now we can construct a type for the reduced program.
Let $c = |s_{\text{mem}}(i, *)|/64\text{Ki}$. Although note that the actual value of c is irrelevant to the rest of the proof case.
We have $S; C \vdash \text{i32.const}\ c : \epsilon; l_1; \Gamma_1; \phi_1 \longrightarrow (\text{i32}\ \alpha); l_1; \Gamma_1, (\text{i32}\ \alpha); \phi_1, (= \alpha\ (\text{i32}\ c))$ by Rule Const.
Then, $S; C \vdash \text{i32.const}\ c : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_1^*\ (\text{i32}\ \alpha); l_1; \Gamma_1, (\text{i32}\ \alpha); \phi_1, (= \alpha\ (\text{i32}\ c))$ by Rule Stack-Poly.
Finally, $\Gamma_1, (\text{i32}\ \alpha) \vdash \phi_1, (= \alpha\ (\text{i32}\ c)) \Rightarrow \phi_1$ trivially.

- Case: $S; C \vdash (\text{i32.const}\ k)\ \text{grow\_memory} : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$
$\wedge\ s; v^*; (\text{i32.const}\ k)\ \text{grow\_memory} \hookrightarrow_i s'; v^*; \text{i32.const}\ |s_{\text{mem}}(i, *)|/64\text{Ki}$,
where $s' = s$ with $\text{mem}(i, *) = s_{\text{mem}}(i, *)(0)^{k \cdot 64\text{Ki}}$
We know $ti_2^* = ti_1^*\ (\text{i32}\ \alpha)$, $l_1 = l_2$, $\Gamma_2 = \Gamma_1, (\text{i32}\ \alpha_k), (\text{i32}\ \alpha)$, $\phi_2 = \phi_1, (= \alpha_k\ (\text{i32}\ k))$, $C_{\text{memory}} = n$, and $\alpha_k, a \notin \Gamma_1$ by Lemma Inversion-On-Instruction-Typing on $S; C \vdash (\text{i32.const}\ k)\ \text{grow\_memory} : ti_1^*; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$.
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.
Now we will construct a type for the reduced instruction sequence.
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.
We have $S; C \vdash \text{i32.const}\ c : \epsilon; l_1; \Gamma_1, (\text{i32}\ \alpha_k); \phi_1, (= \alpha_k\ (\text{i32}\ k)) \longrightarrow (\text{i32}\ \alpha); l_1; \Gamma_1, (\text{i32}\ \alpha_k), (\text{i32}\ \alpha); \phi_1, (= \alpha_k\ (\text{i32}\ k)), (= \alpha\ (\text{i32}\ c))$ by Rule Const.
Then, $\Gamma_1, (\text{i32}\ \alpha_k), (\text{i32}\ \alpha) \vdash \phi_1, (= \alpha_k\ (\text{i32}\ k)), (= \alpha\ (\text{i32}\ c)) \Rightarrow \phi_1, (= \alpha_k\ (\text{i32}\ k))$ trivially.

Now we must ensure that the new store s' is well typed: $\vdash s' : S$.
Recall that we assumed $\vdash s : S$ and that we have $C_{\text{memory}} = n$, 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: $S; C \vdash (\text{i32.const } k)\ \text{grow\_memory} : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$
$\wedge\ s; (\text{i32.const } k)\ \text{grow\_memory} \hookrightarrow_i \text{i32.const } (-1)$
Same as above case since the value of c is irrelevant (and can therefore be -1).

- Case: $S; C \vdash \text{local}_n\{i; v^*\}\ e^* : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$
$\wedge\ s; v_0^*; \text{local}_n\{i; v^*\}\ e^* \hookrightarrow_j s'; v_0^*; \text{local}_n\{i; v'^*\}\ e'^*$
where $s; v^*; e^* \hookrightarrow_i s'; v'^*; e'^*$
First, we will derive the type of the body of the local block.
We know $(\vdash v : (t_v\ \alpha_v); \phi_v)^*$, and $S; (ti^n; \phi) \vdash_i v^*; e^* : ti^n; l_3; \Gamma_3; \phi_3$, where $ti_2^* = ti_1^*\ ti^n$ and $\phi_2 = \phi_1 \cup \phi_3$ by Lemma Inversion-On-Instruction-Typing on $S; C \vdash \text{local}_n\{i; v^*\}\ e^* : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$.
Then $S; S_{\text{inst}}(i), \text{local } t_v^*, \text{return } (ti^n; \phi) \vdash e^* : \epsilon; (t_v\ \alpha_v)^*; \emptyset, (t_v\ \alpha_v)^*; \phi_v^* \rightarrow ti^n; l_3; \Gamma_3; \phi_4$, where $\Gamma_3 \vdash \phi_4 \Rightarrow \phi_3$ because it is a premise of Rule Code.
Now, we invoke the inductive hypothesis and use it to rebuild the original type.
Since $S; S_{\text{inst}}(i), \text{return}(ti^n; \phi) \vdash e^* : \epsilon; (t_v\ \alpha_v)^*; \emptyset, (t_v\ \alpha_v)^*; \phi_v^* \rightarrow ti^n; l_3; \Gamma_3; \phi$, $s \vdash S$, $(\vdash v : (t_v\ \alpha_v); \phi_v)^*$, and $s; v^*; e^* \hookrightarrow_i s'; v'^*; e'^*$, then by the inductive hypothesis we know that $\vdash s' : S$ and $S; S_{\text{inst}}(i), \text{return}(ti^n; \phi) \vdash e'^* : \epsilon; (t_v\ \alpha'_v); \Gamma_4; \phi_4 \rightarrow ti^n; l_3; \Gamma_3; \phi_3$, where $(\vdash v' : (t_v\ \alpha'_v); \phi'_v)^*$, $(t_v\ \alpha'_v) \subset \Gamma_3$, and $(\phi'_v \subset \phi_4)^*$.
We know, $S; (ti^n); \phi) \vdash_i v'^*; e'^* : ti^n; l_3; \Gamma_3; \phi_3$ by Rule Code.
Then, $S; C \vdash \text{local}_n\{i; v'^*\}\ e'^* : \epsilon; l_1; \Gamma_1; \phi_1 \rightarrow ti^n; l_2; \Gamma_2; \phi_2$ by Rule Local.
Thus, $S; C \vdash \text{local}_n\{i; v'^*\}\ e'^* : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$ by Rule Stack-Poly, since $ti_2^* = ti_1^*\ ti^n$

- Case: $S; C \vdash L^k[e^*] : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$
$\wedge\ s; v^*; L^k[e^*] \hookrightarrow_i s'; v'^*; L^k[e'^*]$
where $s; v^*; e^* \hookrightarrow_i s'; v'^*; e'^*$
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: $k = 0$ Expanding $L^0[e^*]$, we get $v_0^*\ e^*\ e_0^*$.
  (1) By Lemma Sequence-Decomposition on $S; C \vdash v_0^*\ e^*\ e_0^* : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$ we have
    (a) $S; C \vdash v_0^* : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_3^*; l_3; \Gamma_3; \phi_3$
    (b) $S; C \vdash e^*\ e_0^* : ti_3^*; l_3; \Gamma_3; \phi_3 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$, and therefore, by Lemma Sequence-Decomposition
    (c) $S; C \vdash e^* : ti_3^*; l_3; \Gamma_3; \phi_3 \rightarrow ti_4^*; l_4; \Gamma_4; \phi_4$
    (d) $S; C \vdash e_0^* : ti_4^*; l_4; \Gamma_4; \phi_4 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$

Now, we invoke the outer inductive hypothesis (for Lemma 24) and rebuild the type using the reduced expression.

(2) By 1c and the inductive hypothesis (for Lemma 24) we know that
  (a) $s; v^*; e^* \hookrightarrow_i s'; v'^*; e'^*$,
  (b) $\vdash s' : S$,
  (c) $S; C \vdash e'^* : ti_3^*; l_5; \Gamma_3, (t\ \alpha)^*; \phi_3, (= \alpha\ (t\ c))^* \bigcup \phi_v \rightarrow ti_4^*; l_4; \Gamma_6; \phi_6$,
  (d) $ti_1\ (t\ \alpha)^*\ l_5 \subset \Gamma_3$,
  (e) $\Gamma_6 \vdash \phi_6 \Rightarrow \phi_4$,
  (f) $l_3 = (t_v\ \alpha_v)$, where $(\vdash v' : (t_v\ \alpha_v); \phi_v)^*$
  (g) $(\alpha \notin \Gamma_3)^*$

(3) By Lemma Values-Any-Context, 1a, and 2e we have that $S; C \vdash v_0^* : ti_1^*; l_5; \Gamma_1, (t\ \alpha)^*; \phi_1, (= \alpha\ (t\ c))^* \bigcup \phi_v \rightarrow ti_3^*; l_5; \Gamma_3, (t\ \alpha)^*; \phi_3, (= \alpha\ (t\ c))^* \bigcup \phi_v$

(4) Then by Lemma Sequence-Composition, 2c, and 3, we have that $S; C \vdash v_0^*\ e'^* : ti_1^*; l_5; \Gamma_1, (t\ \alpha)^*; \phi_1, (= \alpha\ (t\ c))^* \bigcup \phi_v \rightarrow ti_2^*; l_2; \Gamma_6; \phi_6$

(5) By Lemma Strengthening and 1d, we have $S; C \vdash e_0^* : ti_4^*; l_4; \Gamma_6; \phi_6 \rightarrow ti_2^*; l_2; \Gamma_7; \phi_7$, where $\Gamma_7 \vdash \phi_7 \Rightarrow \phi_2$

(6) Finally, by Lemma Sequence-Composition, we have $S; C \vdash v_0^*\ e'^*\ e_0^* : ti_1^*; l_5; \Gamma_1, (t\ \alpha)^*; \phi_1, (= \alpha\ (t\ c))^* \bigcup \phi_v \rightarrow ti_2^*; l_2; \Gamma_7; \phi_7$

– Case: $k > 0$
$L^k[e^*] = v_k^*\ \mathsf{label}_n\{e_0^*\}\ L^{k-1}[e^*]\ \mathsf{end}\ e_k^*$.
By Lemma Inversion-On-Instruction-Typing on $S; C \vdash v_k^*\ \mathsf{label}_n\{e_0^*\}\ L^{k-1}[e^*]\ \mathsf{end}\ e_k^* : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$, we have that $S; C \vdash v^k : ti_1^*; l_1; \Gamma_1; \phi_1 \rightarrow ti_3^*; l_1; \Gamma_3; \phi_3$, $S; C \vdash \mathsf{label}_n\{e_0^*\}\ L^{k-1}[e^*]\ \mathsf{end} : ti_3^*; l_1; \Gamma_3; \phi_3 \rightarrow ti_4^*; l_4; \Gamma_4; \phi_4$, and $S; C \vdash e_k^* : ti_4^*; l_4; \Gamma_4; \phi_4 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$.
(Thought: We're introducing ti_5^*, l_5, ϕ_5 here. ¯_(ツ)_/¯)
By Lemma Inversion-On-Instruction-Typing on $S; C \vdash \mathsf{label}_n\{e_0^*\}\ L^{k-1}[e^*]\ \mathsf{end} : ti_3^*; l_1; \Gamma_3; \phi_3 \rightarrow ti_4^*; l_4; \Gamma_4; \phi_4$, we have that $S; C \vdash e_0^* : ti_5^*; l_5; \emptyset, ti_5^*, l_5; \phi_5 \rightarrow ti_4^*; l_4; \Gamma_4; \phi_4$, and $S; C, \text{label}\ (ti_5^*; l_5; \phi_5) \vdash L^{k-1}[e^*] : ti_3^*; l_1; \Gamma_3; \phi_3 \rightarrow ti_4^*; l_4; \Gamma_4; \phi_4$.
(Thought: It might seem like we can't do this, but the store typing never specifies what the labels of it's contexts are, so we can just do whatever? ¯_(ツ)_/¯)
Now, we invoke the inner inductive hypothesis (for this inductive proof) on $L^{k-1}[e^*]$ and rebuild the type using the reduced expression.
Since $S; C, \text{label}\ (ti_5^*; l_5; \phi_5) \vdash L^{k-1}[e^*] : ti_3^*; l_1; \Gamma_3; \phi_3 \rightarrow ti_4^*; l_4; \Gamma_4; \phi_4$, and $s \vdash S$, then by the inductive hypothesis on $L^{k-1}[e^*]$ we know that $s; v^*; e^* \hookrightarrow_i s'; v'^*; e'^*$, $\vdash s' : S$, $S; C, \text{label}\ (ti_5^*; l_5; \phi_5) \vdash e'^* : ti_3^*; l_6; \Gamma_3, (t_2\ \alpha_2)^*; \phi_3, (= \alpha_2\ (t_2\ c_2))^* \bigcup \phi_v^* \rightarrow ti_4^*; l_4; \Gamma_6; \phi_6$, where $ti_3^*\ (t_v\ \alpha_v)^* \subset \Gamma_6$, $\Gamma_6 \vdash \phi_6 \Rightarrow \phi_4$, $l_6 = (t_v\ \alpha_v)$, where $(\vdash v' : (t_v\ \alpha_v); \phi_v)^*$, and $(\alpha_2 \notin \Gamma_3)^*$.

$S; C \vdash v^k : ti_1^*; l_6; \Gamma_1, (t_v\ \alpha_v), (t_2\ \alpha_2)^*; \phi_1, (= \alpha_2\ (t_2\ c_2))^* \bigcup \phi_v^* \rightarrow ti_3^*; l_6; \Gamma_3, (t_v\ \alpha_v), (t_2\ \alpha_2)^*; \phi_3, (= \alpha_2\ (t_2\ c_2))^* \bigcup \phi_v^*$ by Rule Const and Lemma Threading-Constraints.

$S; C \vdash \mathsf{label}_n\{e_0^*\}\ L^{k-1}[e'^*]\ \mathsf{end} : ti_3^*; l_6; \Gamma_3, (t_v\ \alpha_v), (t_2\ \alpha_2)^*; \phi_3, (= \alpha_2\ (t_2\ c_2))^* \bigcup \phi_v^* \rightarrow ti_4^*; l_4; \Gamma_4; \phi_4$ by Rule Label.

Thus, $S; C \vdash v_k^*\ \mathsf{label}_n\{e_0^*\}\ L^{k-1}[e'^*]\ \mathsf{end}\ e_k^* : \epsilon; ti_{v'}^*; \phi_{v'}^* \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$ by Lemma Sequence-Composition.

□

C.2 Progress Lemmas and Proofs

Lemma 25. Progress If $\vdash_i s; v^*; e^* : ti^*; l; \Gamma; \phi$ then either $e^* = v'^*$, $e^* = \mathsf{trap}$, or $s; v^*; e^* \hookrightarrow_i s'; v'^*; e'^*$.

Proof. Because $\vdash_i s; v^*; e^* : ti^*; l; \Gamma; \phi$, we know that $\vdash s : S$ for some S, and that $S; \epsilon \vdash_i v^*; e^* : ti^*; l; \Gamma; \phi$ by inversion on Rule Program.

Then we know that $(\vdash v : (t_v\ \alpha_v); \phi_v)^*$ and $S; S_{\text{inst}}(i), \text{local}\ (t_v^*) \vdash e^* : \epsilon; (t_v\ \alpha_v)^*; \emptyset, (t_v\ \alpha_v)^*, (t\ \alpha)^*; \phi_v^*, (= \alpha\ (t\ c))^* \rightarrow ti^*; l; \Gamma; \phi_2$, for some a^*, t^*, c^*, where $\Gamma \vdash \phi_2 \Rightarrow \phi$ by inversion on Rule Code.

We have $s_{\text{inst}}(i)_{\text{mem}} = b^n$, where $S_{\text{inst}}(i)_{\text{memory}} = n$, and $s_{\text{inst}}(i)_{\text{tab}} = \{\text{inst}\ i, \text{func}\ \mathsf{func}\ tfi\ \ldots\}^n$, where $S_{\text{inst}}(i)_{\text{table}} = (n, tfi^n)$, as they are subderivations of $\vdash s : S$.

We decompose $e^* = (t.\mathsf{const}\ c)^*\ e_2^*$.

Then, by Lemma Progress-for-Instructions on e_2^*, we have that either

$\exists s'; v'^*; e'^*.s; v^*; e^* \hookrightarrow_i s'; v'^*; e'^*$,

or $e^* = v_2^*$,

or $e^* = \mathsf{trap}$. □

Lemma 26 (Progress-for-Instructions). If $S; S_{\text{inst}}(i) \vdash (t.\mathsf{const}\ c)^* : \epsilon; l_1; \Gamma_1; \phi_1 \rightarrow ti_2^*; l_2; \Gamma_2; \phi_2$,
and $S; S_{\text{inst}}(i) \vdash e^* : ti_2^*; l_2; \Gamma_2; \phi_2 \rightarrow ti_3^*; l_3; \Gamma_3; \phi_3$, where $e^* \neq (t.\mathsf{const}\ c_2)^*$,
and if $(t.\mathsf{const}\ c)^*\ e^* = L^k[\mathsf{br}\ i]$, then $i \leq k$,
and $s_{\text{inst}}(i)_{\text{mem}} = b^n$, where $C_{\text{memory}} = n$,
and $s_{\text{inst}}(i)_{\text{tab}} = \{\text{inst}\ i, \text{func}\ \mathsf{func}\ tfi\ \ldots\}^n$, where $C_{\text{table}} = (n, tfi^n)$,
and $(\vdash v : (t_v\ \alpha_v); \phi_v)^*$, where $C_{\text{local}} = t_v^*$,
then, either $\exists s'; v'^*; e'^*.s; v^*; (t.\mathsf{const}\ c)^*\ e^* \hookrightarrow_i s'; v'^*; e'^*$,
or $e^* = v_2^*$,
or $e^* = \mathsf{trap}$ and $(t.\mathsf{const}\ c)^* = \epsilon$

Proof. We proceed by induction on $S; C \vdash e^* : ti_2^*; l_2; \Gamma_2; \phi_2 \rightarrow ti_3^*; l_3; \Gamma_3; \phi_3$.

We only give the cases for prechecked instructions and cases that use the inductive hypothesis. While block, loop, and if all have inductive typing rules, but their proofs do not require the inductive hypothesis to be used, so they are omitted. For all the other cases, by Theorem 5, if they are well-typed in Wasm-precheck, then they must be well-typed in Wasm, so they must take a step or be irreducible. Then, since the reduction rules in Wasm and Wasm-precheck are equivalent for non-prechecked instructions, they must also take a step or be irreducible in Wasm-precheck.

- Case: $S; C \vdash \mathsf{div}✓ : ti_2^*; l_2; \Gamma_2; \phi_2 \rightarrow ti_3^*; l_3; \Gamma_3; \phi_3$
  We will show that $s; v^*; (t.\mathsf{const}\ c)^*\ t.\mathsf{div}✓ \hookrightarrow s; v^*; (t.\mathsf{const}\ c_3)$ for some $(t.\mathsf{const}\ c_3)$.
  We know $S; C \vdash (t.\mathsf{const}\ c)^* : \epsilon; l_1; \Gamma_1; \phi_1 \rightarrow (t\ \alpha_1)\ (t\ \alpha_2); l_1; \Gamma_1, (t\ \alpha_1), (t\ \alpha_2); \phi_1, (= \alpha_1\ (t\ c_1)), (= \alpha_2\ (t\ c_2))$ where $\Gamma_1, (t\ \alpha_1), (t\ \alpha_2) \vdash \phi_1, (= \alpha_1\ (t\ c_1)), (= \alpha_2\ (t\ c_2)) \Rightarrow \neg(=\ a_2\ 0)$ by Lemma Inversion-On-Instruction-Typing on $S; C \vdash \mathsf{div}✓ : ti_2^*; l_2; \Gamma_2; \phi_2 \rightarrow ti_3^*; l_3; \Gamma_3; \phi_3$.
  Then, since $S; C \vdash (t.\mathsf{const}\ c)^* : \epsilon; l_1; \Gamma_1; \phi_1 \rightarrow (t\ \alpha_1)\ (t\ \alpha_2); l_1; \Gamma_1, (t\ \alpha_1), (t\ \alpha_2); \phi_1, (= \alpha_1\ (t\ c_1)), (= \alpha_2\ (t\ c_2))$, we have $(t.\mathsf{const}\ c)^* = (t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)$.

Since we have $(= \alpha_2\ (t\ c_2))$ on the left hand side (and since a_2 is fresh that can be on the only constraint on a_2, and that implies that $a_2 = 0$, then it must be the case that $c_2 \neq 0$.
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, $s; v^*; (t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ t.\mathsf{div}\checkmark \hookrightarrow_i s; v^*; (t.\mathsf{const}\ c_3)$.

- Case: $S; C \vdash (t.\mathsf{load}\checkmark\ (tp\_sx)\ align\ o) : ti_2^*; l_2; \Gamma_2; \phi_2 \longrightarrow ti_3^*; l_3; \Gamma_3; \phi_3$
We will show that $s; (t.\mathsf{const}\ c)^*\ (t.\mathsf{load}\checkmark\ (tp\_sx)\ align\ o) \hookrightarrow t.\mathsf{const}\ c$ for some c.
We know $S; C \vdash (t.\mathsf{const}\ c)^* : \epsilon; l_1; \Gamma_1; \phi_1 \rightarrow (\mathsf{i32}\ \alpha); l_2; \Gamma_1, (\mathsf{i32}\ \alpha); \phi_1, (= \alpha\ (\mathsf{i32}\ k))$ where
$$\begin{aligned}\phi_1, (= \alpha\ (\mathsf{i32}\ k)) \implies &(\mathsf{ge}\ (\mathsf{add}a\ (\mathsf{i32}\ o))(\mathsf{i32}\ 0)),\\ &(\mathsf{le}\ (\mathsf{add}a\ (\mathsf{add}(\mathsf{i32}\ o + width)))\\ &\quad(\mathsf{i32}\ n_2 * 64\mathrm{Ki}))\end{aligned}$$
where $width = |t|$ if $tp^? = \epsilon$, and $|tp|$ otherwise, and $n_2 * 64\mathrm{Ki} = S_{\mathrm{mem}}(i, j)$ by Lemma Inversion-On-Instruction-Typing on $S; C \vdash (t.\mathsf{load}\checkmark\ (tp\_sx)\ align\ o) : ti_2^*; l_2; \Gamma_2; \phi_2 \longrightarrow ti_3^*; l_3; \Gamma_3; \phi_3$.
Then, since $S; C \vdash (t.\mathsf{const}\ c)^* : \epsilon; l_1; \Gamma_1; \phi_1 \rightarrow (\mathsf{i32}\ \alpha); l_2; \Gamma_1, (\mathsf{i32}\ \alpha); \phi_1, (= \alpha\ (\mathsf{i32}\ k))$, we know that $(t.\mathsf{const}\ c)^* = \mathsf{i32.const}\ k$.
Then, it must be the case that $k + o \geq 0$ and $k + o + width \leq n_2 * 64\mathrm{Ki}$.
Since $n_2 * 64\mathrm{Ki} = C_{\mathrm{memory}}$, we have $s_{\mathrm{inst}}(i)_{\mathrm{mem}}(i, j) = b_2^*$ where $C_memory = n_2 * 64\mathrm{Ki} = |b_2^*|$.
Therefore, it must be the case that $k+o \geq 0$ and $k+o+width < |b_2^*|$, and therefore $s_{\mathrm{mem}}(i, k + o, width) = b_3^*$ for some b_3^* that is a subsequence of b_2^*. Then, $s; v^*; (\mathsf{i32.const}\ k)\ (t.\mathsf{load}\checkmark\ (tp\_sx)\ align\ o)$
$s; v^*; t.\mathsf{const}\ \mathrm{const}_t^{sx}(b_3^*)$.
- Case: $S; C \vdash (t.\mathsf{store}\checkmark\ tp\ align\ o) : ti_2^*; l_2; \Gamma_2; \phi_2 \longrightarrow ti_3^*; l_3; \Gamma_3; \phi_3$
We will show that $s; v^*; (t.\mathsf{const}\ c)^*\ (t.\mathsf{store}\checkmark\ (tp\_sx)\ align\ o) \hookrightarrow s'; v^*; t.\mathsf{const}\ c$ for some s' and $t.\mathsf{const}\ c$.
We know $S; C \vdash (t.\mathsf{const}\ c)^* : \epsilon; l_1; \Gamma_1; \phi_1 \rightarrow (\mathsf{i32}\ \alpha); l_2; \Gamma_1, (\mathsf{i32}\ \alpha); \phi_1, (= \alpha\ (\mathsf{i32}\ k))$ where
$$\begin{aligned}\phi_1, (= \alpha\ (\mathsf{i32}\ k)) \implies &(\mathsf{ge}\ (\mathsf{add}a\ (\mathsf{i32}\ o))(\mathsf{i32}\ 0)),\\ &(\mathsf{le}\ (\mathsf{add}a\ (\mathsf{add}(\mathsf{i32}\ o + width)))\\ &\quad(\mathsf{i32}\ n_2 * 64\mathrm{Ki}))\end{aligned}$$
where $width = |t|$ if $tp^? = \epsilon$, and $|tp|$ otherwise, and $n_2 * 64\mathrm{Ki} = S_{\mathrm{mem}}(i, j)$ by Lemma Inversion-On-Instruction-Typing on $S; C \vdash (t.\mathsf{store}\checkmark\ (tp\_sx)\ align\ o) : ti_2^*; l_2; \Gamma_2; \phi_2 \longrightarrow ti_3^*; l_3; \Gamma_3; \phi_3$.
Then, since $S; C \vdash (t.\mathsf{const}\ c)^* : \epsilon; l_1; \Gamma_1; \phi_1 \rightarrow (\mathsf{i32}\ \alpha); l_2; \Gamma_1, (\mathsf{i32}\ \alpha); \phi_1, (= \alpha\ (\mathsf{i32}\ k))$, we know that $(t.\mathsf{const}\ c)^* = \mathsf{i32.const}\ k$.
Then, it must be the case that $k + o \geq 0$ and $k + o + width \leq n_2 * 64\mathrm{Ki}$.
Since $n_2 * 64\mathrm{Ki} = C_{\mathrm{memory}}$, we have $s_{\mathrm{inst}}(i)_{\mathrm{mem}}(i, j) = b_2^*$ where $C_memory = n_2 * 64\mathrm{Ki} = |b_2^*|$.
Therefore, it must be the case that $k + o \geq 0$ and $k + o + width < |b_2^*|$, and therefore $s_{\mathrm{mem}}(i, k + o, width) = b_3^*$ for some b_3^* that is a subsequence of b_2^*.
Then, we can construct $s' = s$ with $s'_{\mathrm{mem}}(i, k+o, width) = bits_t^{width}(c)$ because $|bits_t^{width}(c) \models |b_3^*|$.
Then,
$$s; v^*; (\mathsf{i32.const}\ k)\ (\mathsf{i32.const}\ c)\ (t.\mathsf{store}\checkmark\ tp\ align\ o) \hookrightarrow_i s'; v^*; \epsilon$$
- Case: $S; C \vdash \mathsf{call\_indirect}\ (ti_4^*; \phi_4 \longrightarrow ti_5^*; \phi_5) : ti_2^*; l_2; \Gamma_2; \phi_2 \longrightarrow ti_3^*; l_3; \Gamma_3; \phi_3$
We will show that $s; v^*; (t.\mathsf{const}\ c)^*\ (\mathsf{call\_indirect}\ (ti_4^*; \phi_4 \longrightarrow ti_5^*; \phi_5)) \hookrightarrow s; v^*; \mathsf{call}\ cl$ for some cl.

We know $S; C \vdash (t.\text{const } c)^* : \epsilon; l_1; \Gamma_1; \phi_1 \longrightarrow (\text{i32 } \alpha); l_2; \Gamma_1, (\text{i32 } \alpha); \phi_1, (= \alpha\ (\text{i32 } k))$, where $\Gamma_1; (\text{i32 } \alpha) \vdash \phi_1, (= \alpha\ (\text{i32 } k)) \Rightarrow (\text{le } \ a\ n) \wedge (\text{gt}(\text{i32 } 0)\ a), C_{\text{table}} = (n, tfi^n)$, and $\forall 0 < j \leq n.\ (\Gamma_1; (\text{i32 } \alpha) \vdash \phi_1, (= \alpha\ (\text{i32 } k)) \Rightarrow \neg(= (\text{i32 } j)\ a)) \vee\ (tfi^*)(i) = (ti_4^*; \phi_4 \longrightarrow ti_5^*; \phi_5)$ as they are premises of Rule CALLINDIRECT, which we have assumed to hold.
Then, since $S; C \vdash (t.\text{const } c)^* : \epsilon; l_1; \Gamma_1; \phi_1 \longrightarrow (\text{i32 } \alpha); l_2; \Gamma_1, (\text{i32 } \alpha); \phi_1, (= \alpha\ (\text{i32 } k))$, we know that $(t.\text{const } c)^* = \text{i32.const } k$.
Then it has to be the case that $0 \leq k < n$, and that $C_{\text{table}}(k) = ti_4^*; \phi_4 \longrightarrow ti_5^*; \phi_5$.
Therefore $s_{\text{inst}}(i)_{\text{tab}}(k) = \{\text{inst } j,\ \text{func func } (ti_4^*; \phi_4 \longrightarrow ti_5^*; \phi_5) \text{ local } t^*\ e^*\}$, because it is an assumption of the proof.
Thus, $s; (\text{i32.const } c)\ \text{call\_indirect } \epsilon; l_1; \Gamma_1; \phi_1 \longrightarrow ti_3^*; l_3; \Gamma_3; \phi_3$.
$\hookrightarrow s; v^*; \text{call } \{\text{inst } j,\ \text{func func } (ti_4^*; \epsilon; \Gamma_4; \phi_4 \longrightarrow ti_5^*; \epsilon; \Gamma_5; \phi_5) \text{ local } t^*\ e^*\}$

- Case: $S; C \vdash e_1^*\ e_2 : ti_2^*; l_2; \Gamma_2; \phi_2 \longrightarrow ti_3^*; l_3; \Gamma_3; \phi_3$
  We have $S; C \vdash e_1^* : ti_2^*; l_2; \Gamma_2; \phi_2 \longrightarrow ti_4^*; l_4; \Gamma_4; \phi_4$ and $S; C \vdash e_2 : ti_4^*; l_4; \Gamma_4; \phi_4 \longrightarrow ti_3^*; l_3; \Gamma_3; \phi_3$ as they are sub-derivations of $S; C \vdash e_1^*\ e_2 : ti_2^*; l_2; \Gamma_2; \phi_2 \longrightarrow ti_3^*; l_3; \Gamma_3; \phi_3$.
  We proceed on cases by the inductive hypothesis on e^*:
  - Case: $s; v^*; (t.\text{const } c)^*\ e_1^* \hookrightarrow_i s'; v'^*; e'^*$.
    Then we have $s; v^*; L^0[(t.\text{const } c)^*\ e_1^*] \hookrightarrow_i s'; v'^*; L^0[e'^*]$, where $L^0 = \epsilon\ []\ e_2$.
  - Case: $e_1^* = \text{trap}$ and $(t.\text{const } c)^* = \epsilon$
    Then, $e_1^*\ e_2 = L^0[\text{trap}]$, and therefore $s; v^*; e_1^*\ e_2 \hookrightarrow_i s; v^*; \text{trap}$.
  - Case: $e_1 = (t_2.\text{const } c_2)^*$
    We proceed by cases on the inductive hypothesis on e_2:
    * Case: $s; v^*; (t.\text{const } c)^*\ (t_2.\text{const } c_2)^*\ e_2 \hookrightarrow_i s'; v'^*; e'^*$.
    * Case:$e_2 = \text{trap}$ and $(t.\text{const } c)^*\ (t_2.\text{const } c_2)^* = \epsilon$
      Then we have $e_1^*\ e_2 = \text{trap}$
    * Case: $e_2 = (t_3.\text{const } c_3)$
      We have $e_1^*\ e_2 = (t_2.\text{const } c_2)^*\ t_3.\text{const } c_3)$.
- Case: $S; C \vdash e^* : ti^*\ ti_2^*; l_2; \Gamma_2; \phi_2 \longrightarrow ti^*\ ti_3^*; l_3; \Gamma_3; \phi_3$
  We know $S; C \vdash (t.\text{const } c)^* : \epsilon; l_1; \Gamma_1; \phi_1 \longrightarrow ti^*\ ti_2^*; l_2; \Gamma_1, ti^*\ ti_2^*; \phi_2$, where $\Gamma_1 = \Gamma_1, ti^*\ ti_2^*$, by Lemma INVERSION-ON-INSTRUCTION-TYPING on $S; C \vdash e^* : ti^*\ ti_2^*; l_2; \Gamma_2; \phi_2 \longrightarrow ti^*\ ti_3^*; l_3; \Gamma_3; \phi_3$.
  Then, by Lemma INVERSION-ON-INSTRUCTION-TYPING on $S; C \vdash (t.\text{const } c)^* : \epsilon; l_1; \Gamma_1; \phi_1 \longrightarrow ti^*\ ti_2^*; l_2; \Gamma_1, ti^*\ ti_2^*; \phi_2$, we know that $(t.\text{const } c)^* = (t_1.\text{const } c_1)^*\ (t.\text{const } c_2)^*$, where $S; C \vdash (t_1.\text{const } c_1)^* : \epsilon; l_1; \Gamma_1; \phi_1 \longrightarrow ti^*; l_2; \Gamma_1, ti^*; \phi_4$, and $S; C \vdash (t_2.\text{const } c_2)^* : ti^*; l_2; \Gamma_1, ti^*; \phi_4 \longrightarrow ti^*\ ti_2^*; l_2; \Gamma_1, ti^*\ ti_2^*; \phi_2$.
  Then, $S; C \vdash (t_2.\text{const } c_2)^* : \epsilon; l_2; \Gamma_1, ti^*; \phi_4 \longrightarrow ti_2^*; l_2; \Gamma_1, ti_2^*; \phi_2$
  We have $S; C \vdash e^* : ti_2^*; l_2; \Gamma_2; \phi_2 \longrightarrow ti_3^*; l_3; \Gamma_3; \phi_3$ as a subderivation of $S; C \vdash e^* : ti^*\ ti_2^*; l_2; \Gamma_2; \phi_2 \longrightarrow ti^*\ ti_3^*; l_3; \Gamma_3; \phi_3$.
  Then, we proceed by cases on the inductive hypothesis on e^*:
  - Case: $s; v^*; (t_2.\text{const } c_2)^*\ e^* \hookrightarrow_i s'; v'^*; e'^*$
    Then, we have $s; v^*; L^0[(t_2.\text{const } c_2)^*\ e^*] \hookrightarrow_i s'; v'^*; L^0[e'^*]$, where $L^0 = (t_1.\text{const } c_1) * []\epsilon$.
    Thus $s; v^*; (t_1.\text{const } c_1)^*\ (t_2.\text{const } c_2)^*\ e^* \hookrightarrow_i s'; v'^*; (t_1.\text{const } c_1)^*\ e'^*$
  - $e^* = (t_3.\text{const } c_3)^*$
  - $e^* = \text{trap}$ and $(t_2.\text{const } c_2)^* = \epsilon$
    Then, $(t_1.\text{const } c_1)^*\ e^* = L^0[\text{trap}]$, and therefore $s; v^*; (t_1.\text{const } c_1)^*\ e^* \hookrightarrow_i s; v^*; \text{trap}$.
- Case: $S; C \vdash \text{label}_n\{e_0^*\}\ e^*\ \text{end} : \epsilon; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$
  We know $S; C, \text{label}(ti_3^*; l_3; \Gamma_3; \phi_3) \vdash e^* : \epsilon; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$, and $S; C \vdash e_0^* : ti_3^*; l_3; \Gamma_3; \phi_3 \longrightarrow ti_2^n; l_2; \Gamma_2; \phi_2$, as they are subderivations of $S; C \vdash \text{label}_n\{e_0^*\}\ e^*\ \text{end} : \epsilon; l_1; \Gamma_1; \phi_1 \longrightarrow ti_2^*; l_2; \Gamma_2; \phi_2$.

If $e^* = L^k[\text{br } k]$, then we reuse the Wasm proof [Watt 2018] using Theorem 5 and Lemma Erasure-Same-Semantics.
Otherwise, we decompose $e^* = (t_2.\text{const } c_2)^*\ e_2^*$ and then we can invoke the inductive hypothesis on e_2^*. We proceed by cases:

- Case: $s; v^*; (t_2.\text{const } c_2)^*\ e_2^* \hookrightarrow_i s'; v'^*; e'^*$
  Let $(t.\text{const } c)^*\ \text{label}_n\{e_0^*\}\ (t_2.\text{const } c_2)^*\ e_2^*\ \text{end} = L^{k+1}[(t_2.\text{const } c_2)^*\ e_2^*]$, where $L^k = (t_2.\text{const } c_2)^*\ e_2^*$.
  Then, $s; v^*; L^{k+1}[e^*] \hookrightarrow_i s'; v'^*; L^{k+1}[e'^*]$, since $s; v^*; (t_2.\text{const } c_2)^*\ e_2^* \hookrightarrow_i s'; v'^*; e'^*$.
  Thus, $s; v^*; (t.\text{const } c)^*\ \text{label}_n\{e_0^*\}\ e^*\ \text{end} \hookrightarrow_i s'; v'^*; (t.\text{const } c)^*\ \text{label}_n\{e_0^*\}\ e'^*\ \text{end}$
- Case: $e_2^* = (t_3.\text{const } c_3)^*$
  Then, $\text{label}_n\{e_0^*\}\ (t_2.\text{const } c_2)^*\ e_2^*\ \text{end} = \text{label}_n\{e_0^*\}\ (t_2.\text{const } c_2)^*\ (t_3.\text{const } c_3)^*\ \text{end}$.
  Then, $s; v^*; \text{label}_n\{e_0^*\}\ e^*\ \text{end} \hookrightarrow_i s; v^*; (t_2.\text{const } c_2)^*\ (t_3.\text{const } c_3)^*$.
  We have $s; v^*; L^0[\text{label}_n\{e_0^*\}\ e^*\ \text{end}] \hookrightarrow_i s'; v'^*; L^0[(t_2.\text{const } c_2)^*\ (t_3.\text{const } c_3)^*]$, where $L^0 = (t.\text{const } c)*\ []\epsilon$.
  Thus, $s; v^*; (t.\text{const } c)*\ \text{label}_n\{e_0^*\}\ e^*\ \text{end} \hookrightarrow_i s'; v'^*; (t.\text{const } c)*\ (t_2.\text{const } c_2)^*\ (t_3.\text{const } c_3)^*$
- Case: $e_2^* = \text{trap}$ and $(t_2.\text{const } c_2)^* = \epsilon$
  Then, $\text{label}_n\{e_0^*\}\ (t_2.\text{const } c_2)^*\ e_2^*\ \text{end} = \text{label}_n\{e_0^*\}\ \text{trap end}$
  Then, $s; v^*; \text{label}_n\{e_0^*\}\ \text{trap end} \hookrightarrow_i s; v^*; \text{trap}$.
  We have $s; v^*; L^0[\text{label}_n\{e_0^*\}\ \text{trap end}] \hookrightarrow_i s'; v'^*; L^0[\text{trap}]$, where $L^0 = (t.\text{const } c)*\ []\epsilon$.
  Thus, $s; v^*; (t.\text{const } c)*\ \text{label}_n\{e_0^*\}\ \text{trap end} \hookrightarrow_i s'; v'^*; (t.\text{const } c)*\ \text{trap}$

• Case: $S; C \vdash \text{local}_n\{j; v_l^*\}\ e^*\ \text{end} : \epsilon; l_1; \Gamma_1; \phi_1 \to ti_2^*; l_2; \Gamma_2; \phi_2$
We know $S; C, \text{local}(t_l^*), \text{return}(ti_3^*; l_3; \Gamma_3; \phi_3) \vdash e^* : \epsilon; (t_l\ \alpha_l)^*; (t_l\ \alpha_l)^*; \bigcup \phi_l^* \to ti_2^*; l_2; \Gamma_2; \phi_2$, where $v_l^* = (t_l.\text{const } c_l)^*$, and $(\vdash v_l : (t_l\ \alpha_l); \phi_l)^*$, as they are subderivations of $S; C \vdash \text{local}_n\{j; v_l^*\}\ e^*\ \text{end} : \epsilon; l_1; \Gamma_1; \phi_1 \to ti_2^*; l_2; \Gamma_2; \phi_2$.
If $e^* = L^k[\text{return}k]$, then we reuse the Wasm proof [Watt 2018] using Theorem 5 and Lemma Erasure-Same-Semantics.
Otherwise, we decompose $e^* = (t_2.\text{const } c_2)^*\ e_2^*$ and then we can invoke the inductive hypothesis on e_2^*. We proceed by cases:
- Case: $s; v_l^*; (t_2.\text{const } c_2)^*\ e_2^* \hookrightarrow_j s'; v_l'^*; e'^*$
  Then, $s; v^*; \text{local}_n\{j; v_l\}\ e^*\ \text{end} \hookrightarrow_i s'; v^*; \text{local}_n\{j; v_l'^*\}\ e^*\ \text{end}$, since $s; v^*; (t_2.\text{const } c_2)^*\ e_2^* \hookrightarrow_j s'; v'^*; e'^*$.
  We have $s; v^*; L^0[\text{local}_n\{j; v_l^*\}\ e^*\ \text{end}] \hookrightarrow_i s'; v^*; L^0[\text{local}_n\{j; v_l'^*\}\ e^*\ \text{end}]$, where $L^0 = (t.\text{const } c)*\ []\epsilon$.
  Thus, $s; v^*; (t.\text{const } c)^*\ \text{local}_n\{j; v_l^*\}\ e^*\ \text{end} \hookrightarrow_i s'; v^*; (t.\text{const } c)^*\ \text{local}_n\{j; v_l'^*\}\ e'^*\ \text{end}$
- Case: $e_2^* = (t_3.\text{const } c_3)^*$
  Then, $\text{local}_n\{j; v_l^*\}\ (t_2.\text{const } c_2)^*\ e_2^*\ \text{end} = \text{local}_n\{j; v_l^*\}\ (t_2.\text{const } c_2)^*\ (t_3.\text{const } c_3)^*\ \text{end}$.
  Then, $s; v^*; \text{local}_n\{j; v_l^*\}\ e^*\ \text{end} \hookrightarrow_i s; v^*; (t_2.\text{const } c_2)^*\ (t_3.\text{const } c_3)^*$.
  We have $s; v^*; L^0[\text{local}_n\{j; v_l^*\}\ e^*\ \text{end}] \hookrightarrow_i s'; v'^*; L^0[(t_2.\text{const } c_2)^*\ (t_3.\text{const } c_3)^*]$, where $L^0 = (t.\text{const } c)*\ []\epsilon$.
  Thus, $s; v^*; (t.\text{const } c)*\ \text{local}_n\{j; v_l^*\}\ e^*\ \text{end} \hookrightarrow_i s'; v'^*; (t.\text{const } c)*\ (t_2.\text{const } c_2)^*\ (t_3.\text{const } c_3)^*$
- Case: $e_2^* = \text{trap}$ and $(t_2.\text{const } c_2)^* = \epsilon$
  Then, $\text{local}_n\{j; v_l^*\}\ (t_2.\text{const } c_2)^*\ e_2^*\ \text{end} = \text{local}_n\{j; v_l^*\}\ \text{trap end}$
  Then, $s; v^*; \text{local}_n\{j; v_l^*\}\ \text{trap end} \hookrightarrow_i s; v^*; \text{trap}$.
  We have $s; v^*; L^0[\text{local}_n\{j; v_l^*\}\ \text{trap end}] \hookrightarrow_i s'; v'^*; L^0[\text{trap}]$.
  Thus, $s; v^*; (t.\text{const } c)*\ \text{local}_n\{j; v_l^*\ \text{trap end} \hookrightarrow_i s'; v'^*; (t.\text{const } c)*\ \text{trap}$

• Otherwise, we reuse the Wasm proof [Watt 2018] using Theorem 5 and Lemma Erasure-Same-Semantics.

Table 2. Comparison of the binary sizes of hand-annotated Wasm-precheck programs vs Wasm versions. All numbers are in bytes.

	Wasm		Wasm-precheck	
Benchmark	Code	Type	Code	Type
correlation	20376	204	20525	1852
covariance	20178	203	20297	1259
2mm	20394	218	20633	2707
3mm	20513	206	20825	3761
atax	19995	193	20154	1126
bicg	20107	193	20330	1195
doitgen	20165	205	20293	1848
mvt	20204	193	20459	1562
gemm	20170	204	20344	1737
gemver	20400	227	20847	3496
gesummv	20071	206	20300	1091
symm	20251	204	20419	1519
syr2k	20171	204	20336	1502
syrk	20080	203	20194	1100
trmm	20040	202	20161	923
cholesky	20327	193	20440	1563
durbin	19954	193	20063	729
gramschmidt	20367	193	20541	1438
lu	20309	193	20419	1480
ludcmp	20630	193	20901	2632
trisolv	19920	193	20092	858
deriche	21106	237	21262	1653
floyd-warshall	16582	176	16645	420
nussinov	16749	176	16843	759
adi	20532	193	20693	1807
fdtd-2d	20609	193	20827	2145
heat-3d	20446	193	20570	1008
jacobi-1d	19890	193	20002	566
jacobi-2d	20139	193	20259	704
seidel-2d	20012	193	20079	529

□

D EXTRA EXPERIMENT DETAILS

D.1 Full Data for Size of Binaries

We present the full data for the comparison between the binary sizes of hand-annotated Wasm-precheck programs and their unmodified Wasm counterparts in Table 2. Unlike in the main body of the paper, the sizes here are broken down into code and type sections.

D.2 Full System Details

System and Hardware details. The benchmarks are performed on a cloud compute instance, running on OpenStack Ussuri. The instance runs Ubuntu-22.04.2-Jammy-x64-2023-02.

The instance is allocated 16 vCPUs, from a pool of Intel® Xeon® Processor E5-2680 v4 physical CPUs. The instance has 120GiB of ECC RAM, of unknown speed.

Environment Variables. We include the full environment variables of the machine on which the benchmarks were run in Figure 11, as these have been shown to impact cache behaviour and thus performance measurements [Curtsinger and Berger 2013].

Fig. 11. Fully reproduced environment variables of the machine we used to run the experiments.

```
SHELL=/bin/bash
PWD=/home/ubuntu
LOGNAME=ubuntu
XDG_SESSION_TYPE=tty
MOTD_SHOWN=pam
HOME=/home/ubuntu
LANG=C.UTF-8
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd
    ↪ =40;33;01:cd=40;33;01:or=40;31;01:mi=00:su=37;41:sg=30;43:ca=30;41:
    ↪ tw=30;42:ow=34;42:st=37;44:ex=01;32:*.tar=01;31:*.tgz=01;31:*.arc
    ↪ =01;31:*.arj=01;31:*.taz=01;31:*.lha=01;31:*.lz4=01;31:*.lzh
    ↪ =01;31:*.lzma=01;31:*.tlz=01;31:*.txz=01;31:*.tzo=01;31:*.t7z
    ↪ =01;31:*.zip=01;31:*.z=01;31:*.dz=01;31:*.gz=01;31:*.lrz=01;31:*.lz
    ↪ =01;31:*.lzo=01;31:*.xz=01;31:*.zst=01;31:*.tzst=01;31:*.bz2
    ↪ =01;31:*.bz=01;31:*.tbz=01;31:*.tbz2=01;31:*.tz=01;31:*.deb=01;31:*.
    ↪ rpm=01;31:*.jar=01;31:*.war=01;31:*.ear=01;31:*.sar=01;31:*.rar
    ↪ =01;31:*.alz=01;31:*.ace=01;31:*.zoo=01;31:*.cpio=01;31:*.7z
    ↪ =01;31:*.rz=01;31:*.cab=01;31:*.wim=01;31:*.swm=01;31:*.dwm=01;31:*.
    ↪ esd=01;31:*.jpg=01;35:*.jpeg=01;35:*.mjpg=01;35:*.mjpeg=01;35:*.gif
    ↪ =01;35:*.bmp=01;35:*.pbm=01;35:*.pgm=01;35:*.ppm=01;35:*.tga
    ↪ =01;35:*.xbm=01;35:*.xpm=01;35:*.tif=01;35:*.tiff=01;35:*.png
    ↪ =01;35:*.svg=01;35:*.svgz=01;35:*.mng=01;35:*.pcx=01;35:*.mov
    ↪ =01;35:*.mpg=01;35:*.mpeg=01;35:*.m2v=01;35:*.mkv=01;35:*.webm
    ↪ =01;35:*.webp=01;35:*.ogm=01;35:*.mp4=01;35:*.m4v=01;35:*.mp4v
    ↪ =01;35:*.vob=01;35:*.qt=01;35:*.nuv=01;35:*.wmv=01;35:*.asf=01;35:*.
    ↪ rm=01;35:*.rmvb=01;35:*.flc=01;35:*.avi=01;35:*.fli=01;35:*.flv
    ↪ =01;35:*.gl=01;35:*.dl=01;35:*.xcf=01;35:*.xwd=01;35:*.yuv=01;35:*.
    ↪ cgm=01;35:*.emf=01;35:*.ogv=01;35:*.ogx=01;35:*.aac=00;36:*.au
    ↪ =00;36:*.flac=00;36:*.m4a=00;36:*.mid=00;36:*.midi=00;36:*.mka
    ↪ =00;36:*.mp3=00;36:*.mpc=00;36:*.ogg=00;36:*.ra=00;36:*.wav=00;36:*.
    ↪ oga=00;36:*.opus=00;36:*.spx=00;36:*.xspf=00;36:
SSH_CONNECTION=154.16.162.143 62340 192.168.0.6 22
LESSCLOSE=/usr/bin/lesspipe %s %s
XDG_SESSION_CLASS=user
TERM=xterm-256color
LESSOPEN=| /usr/bin/lesspipe %s
USER=ubuntu
SHLVL=1
XDG_SESSION_ID=2169
XDG_RUNTIME_DIR=/run/user/1000
SSH_CLIENT=154.16.162.143 62340 22
XDG_DATA_DIRS=/usr/local/share:/usr/share:/var/lib/snapd/desktop
PATH=/home/ubuntu/.local/bin:/home/ubuntu/.cargo/bin:/usr/local/sbin:/usr/
    ↪ local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games
    ↪ :/snap/bin
DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/1000/bus
SSH_TTY=/dev/pts/0
_=/usr/bin/printenv
```