Indexed Types for a Statically Safe WebAssembly

ADAM T. GELLER, University of British Columbia, Canada
JUSTIN 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 from 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 configurations with explicit dynamic checks, and find an average run-time performance gain of 1.71x 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.

CCS Concepts: • **Theory of computation → Program analysis**; **Type structures**; • **Software and its engineering → Formal software verification**; *Software performance.*

Additional Key Words and Phrases: WebAssembly, Indexed Types, Program Logics, Optimization and Compiler Design, Type Systems

ACM Reference Format:
Adam T. Geller, Justin Frank, and William J. Bowman. 2024. Indexed Types for a Statically Safe WebAssembly. *Proc. ACM Program. Lang.* 8, POPL, Article 80 (January 2024), 31 pages. https://doi.org/10.1145/3632922

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 explicit dynamic memory bounds checks cause an average of 1.76x slowdown. This demonstrates the significance of finding a strategy to safely avoid performing these explicit dynamic checks.

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

2475-1421/2024/1-ART80
https://doi.org/10.1145/3632922

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].

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 1.71x over Wasm_dyn, a configuration of Wasm with explict dynamic checks. This speed-up comes with 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

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

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).
- 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 comparing Wasm-precheck to various configurations of Wasm (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 (Section 4.1).
- 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.

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

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 $t.binop$ 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 $binop(c_1, c_2)$ 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.

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

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 of 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.

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

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.

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

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.

$$
\begin{array}{l}
\quad testop ::= \mathsf{eqz} \qquad binop ::= \mathsf{add} \mid \mathsf{sub} \mid \mathsf{shl} \mid \mathsf{or} \mid \ldots \qquad relop ::= \mathsf{eq} \mid \mathsf{ne} \mid \mathsf{gt} \mid \mathsf{ge} \mid \ldots \\
e ::= \mathsf{unreachable} \mid \mathsf{nop} \mid \mathsf{drop} \mid \mathsf{select} \mid \mathsf{block}\ t_1^* \to t_2^*\ e^*\ \mathsf{end} \mid \mathsf{loop}\ t_1^* \to t_2^*\ e^*\ \mathsf{end} \\
\quad \mid \mathsf{if}\ t_1^* \to t_2^*\ e^*\ \mathsf{else}\ e^*\ \mathsf{end} \mid \mathsf{br}\ i \mid \mathsf{br\_if}\ i \mid \mathsf{br\_table}\ i^+ \mid \mathsf{return} \mid \mathsf{call}\ i \mid \mathsf{call\_indirect}\ ti_1^*;\ \phi_1 \to ti_2^*;\ \phi_2 \\
\quad \mid \mathsf{get\_local}\ i \mid \mathsf{set\_local}\ i \mid \mathsf{tee\_local}\ i \mid \mathsf{get\_global}\ i \mid \mathsf{set\_global}\ i \mid \mathsf{current\_memory} \mid \mathsf{grow\_memory} \\
\quad \mid t.\mathsf{const}\ c \mid t.testop \mid t.relop \mid t.binop \mid t.\mathsf{load}\ (tp\_sx)^?\ o \mid t.\mathsf{store}\ tp^?\ o \\
\quad \mid t.\mathsf{div}\checkmark \mid t.\mathsf{call\_indirect}\checkmark\ ti_1^*;\ \phi_1 \to ti_2^*;\ \phi_2 \mid t.\mathsf{load}\checkmark\ (tp\_sx)^?\ o \mid t.\mathsf{store}\checkmark\ tp^?\ o
\end{array}
$$

Fig. 1. Wasm-precheck instruction syntax

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 (Section 4.1). 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 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 can improve performance by safely removing dynamic checks 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. Our evaluation shows that, using Wasm-precheck, we can remove 97% of the performance overhead of explicit dynamic checks on average, resulting in an average performance speed-up of 1.71x. 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 a 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. Four administrative instructions, which can only appear during evaluation, are omitted here, as we do not discuss them in detail. 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

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

$$\begin{array}{rll}
(t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ t.\mathit{binop} \hookrightarrow & t.\mathsf{const}\ c & \text{if } c = \mathit{binop}(c_1, c_2) \\
(t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ t.\mathit{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}\ \mathit{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 } \mathrm{mem}(i, k + o, |t|) = \mathit{bits}_t(c) \\
s;\ (\mathsf{i32.const}\ k)\ (t.\mathsf{const}\ c)\ (t.\mathsf{store}\ o) \hookrightarrow_i & \mathsf{trap} & \text{otherwise} \\
\hline
\multicolumn{3}{c}{\textit{New rules}} \\
(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_{\text{tab}}(i, j) & \\
 & \multicolumn{2}{l}{\text{where } s_{\text{tab}}(i, j) = (\mathsf{func}\ (ti_1^*; \phi_1 \to ti_2^*; \phi_2)\ \ldots)} \\
s;\ (\mathsf{i32.const}\ k)\ (t.\mathsf{load}\checkmark\ o) \hookrightarrow_i & t.\mathsf{const}\ \mathit{const}_t(b^*) & \text{where } s_{\mathit{mem}}(i, k + o, |t|) = b^* \\
s;\ (\mathsf{i32.const}\ k)\ (t.\mathsf{const}\ c)\ (t.\mathsf{store}\checkmark\ o) \hookrightarrow_i & s'; \epsilon & \text{where } s' = s \text{ with } \mathrm{mem}(i, k + o, |t|) = \mathit{bits}_t^{|t|}(c)
\end{array}$$

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

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 evaluation rules, 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.const c). As in Wasm, the stack is represented as a sequence of values at the head of the instruction sequence e^*. Following Wasm, the store s, local variables v^*, and the instance subscript i are elided when they are unchanged and unused (hence, s and v^* do not appear in Figure 3).

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

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

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

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

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 second-to-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.

Function calls, both direct and indirect, must first determine which closure represents the function being called, and then the body of the closure will be evaluated in an environment specified by the closure. A direct function call, call , uses a statically provided function index to get the closure from the list of functions in the current module. An indirect function call, call_indirect , first dynamically looks up the function index; this process is explained more below.

Closures in Wasm are a combination of a function and a pointer to the module environment that the function should operate within. The module environment contains all of the global variables, table, memory, and functions that can be referred to within the closure. Evaluating a closure introduce a return label in the form of a local administrative block instruction, which also holes the local variables for the function, and the module environment pointer i. The local variables in the local block represent arguments consumed by the function call (v^n), and a number of additional local variables specified as part of the function, which are initialized to zero ($(t.\mathsf{const}\ \ 0)^k$).

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

$$l ::= ti^* \qquad l ::= ti^* \qquad \phi ::= \emptyset \mid \phi, P \qquad \Gamma ::= \emptyset \mid \Gamma, (t\ \alpha) \qquad \mathit{tfi} ::= ti^*; \phi \rightarrow ti^*; \phi$$

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

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

Fig. 4. Wasm-precheck Typing and Index Language Syntax

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 alignment does not affect the semantics in any way, so we omit this from the formal model). 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 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; it is represented as a set of indexed types. 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 available as part of the supplementary material [Geller et al. 2023].

Constraints in an instruction type are written in the *index language*, given in Figure 4:

- P is a *constraint* about index terms: either an equality constraint, or a proposition combining constraints using a simple first-order logic;
- r is an *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: $\Gamma \vdash \phi_1 \Rightarrow \phi_2$ if every valid variable assignment for ϕ_1 is also valid for ϕ_2. Formally: $\Gamma \vdash \phi_1 \Rightarrow \phi_2 \triangleq \forall (t\ \alpha) \in \Gamma.\ \forall y \in t.\ \phi_1[\alpha := y]^*$ 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 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 hold if ϕ_1 held. Viewing the the stack type as a function, ϕ_1 would be a refinement of the function inputs, and ϕ_2 a refinement of the outputs.

We gradually present the (simplified to elide Γ) typing rules inline; the complete definitions are available as part of the supplementary material [Geller et al. 2023].

The typing rules are presented in a *declarative form*, so they describe what types different instructions can have, but are not always sufficient for constructing a type for an instruction. This causes a minor difference 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.

When implementing these rules, we add syntactic annotations on block instructions (see Section 5.1 and Section 6.4), so type checking is syntax directed. However, adding syntactic annotations is straightforward, so we omit them for simplicity. Further, by omitting them, we give our model more flexibility for different implementations. For example, instead of using annotations, it may also be possible to construct the types using an inference algorithm, discussed in Section 8.

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.

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

The constant instruction is a simple example of indexed types. Intuitively, t.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.

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

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.

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

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

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

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

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

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 constrain the result variable α to depend on the truthiness of α_3: (if $(=\alpha_3\ (\mathsf{i32}\ 0))\ (=\alpha\ \alpha_2)\ (=\alpha\ \alpha_1)$). Note that this "if" only introduces syntax that is only evaluated when checking constraint satisfaction between constraint sets.

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

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. The body of a block instruction is not type checked with the same module type context C of the block, but rather with 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} (this is described more below when discussing Rule Br).

The end of an block can be reached either through a branching instruction or by the body e^* being evaluated to a sequence of values. Thus, the label type and postcondition of the body e^* must agree, so that the postcondition of the block is guaranteed to hold no matter how the end of the block is reached. To ensure this, the label type and body's postcondition must have the same stack and index local store ($(t_2\ \alpha_2)^m$ and $(t_l\ \alpha_{l2})^*$), and the constraint set ϕ_3 from the postcondition of the body e^* must imply the label type's constraint set ϕ_2, which is then the same constraint set in the postcondition of the block. Thus, ϕ_2 represents the point of agreement between executing the body, e^*, and any branching instruction from within the body. 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 the aforementioned postcondition $(t_2\ \alpha_2)^m;\ (t_l\ \alpha_{l2})^*;\ \phi_3$.

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 from the beginning, so the precondition is essentially the loop invariant. Thus, the label type and precondition must agree, in contrast to others blocks where the label type and postcondition must agree. If the body e^* is reduced to a sequence of values, these values are returned and the loop exits, as described by the postcondition. When type checking the loop body, the label type is the precondition, and the postcondition of the loop as a whole, $(t_2\ \alpha_2)^*;\ (t_l\ \alpha_{l2})^n;\ \phi_2$ is the same as the postcondition of the body e^* up to implication ($\phi_4 \rightsquigarrow \phi_2$). Instead of checking the body e^* against the precondition of the 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 branching or the first time the loop is executed.

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

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

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

Branching ($\text{br}\ j$) consumes values v^n and jumps to the jth label in the evaluation context, continuing executing with v^n on the stack. The instructions following a branch are not executed, so the postcondition, $ti_2^*;\ l_2;\ \phi_2$, is arbitrary. Similarly, in addition to the consumed values (represented by ti_3^*), the stack may contain arbitrary other values ti_1^*, which are discarded when branching. The precondition of br checks that the current program state satisfies the j-th (counting backwards from the top) label type on the stack C_{label}, ensuring that the condition for branching is met.

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: $ti_2^*; l_2; \phi_2$.

The conditional branch instruction, br_if, consumes a value α from the stack and branches if it is truthy. 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. This conditional information is captured by Rule Br-If: the check against the label type can assume that α is truthy, and the instructions following the br_if can assume that α.

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.

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

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

Direct function calls call i look up the type annotation of the function, $ti_1^*; \phi_3 \to ti_2^*; \phi_2$, 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, producing a union of the two constraint sets. Note, this union is different from most other rules where a single constraint is added to a constraint set using the syntactic constructor form ϕ, P. We extend the constraint set 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 $(ti_1^*; \phi_2 \to ti_2^*; \phi_3)$. 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 the expected type: $\forall 0 \le i < n.(\phi \rightsquigarrow \neg(= (\text{i32}\ i)\ \alpha)) \vee tfi^n(i) = tfi_2$. $tfi^n(i)$ is a shorthand for looking up the ith function type in the sequence tfi^n. 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: $\phi \rightsquigarrow (\text{lt}\ \alpha\ n)$.

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

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

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

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.

$$\textsc{Get-Local}\ \frac{C_{\text{local}}(i) = t \qquad l(i) = (t\ \alpha) \qquad \alpha_2\ \ \texttt{fresh}}{C \vdash \text{get\_local } i : \epsilon;\ l;\ \phi \to (t\ \alpha_2);\ l;\ \phi, (=\alpha\ \alpha_2)} \qquad \begin{array}{l}\textsc{Set-Local} \\ \dfrac{C_{\text{local}}(i) = t \qquad l_2 = l_1[i := (t\ \alpha)]}{C \vdash \text{set\_local } i : (t\ \alpha);\ l_1;\ \phi \to \epsilon;\ l_2;\ \phi}\end{array}$$

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

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.

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

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 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.

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

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

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

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

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

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

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.

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

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

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

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 supplementary material [Geller et al. 2023].

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

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

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 $\vdash embed_m(\textbf{module}\ f^*\ glob^*\ tab^?\ mem^?)$

Proof. (Sketch) The proof follows by induction on the structure of the Wasm typing derivation. The full proof is available as part of the supplementary material [Geller et al. 2023] □

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                                                      module
  func ((i32 α1); (= (i32 1) (i32.gt_u α1 (i32 0)))           func (i32 → i32) local ε
       → (i32 α2); (i32.gt_u α2 (i32 0))) local ε               get_local 0
    get_local 0                                                  i32.const 1
    i32.const 1                                                  i32.div
    i32.div✓                                                   end
  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 $\vdash_i \mathit{erase}_s(s); v^*; \mathit{erase}_{e^*}(e^*) : t^*$

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

Lemma 1 (Instruction Erasure Preserves Typing)**.** If $C \vdash e^* : (t_1\ \alpha_1)^*; l_1; \phi_1 \rightarrow (t_2\ \alpha_2)^*; l_2; \phi_2$, then $\mathit{erase}_C(C) \vdash \mathit{erase}_{e^*}(e^*) : t_1^* \rightarrow t_2^*$

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. In addition, since prechecked instructions cannot trap, as it is not part of their semantics, the type safety of Wasm-precheck ensures that they always successfully reduce to a value.

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 available as part of the supplementary material [Geller et al. 2023].

For the type safety proof, we define an evaluation function $\mathit{eval}(s; v^*; e^*)$ 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. $\boxed{\mathit{eval}(s; v^*; e^*)}$

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

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

Proof. Follows from Lemma 4 (Progress) 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 as part of the supplementary material [Geller et al. 2023]. □

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 $\phi_4 \implies \phi_2$ (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 available as part of the supplementary material [Geller et al. 2023]. □

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 4 (Progress) 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 4 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 $s_{\text{inst}}(i)_{\text{tab}} = \{\text{inst}\ i, \text{func}\ \text{func}\ \mathit{tfi}\ \ldots\}^n$, where $C_{\text{table}} = (n, \mathit{tfi}^n)$ (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 as part of the supplementary material [Geller et al. 2023]. □

5 IMPLEMENTATION

We implement Wasm-precheck as an extension of Wasmtime [Bytecode Alliance 2019], a fast, secure, compliant, 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. Our implementation is available for use as part of the supplementary material [Geller et al. 2023].

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 $\rightsquigarrow$, 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?

In addition, we provide a description of the type annotation process for the benchmarks, and what these annotations look like. This gives an idea of the amount of time and work required to use Wasm-precheck in practice for improved performance, but also suggests the potential of using a static analysis to generate/infer the annotations.

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 four versions of Wasmtime: a "Wasm_dyn" 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); a "Wasm-precheck" version, our extension of Wasmtime implementing Wasm-precheck; and a "Wasm_vm" version, the default configuration of Wasmtime with virtual memory guard pages. 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_dyn	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
Wasm_vm	Unmodified Wasmtime, uses 8GBs of VM for checks (safe)

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 annotation process is described in Section 6.4. The unmodified version is used with the "Wasm_dyn", "Wasm_vm", and "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 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.

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

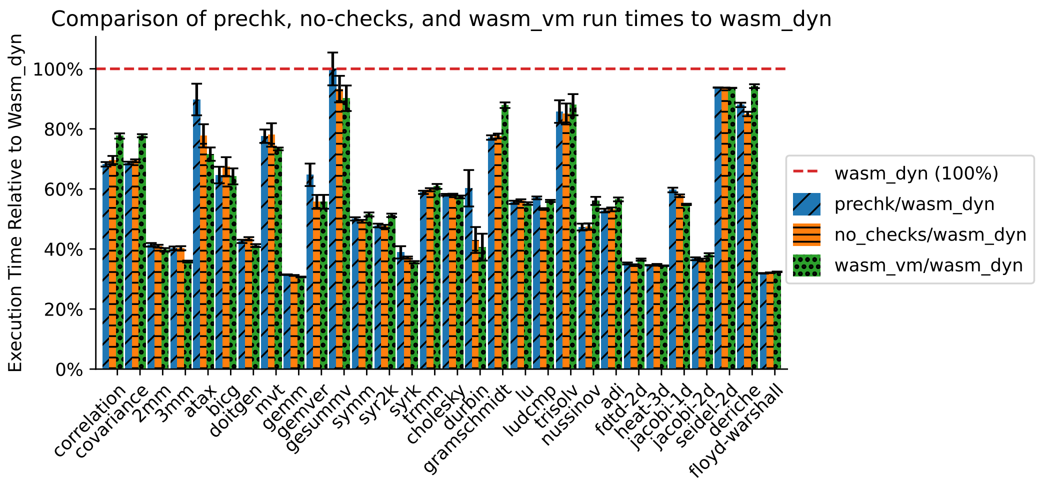

Fig. 5. Comparison of the average run time of PolyBenchC programs; Wasm vs Wasm-no-checks and Wasm vs Wasm-prechk. The error bars show the Standard Error of the Mean.

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 supplementary material includes environment variables [Geller et al. 2023], which have been shown to affect cache behaviour [Curtsinger and Berger 2013].

6.2 Run-time Performance Analysis

Wasm-no-checks vs Wasm_dyn. Compared to Wasm_dyn, the unsafe removal of dynamic checks in Wasm-no-checks achieved an average speed-up of 1.76x, up to a maximum of 3.21x (Figure 5). 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_vm vs Wasm_dyn. Compared to Wasm_dyn, using virtual memory in Wasmtime to optimize memory bounds checks led to an average speed-up of 1.73x, up to a maximum of 3.26x (Figure 5). The run times generally correspond closely to Wasm-no-checks, with some outliers where Wasm-no-checks (and Wasm-precheck) outperform Wasm_vm. We conjecture that in the short-lived benchmarks, there may be measurable cost associated with reserving the necessary virtual memory.

Wasm-precheck vs the rest. Compared to Wasm_dyn, the safe removal of dynamic checks in Wasm-precheck led to an average speed-up of 1.71x, up to a maximum of 3.18x (Figure 5). That is about 97% of the speed-up achieved by Wasm-no-checks on average, and about 99% of the speed-up achieved by Wasm_vm 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 memory intensive programs, Wasm-precheck can accomplish nearly the same mitigation of the costs of dynamic checks as the default Wasm configuration, without the need for VM guard pages.

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 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%.

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.

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 a difference in the performance of the frontend between `wasm-tools` and `wasmtime`, which use the same backend typechecker, but the difference is within the 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.

6.4 Annotation Burden

To give an idea of what annotation burden in Wasm-prechk is like, we relay the process for annotating our PolyBench suite to run in Wasm-prechk, and walk through an example.

Type annotations were all added by a single junior author. We estimate that annotating the first few benchmarks took on the scale of a few hours (some of which was spent debugging Wasm-prechk). After a handful of benchmarks, when a clear pattern emerged and the implementation was reliable, each benchmark took around 30 minutes to annotate, partly constrained by the speed of typechecking when debugging the annotations.

The process of adding the annotations became rather formulaic:

(1) Identify the arrays and their sizes and constrain them to be within the memory bounds, adding a hand-written check for this at the top of the function.
(2) Identify the loop iteration variables, constrain them based on the loop bounds.
(3) Constrain any "free variables" in inner loops based on their assignments in outer loops

A few of the benchmarks required extra annotations, usually for if blocks within the loops, which generally just said that everything stayed the same, but in at least one instance had to do reasoning about the condition for the if (the reasoning is performed automatically by Wasm-precheck).

The majority of the annotations in our suite followed from, in essence, a manual range analysis. We conjecture that much of the analysis could be automated, and provide similar annotations for these types of functions: loops (possibly nested) over an array whose size is known.

An example. The following is an annotation on an inner loop from the `seidel_2d` benchmark.

```
(pre
 (eq (i32 1) (i32.lt_u (local 0) (i32 67108864)))
 (eq (i32 1) (i32.lt_u (i32.add (local 0) (i32 32000000)) (i32 67108864)))
 (eq (i32 1) (i32.lt_u (local 6) (i32 2000)))
 (eq (i32 1) (i32.gt_u (local 6) (i32 1)))
 (eq (local 12) (i32.sub (local 6) (i32 2)))
 (eq (local 3) (i32.add (local 0) (i32.mul (i32 16000)
                                           (i32.sub (local 6) (i32 1)))))
 (eq (i32 1) (i32.lt_u (local 2) (i32 1999)))
 (eq (i32 1) (i32.gt_u (local 2) (i32 0))))
(post
 (eq (local 0) (old_local 0))
 (eq (local 6) (old_local 6)))
```

Recall that the precondition on a loop is essentially a loop invariant; the postcondition is only taken into account for after the loop exits. Local 0 represents an array with a (statically known) size of 32,000,000 bytes. The precondition checks that both the starts and ends of the array are within the static size of memory, 67,108,864 bytes (1024 pages of 65536 bytes each), and the postcondition ensures that the size is unchanged when the loop exits. Locals 2 and 6 are loop iteration variables with respective statically known loop bounds of [1, 1998] and [2, 1999] (inclusive). The loop bounds are reflected in the precondition. Because we are looking at an inner loop, the iteration variable from the outer loop, local 6, should be unchanged when this inner loop exits.

Finally, locals 12 and 3 store the result of computations performed prior to this loop; they are constrained using their respective computations. Essentially, they are the result of moving a computation that is invariant in the loop to be calculated once before the loop instead of on every iteration.

In some cases, these stored computations were constrained based on an upper bound rather than how they were assigned. Using a maximum bound results in more succinct but less precise annotations. Conversely, spelling out the constraints based on the computation, as on line 7 of the above example, is more verbose but also more precise. In the above example, the extra precision makes sense, as local 6 has a lower bound as well as an upper bound. However, there are other cases where this extra precision is necessary, usually when a stored computation is used to access multiple arrays of different sizes. If we did not have the lower bound on local 6, we could substitute the upper bounds for locals 0 and 6 in line 7 of the above example, resulting in the following more compact constraint on local 3:

```
(eq (i32 1) (i32.lt_u (local 3) (i32 63968000)))
```

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 prechecked 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,

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

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. 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 $\alpha_m = oldsize$ if the result is -1, and $\alpha_m = newsize$ 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 the 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 by safely removing explicit dynamic checks 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.

ACKNOWLEDGMENTS

We wish to gratefully acknowledge the feedback of those who helped with our work or its presentation in many ways. Paul Chen and Mohammad Shahrad for providing context for the requirements and possible role of Wasm in the serverless, cloud, and microservices space. Ivan Bestchastnikh, for his contributions during the conception of the research question. Ross Tate for connecting us with the Wasm community and offering feedback on design decisions based on the desires of the community. Andreas Rossburg for pointing us towards Wasmtime as a candidate runtime/compiler to experiment with. Deian Stefan for pointing us towards the PolyBenchC suite.

We are also thankful for the thoughtful suggestions for improvement provided by the reviewers, as well as past reviewers of this paper in other venues.

We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC), funding reference number RGPIN-2019-04207. Cette recherche a été financée par le Conseil de recherches en sciences naturelles et en génie du Canada (CRSNG), numéro de référence RGPIN-2019-04207. This work was also supported by Huawei Technologies Canada Co., Ltd.

DATA AVAILABILITY STATEMENT

The appendix, containing the full typing rules and proof details, as well as the software artifacts and annotated benchmarks used in the evaluation (see Section 5 and Section 6), have been made publically available [Geller et al. 2023].

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 *Theory and Practice of Software, International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS/ETAPS)*. https://doi.org/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

Adam T. Geller, Justin Frank, and William J. Bowman. 2023. *Indexed Types for a Statically Safe WebAssembly Artifact*. https://doi.org/10.5281/zenodo.8423363

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 *International Conference on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/3062341.3062363

Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha. 2019. Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code. In *USENIX Annual Technical Conference (USENIX ATC)*. https://doi.org/10.5555/3358807.3358817

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*. https://doi.org/10.1007/978-3-319-73721-8_13

A. Mine. 2001. The octagon abstract domain. In *Working Conference on Reverse Engineering*. 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)* (1999). https://doi.org/10.1145/319301.319345

George C. Necula. 1997. Proof-Carrying Code. In *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. *OOPSLA* (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 *International 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

Hongwei Xi and Robert Harper. 2001. A Dependently Typed Assembly Language. In *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* (1997). https://doi.org/10.1016/S0304-3975(97)00062-5

Received 2023-07-11; accepted 2023-11-07