A Type System with Subtyping for WebAssembly's Stack Polymorphism

Dylan McDermott[1], Yasuaki Morita[1], and Tarmo Uustalu[1]

Reykjavik University, Reykjavík, Iceland
dylanm@ru.is, yasuaki20@ru.is, tarmo@ru.is

Abstract. We propose a new type system for WebAssembly. It is a refinement of the type system from the language specification and is based on type qualifiers and subtyping. In the WebAssembly specification, a typable instruction sequence gets many different types, depending in particular on whether it contains instructions such as **br** (unconditional branch) that are stack-polymorphic in an unusual way. But in general, one cannot single out a canonical type for a typable instruction sequence. We introduce qualifiers on code types to describe their flavor of stack polymorphism and a subtyping relation on such qualified types. Our type system gives every typable instruction sequence a canonical type that is principal. We show that the new type system is in a precise relationship to the type system given in the WebAssembly specification. In addition, we describe a typed functional-style big-step semantics based on this new type system underpinned by an indexed graded monad and prove that it prevents certain kinds of runtime errors. We have formalized our type system, inference algorithm and semantics in Agda.

1 Introduction

WebAssembly (Wasm) [?] is a statically typed, stack-oriented bytecode language. Wasm has been designed with a formal semantics [?]. Watt [?] formalized the type system, the type checker, the small-step semantics and a proof of type soundness in Isabelle. Later, Wasm 1.0 became a W3C Recommendation [?], and Huang [?] and Watt et al. [?] came with formalizations in Coq. As type soundness gives safety, Wasm's type system plays a significant role in its semantics.

A key feature of the type system of Wasm is that it tracks how the stack shape evolves in program execution. Stacks are typed by their shapes, which are lists of value types. A piece of code is typed by a pair of stack types, a pretype and posttype. In Wasm, most instructions are typed monomorphically with their (net) stack effect, i.e., types for the portions of stack they pop and push. Instructions for unconditional control transfer like **br** however are typed differently, polymorphically and in an unusual way. Instruction sequences are typed polymorphically (in particular one cannot read off from the type how long a prefix of the initial stack is actually touched) and typing of instruction sequences involving **br** becomes subtle.

In this paper, we analyze the stack polymorphism of the type system of Wasm in detail on a minimalistic fragment of the language. We introduce a variant type system (Dir) that uniformizes the typing of instructions and instruction sequences making both stack-polymorphic in an adequate sense. Dir stands in a precise relationship to the type system of the language specification (which we call Spec); in particular instruction sequences get exactly the same types. Then we refine this type system to another one (which we call Sub) that has subtyping and equips all instructions and instruction sequences, notably **br** and instruction sequences involving **br**, with canonical types in the form of principal types. We achieve this by introducing the distinction between ordinary ("univariate") stack polymorphism (in the type of the untouched suffix of the stack) from the unusual "bivariate" stack polymorphism of Wasm characteristic to **br** and instruction sequences involving it. On top of Sub, we build a typed big-step operational semantics in which run-time errors cannot occur. We also define an untyped big-step semantics that agrees with this typed semantics on typed programs when invoked on initial stacks that the typed semantics accepts.

Our type system, inference algorithm with their properties and the typed and untyped big-step semantics have been formalized in Agda; the development is available at `https://github.com/moritayasuaki/NFM2022-proofs`.

2 A small fragment of Wasm

For the sake of simplicity, we work with a minimalistic fragment of Wasm. The syntax of the language is given in Figure 1. A piece of code in this language is either an instruction or an instruction sequence.

$a, r, m, d, e \ \in \ \mathbb{N}$		stack types (called result types in the spec.)
$t ::= a \to r$		code types (called stack types in the spec.)
$\ell \ \in \ \mathbb{N}$		label indices
$z \ \in \ \mathbb{Z}_{32}$		32-bit integers
$uop ::= \textbf{eqz} \mid \ldots$		unary numeric operations
$bop ::= \textbf{add} \mid \ldots$		binary numeric operations
$i ::= \textbf{const}\, z \mid uop \mid bop$		numeric instructions
$\mid \ \textbf{block}_t\, is\, \textbf{end} \mid \textbf{loop}_t\, is\, \textbf{end}$		block-like instructions
$\mid \ \textbf{br\_if}\, \ell \mid \textbf{br}\, \ell$		branch instructions
$is ::= \varepsilon \mid is\ i$		instruction sequences
$c ::= i \mid is$		code

Fig. 1: Syntax of reduced Wasm

Since our focus is on stack manipulation and typing thereof, we have left out all unrelated aspects of Wasm, even the linear memory; also we do not have functions. To keep the presentation as clean as possible, we do not even have multiple value types. Of Wasm's value types **i32**, **i64**, **f32**, **f64** etc., we have kept only one, **i32**. A stack type in Wasm is a list of value types. Since in our reduced language, there is just one value type, a stack type boils down to a natural number (for the length of the stack). With this simplification, issues such as values of wrong type in the stack and value-polymorphism (of, e.g., the **drop** instruction) disappear. Having just numbers as stack types is arguably a significant simplification. Still all phenomena we want to discuss are maintained and the arguments in this paper scale to lists of value types as stack types by replacing the total order on natural numbers by the (prefix) partial order on lists. The possibility of value-type mismatch then leads to partiality of the central operations on stack types and code types that are total in this paper.

There are three main categories of instructions (numeric, block-like and branch instructions), and execution of each instruction is defined in the same way as in [?,?]. A numeric instruction pops some arguments from the current local stack (the global stack or the local stack of the closest encompassing block-like instruction), performs the corresponding operation, and pushes the result.

A block-like instruction **block** or **loop** type-annotated with $a \to r$ pops a values ("arguments") from the current local stack, constructs its own local stack containing these arguments, and executes the inner instruction sequence on this new local stack as current. If this terminates normally, there must be r values ("results") left on this local stack. The local stack is then destroyed and the r values are pushed to the parent local stack, which becomes current.

The unconditional branch instruction **br** ℓ is a jump instruction targeting either the end or the beginning of the ℓ-th encompassing block-like instruction, depending on whether it is a block or a loop. If the type annotation on this instruction is $a \to r$, then, before the jump, either r or a values are popped from the current local stack, the local stacks of enclosing block-like instructions up to the jump target are emptied and destroyed, the local stack of the jump target is emptied and the r or a values are pushed to it; and it becomes current. The conditional branch instruction **br_if** ℓ behaves similarly except that it consumes the top of the current local stack as a condition.

Type system

Figure 2 shows the typing rules of our chosen subset of Wasm. This type system matches the Wasm specification, and we call this type system Spec.

Typing judgements for instructions i and instruction sequences is have similar forms $rs \vdash^{\mathsf{I}} i : a \to r$ and $rs \vdash^{\mathsf{S}} is : a \to r$ where the code type $a \to r$ describes in each case in some way (which we will discuss in detail) the stack effect of i or is in terms of a pair of stack types: the shapes of the local stack before (a, for "arguments") and after (r, for "results") a possible execution. The typing context rs, which is a list of stack shapes, records the result resp. argument types of the **block** or **loop** instructions encompassing i or is, in the inside-out order. We write $rs \mathbin{!!} \ell$ for the ℓ-th element of rs ($\ell < |rs|$).

In this type system, every instruction except for **br** gets a unique code type (if it gets one at all). For numeric instructions, the meaning of this type is clear: $a \to r$ reflects the numbers of arguments and results of the operation,

$$\frac{}{rs \vdash^{\mathsf{I}} \textsf{const}\ z : 0 \to 1}\ \textsc{Const} \qquad \frac{}{rs \vdash^{\mathsf{I}} uop : 1 \to 1}\ \textsc{Uop} \qquad \frac{}{rs \vdash^{\mathsf{I}} bop : 2 \to 1}\ \textsc{Bop}$$

$$\frac{r :: rs \vdash^{\mathsf{S}} is : a \to r}{rs \vdash^{\mathsf{I}} \textsf{block}_{a \to r}\ is\ \textsf{end} : a \to r}\ \textsc{Block} \qquad \frac{a :: rs \vdash^{\mathsf{S}} is : a \to r}{rs \vdash^{\mathsf{I}} \textsf{loop}_{a \to r}\ is\ \textsf{end} : a \to r}\ \textsc{Loop}$$

$$\frac{rs\ !!\ \ell = r}{rs \vdash^{\mathsf{I}} \textsf{br\_if}\ \ell : 1 + r \to r}\ \textsc{Br\_if} \qquad \frac{rs\ !!\ \ell = r}{rs \vdash^{\mathsf{I}} \textsf{br}\ \ell : r + d \to e}\ \textsc{Br}$$

$$\frac{}{rs \vdash^{\mathsf{S}} \varepsilon : a \to a}\ \textsc{Empty} \qquad \frac{rs \vdash^{\mathsf{S}} is : a \to m + d \quad rs \vdash^{\mathsf{I}} i : m \to r}{rs \vdash^{\mathsf{S}} is\, i : a \to r + d}\ \textsc{Seq}$$

Fig. 2: Typing rules of type system Spec, following the specification of Wasm

the numbers of elements popped from and pushed onto the stack. The type of **br_if** ℓ according to the rule Br_if also reflects the operational semantics: **br_if** ℓ pops the top of the stack as a condition and then pops $r(= rs\ !!\ \ell)$ next elements additionally if this condition is non-zero (true). The argument type of **br_if** ℓ is therefore $1 + r$. Although **br_if** ℓ terminates abnormally by a jump in this case (thereby not posing any requirement on the result type), the same r next elements remain on the stack if the condition is zero (false). Therefore, the result type must be r since the code type must cover both cases; in the false case, we have to pretend that $1 + r$ elements are popped and the r last of those are pushed back (even if in reality only one element is popped and none pushed). We postpone a discussion of **br** ℓ.

In contrast, every instruction sequence gets many code types. For instance, the empty sequence ε in Empty gets code types $a \to a$ for any natural number a. If we take 0 for a, then it becomes $0 \to 0$. This choice can be said the tightest because the empty sequence consumes and produces nothing on the stack. The rule also allows us to choose $a = 1$. It is natural to think of the empty sequence as the identity function on the stack. However, the type $1 \to 1$ no longer tells us that the value at the top of the stack remains unchanged. In such a sense, we would say $\varepsilon : 1 \to 1$ is a reasonable typing but loose in comparison to $\varepsilon : 0 \to 0$. Though the specification does not give a specific term for this phenomenon, we call it *univariate stack polymorphism*, or simply, *univariate polymorphism* (as opposed to bivariate polymorphism, which comes later).[1] Univariate polymorphism allows code types to be loosened by adding the *same* number to both the argument and result type corresponding to an untouched part of the local stack.

The premises of the typing rule Seq for the sequencing $is\ i$ of is and i require the result type $m + d$ of is to be at least the argument type m of i. This rule can be intuitively motivated relying on univariate polymorphism of instructions (which this type system does enjoy, but which is semantically justified). First, we think of the type $m + d \to r + d$ as a loosened version of the type $m \to r$ of i, although no typing rules allow us to give i this type officially. Since this

[1] We use the term 'stack polymorphism' in the sense of Morrisett et al. [?], viz. polymorphism of stack functions in the type of the untouched part of the stack.

loosening has made the types at the middle equal (the result type of is and the argument type of i have both become $m+d$), we can consider that the argument type a of is and the result type $r+d$ of i form a type for the sequence $is\,i$.

We notice that an instruction i and the singleton instruction sequence i (i.e., $\varepsilon\ i$) are not treated the same way in Spec. For example, **const** 17 as an instruction only has type $0 \to 1$ in any context, but as an instruction sequence it has the type $d \to 1+d$ for any d (since ε admits the type $d \to 0+d$).

Bivariate stack polymorphism

Although **br** ℓ is operationally the same as (**const** 1) (**br_if** ℓ), it has different characteristics in the type system (which does not involve any constant propagation analysis). The rule Br assigns many types to the instruction **br** ℓ: the d and e in the conclusion are arbitrary natural numbers. This is a big difference from the other instructions, which all get at most one type. Although the Wasm specification takes *stack polymorphism* to mean only this phenomenon, we will refer to it more specifically as *bivariate stack polymorphism*, or simply, *bivariate polymorphism* since d and e are independent metavariables for stack types. The natural intuition "code type = local stack type before and after" is no longer useful, since an execution of **br** cannot terminate normally at "after"; the next instructions in an encompassing block-like instruction or the end of it are never reached. Thanks to bivariate polymorphism, it is possible to place any instruction immediately after **br**, and this instruction will be unreachable code. In [**?**], an example of the use of bivariate stack polymorphism in compilers is discussed.

Typing of unreachable code is quite subtle in this type system. For example, the following instruction sequence is untypable when $r = 0$ and typable when $r \geq 1$, even though the instruction **const** 17 and the end of the **loop** are unreachable:

$$\textbf{block}_{0\to 0}\ \textbf{loop}_{0\to r}\ (\textbf{br}\ 1)\ (\textbf{const}\ 17)\ \textbf{end}\ (\textbf{br}\ 0)\ \textbf{end}$$

We notice that the design of Spec is uneven in that **br** and instruction sequences are stack-polymorphic, but instructions other than **br** are not. Yet "morally" they should all be stack-polymorphic. The rules for sequencing "fix" this discrepancy—or cover it up, depending on how one looks at this. In the next section, we introduce a variant type system Dir, which remedies this issue.

3 Type system Dir with "direct" sequential composition

The typing rules of the type system Dir are given in Figure 3. They give many types not only to **br**, but also to other single instructions. The typing rule in Dir loosens the type assigned to an instruction by Spec by adding any natural number d to both the argument and result types. For the bivariate polymorphic instruction **br**, the typing rule is as in Spec. In other words, Dir has stack polymorphism (univariate or bivariate) for all instructions. The rule for sequencing is "direct": it only admits the case where the result type of is and the argument

$$\frac{}{rs \vdash \textsf{\textbf{const}}\ z : d \to 1 + d}\ \textsc{Const}$$

$$\frac{}{rs \vdash uop : 1 + d \to 1 + d}\ \textsc{Uop} \qquad \frac{}{rs \vdash bop : 2 + d \to 1 + d}\ \textsc{Bop}$$

$$\frac{r :: rs \vdash is : a \to r}{rs \vdash \textsf{\textbf{block}}_{a \to r}\ is\ \textsf{\textbf{end}} : a + d \to r + d}\ \textsc{Block} \qquad \frac{a :: rs \vdash is : a \to r}{rs \vdash \textsf{\textbf{loop}}_{a \to r}\ is\ \textsf{\textbf{end}} : a + d \to r + d}\ \textsc{Loop}$$

$$\frac{rs \mathbin{!!} \ell = r}{rs \vdash \textsf{\textbf{br\_if}}\ \ell : 1 + r + d \to r + d}\ \textsc{Br\_if} \qquad \frac{rs \mathbin{!!} \ell = r}{rs \vdash \textsf{\textbf{br}}\ \ell : r + d \to e}\ \textsc{Br}$$

$$\frac{}{rs \vdash \varepsilon : r \to r}\ \textsc{Empty} \qquad \frac{rs \vdash is : a \to m \quad rs \vdash i : m \to r}{rs \vdash is\, i : a \to r}\ \textsc{Seq}$$

Fig. 3: Typing rules in the type system Dir

type of i coincide. This is fine now since all instructions have become stack-polymorphic.

For single instructions, Dir gives more valid types than Sub does. For example, $rs \vdash \textsf{\textbf{const}}\ 17 : d \to 1 + d$ in Dir, but in Spec, only $rs \vdash^{\mathsf{I}} \textsf{\textbf{const}}\ 17 : 0 \to 1$ can be derived. (But also recall that Dir does derive $rs \vdash^{\mathsf{S}} \textsf{\textbf{const}}\ 17 : d \to 1 + d$: for instruction sequences the two type systems give the same types.)

Theorem 1 **(Dir vs. Spec).**

$$rs \vdash_{\mathsf{Dir}} i : a \to r \iff (\exists d, a', r'.\, a = a' + d \wedge r = r' + d \wedge rs \vdash^{\mathsf{I}}_{\mathsf{Spec}} i : a' \to r')$$
$$rs \vdash_{\mathsf{Dir}} is : a \to r \iff rs \vdash^{\mathsf{S}}_{\mathsf{Spec}} is : a \to r$$

Proof. ($\Longrightarrow$) By mutual induction on the derivation trees of $rs \vdash_{\mathsf{Dir}} i : a \to r$ and $rs \vdash_{\mathsf{Dir}} is : a \to r$.

($\Longleftarrow$) We replace the backwards implication of the statement for i with the equivalent property that

$$(\forall d.\ rs \vdash_{\mathsf{Dir}} i : a + d \to r + d) \Longleftarrow rs \vdash^{\mathsf{I}}_{\mathsf{Spec}} i : a \to r$$

and then proceed by mutual induction on the derivation trees of $rs \vdash_{\mathsf{Spec}} i : a \to r$ and $rs \vdash_{\mathsf{Spec}} is : a \to r$.

The type system Dir is free of some of the problems of Spec: both instructions and instruction sequences get all types they should reasonably get. However, there is no single canonical type among them in all cases. Instructions other than **br** and the empty sequence do have "tightest" types, but **br** and general instruction sequences (specifically those containing **br**) do not. We will now improve on this and introduce a type system Sub where even **br** and instruction sequences have canonical types.

4 Type system Sub with qualifiers and subtyping

We introduce two qualifiers uni and bi (for "univariate" and "bivariate", using q as a typical metavariable for these qualifiers), and a partial order $\leq$ on them:

$$\frac{}{\mathsf{bi} \leq q} \qquad \frac{}{\mathsf{uni} \leq \mathsf{uni}}$$

$$\frac{}{a \to_{\mathsf{bi}} r <: a + d \to_q r + e}\ \textsc{Subt}_{\mathsf{bi}} \qquad \frac{}{a \to_{\mathsf{uni}} r <: a + d \to_{\mathsf{uni}} r + d}\ \textsc{Subt}_{\mathsf{uni}}$$

$$\frac{}{rs \vdash \textbf{const}\ z : 0 \to_{\mathsf{uni}} 1}\ \textsc{Const} \qquad \frac{}{rs \vdash uop : 1 \to_{\mathsf{uni}} 1}\ \textsc{Uop} \qquad \frac{}{rs \vdash bop : 2 \to_{\mathsf{uni}} 1}\ \textsc{Bop}$$

$$\frac{r :: rs \vdash is : a \to_{\mathsf{uni}} r}{rs \vdash \textbf{block}_{a \to r}\ is\ \textbf{end} : a \to_{\mathsf{uni}} r}\ \textsc{Block} \qquad \frac{a :: rs \vdash is : a \to_{\mathsf{uni}} r}{rs \vdash \textbf{loop}_{a \to r}\ is\ \textbf{end} : a \to_{\mathsf{uni}} r}\ \textsc{Loop}$$

$$\frac{rs\ !!\ \ell = r}{rs \vdash \textbf{br\_if}\ \ell : 1 + r \to_{\mathsf{uni}} r}\ \textsc{Br\_if} \qquad \frac{rs\ !!\ \ell = r}{rs \vdash \textbf{br}\ \ell : r \to_{\mathsf{bi}} 0}\ \textsc{Br}$$

$$\frac{}{rs \vdash \varepsilon : 0 \to_{\mathsf{uni}} 0}\ \textsc{Empty} \qquad \frac{rs \vdash is : a \to_q m \quad rs \vdash i : m \to_{q'} r}{rs \vdash is\, i : a \to_{q \sqcap q'} r}\ \textsc{Seq}$$

$$\frac{rs \vdash c : t' \quad t' <: t}{rs \vdash c : t}\ \textsc{Subs}$$

Fig. 4: Subtyping and typing rules in the type system Sub

In the type system Sub code types have the form $a \to_q r$; the qualifier q specifies whether the code is univariately or bivariately stack-polymorphic. Code types are ordered by a subtyping relation $<:$, defined by the top two rules of Figure 4.

The remainder of Figure 4 consists of typing rules of Sub. All instructions except **br** are assigned a uni-type by their typing rule; **br** gets a bi-type. This way, all single instructions including **br**, and the empty instruction sequence, get assigned their tightest type. The typing rule Seq for sequencing is as in Dir, but the qualifier in the conclusion is the meet $\sqcap$ of the qualifiers in the premises. This operation is defined by $\mathsf{uni} \sqcap \mathsf{uni} = \mathsf{uni}$ and $q \sqcap q' = \mathsf{bi}$ otherwise. All looseness of typing is introduced by a subsumption rule Subs that applies to both instructions and instruction sequences.

Proposition 1 **(**Sub **vs.** Dir**, take 1).**

$$rs \vdash_{\mathsf{Sub}} c : a \to_{\mathsf{uni}} r \iff rs \vdash_{\mathsf{Dir}} c : a \to r$$

Proof. ($\Longrightarrow$) By induction on the derivation of $rs \vdash_{\mathsf{Sub}} c : a \to_{\mathsf{uni}} r$ (by which we mean mutual induction on the derivation of $rs \vdash_{\mathsf{Sub}} c : a \to_{\mathsf{uni}} r$ for the two cases i and is of c).

($\Longleftarrow$) By induction on the derivation in $rs \vdash_{\mathsf{Dir}} c : a \to r$.

Lemma 1.

$$rs \vdash_{\mathsf{Dir}} c : a{+}d \to r \wedge rs \vdash_{\mathsf{Dir}} c : a \to r{+}e \wedge (d > 0 \vee e > 0) \implies rs \vdash_{\mathsf{Sub}} c : a \to_{\mathsf{bi}} r$$

Proof. By induction on c (by which we mean mutual induction on c for the two cases i and is of c).[2]

Theorem 2 **(**Sub **vs.** Dir**).**

$$rs \vdash_{\mathsf{Sub}} c : a_0 \to_q r_0 \iff (\forall a, r.\, a_0 \to_q r_0 <: a \to_{\mathsf{uni}} r \implies rs \vdash_{\mathsf{Dir}} c : a \to r)$$

[2] For a detailed proof, see Appendix A.

Proof. From Proposition 1 and Lemma 1.[3]

Type inference

We define a type inference algorithm for Sub. We prove this algorithm computes a principal type for a given piece of code c for a given type context rs, provided it is typable in it at all, i.e., it computes a derivable type which is a subtype of every other derivable type.

The algorithm is defined as a function infer recursive on c (i.e., mutually recursive on the two cases of c being an instruction or an instruction sequence) in Fig. 5. The inferred type is for every instruction and also for the empty sequence the one from the conclusion of the typing rule, but not in the case of sequencing. For numeric instructions and the empty sequence ε, their typing rules give them one type and this is the type inferred. For a given context, the types of **br** and **br_if** are also determined uniquely, but differently from all other instructions **br** gets a bi-type. The types of **block** and **loop** are determined by the annotation, but the instruction sequence inside may fail to admit this type. For this reason, infer is called recursively on this sequence to check its compatibility with the annotation.

The inferred type for a sequence $is\ i$ is defined by an operation $\oplus$ on qualified code types. Firstly, to satisfy the premises of the rule Seq, the operation $\oplus$ needs to reconcile the middle stack types m and m' of the inferred types $a \to_q m$ and $m' \to_{q'} r$ of is and i. The unified middle type is actually $\max(m, m')$, whatever q and q' are.[4] But the possible invocations of Subs differ depending on q and q'. For example, if we have $rs \vdash is : a \to_{\mathsf{bi}} m$, then we can achieve $rs \vdash is : a \to_{\mathsf{bi}} \max(m, m')$, but if we have $rs \vdash is : a \to_{\mathsf{uni}} m$, then we only get $rs \vdash is : a + (\max(m, m') - m) \to_{\mathsf{uni}} \max(m, m')$. As a result of exactly the same thing happening for $rs \vdash i : m' \to_q r$, the operation $\oplus$ can be defined uniformly in the four cases of q, q' using the "monus" operation $m \mathbin{\dot{-}} m' = \max(m, m') - m$ and its qualified version $m \mathbin{\dot{-}}_{\mathsf{uni}} m' = m \mathbin{\dot{-}} m'$, $m \mathbin{\dot{-}}_{\mathsf{bi}} m' = 0$. We define

$$(a \to_q m) \oplus (m' \to_{q'} r) = a + (m' \mathbin{\dot{-}}_q m) \to_{q \sqcap q'} r + (m \mathbin{\dot{-}}_{q'} m')$$

Theorem 3 (Soundness of type inference of Sub).

$$\mathsf{infer}\ c\ rs = \mathsf{Just}\ t \implies rs \vdash c : t$$

Proof. By induction on c.

Theorem 4 (Completeness of type inference of Sub).

$$rs \vdash c : t \implies (\exists t_0.\ \mathsf{infer}\ c\ rs = \mathsf{Just}\ t_0 \wedge t_0 <: t)$$

Proof. By induction on the derivation of $rs \vdash c : t$.

[3] For a detailed proof, see Appendix A.

[4] The intermediate type $\max(m, m')$ here is always defined just because we have one value type and stack types are natural numbers. If we consider multiple value types, the stack types m and m' are no longer natural numbers but lists of value types. In this setting, the unified middle type is defined only if one of m and m' is a prefix of the other; when this is not the case, the instruction sequence is not typable.

$$\mathsf{infer}\ c\ rs : \mathsf{Maybe}\ \mathit{CodeType}$$

$$
\begin{aligned}
\mathsf{infer}\ (\mathbf{const}\ z)\ rs &= \mathsf{Just}(0 \to_{\mathsf{uni}} 1)\\
\mathsf{infer}\ uop\ rs &= \mathsf{Just}(1 \to_{\mathsf{uni}} 1)\\
\mathsf{infer}\ bop\ rs &= \mathsf{Just}(2 \to_{\mathsf{uni}} 1)\\
\mathsf{infer}\ (\mathbf{block}_{a \to r}\ is\ \mathbf{end})\ rs &= \mathsf{do}\\
&\quad tis \leftarrow \mathsf{infer}\ is\ (r :: rs)\\
&\quad \mathsf{if}\ tis <: a \to_{\mathsf{uni}} r\ \mathsf{then}\ \mathsf{Just}(a \to_{\mathsf{uni}} r)\ \mathsf{else}\ \mathsf{Nothing}\\
\mathsf{infer}\ (\mathbf{loop}_{a \to r}\ is\ \mathbf{end})\ rs &= \mathsf{do}\\
&\quad tis \leftarrow \mathsf{infer}\ is\ (a :: rs)\\
&\quad \mathsf{if}\ tis <: a \to_{\mathsf{uni}} r\ \mathsf{then}\ \mathsf{Just}(a \to_{\mathsf{uni}} r)\ \mathsf{else}\ \mathsf{Nothing}\\
\mathsf{infer}\ (\mathbf{br\_if}\ \ell)\ rs &= \mathsf{if}\ \ell < |rs|\ \mathsf{then}\ \mathsf{Just}((1 + rs\ !!\ \ell) \to_{\mathsf{uni}} (rs\ !!\ \ell))\ \mathsf{else}\ \mathsf{Nothing}\\
\mathsf{infer}\ (\mathbf{br}\ \ell)\ rs &= \mathsf{if}\ \ell < |rs|\ \mathsf{then}\ \mathsf{Just}((rs\ !!\ \ell) \to_{\mathsf{bi}} 0)\ \mathsf{else}\ \mathsf{Nothing}
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{infer}\ \varepsilon\ rs &= \mathsf{Just}(0 \to_{\mathsf{uni}} 0)\\
\mathsf{infer}\ (is :: i)\ rs &= \mathsf{do}\\
&\quad tis \leftarrow \mathsf{infer}\ is\ rs\\
&\quad ti \leftarrow \mathsf{infer}\ i\ rs\\
&\quad \mathsf{Just}(tis \oplus ti)
\end{aligned}
$$

Fig. 5: Type inference for Sub

Pomonoid

The set of code types of Sub, together with its subtyping relation $<:$, the element $0 \to_{\mathsf{uni}} 0$ and the operation $\oplus$ form a *pomonoid* (a partially ordered monoid). This pomonoid is a generalization for the qualified case of the *stack effect pomonoid* first considered as such by Pöial [**?**] (see also [**?**]) and studied earlier in algebra as the polycyclic monoid (the inverse envelope of a free monoid) by Nivat and Perrin [**?**] (modulo the fact that we have replaced lists of value types as stack types by natural numbers, which gives the bicyclic monoid).

That we get a pomonoid is very reasonable: it is reflects the expectation that sequential composition of two pieces of code should be associative (up to semantic equivalence) and have the empty code as the unit, also that it should not matter whether subsumption is applied to one of the two pieces of code or to the composition. (Notice though that in Wasm we have no syntactic operation of composition of two sequences of instructions.) We have a reason to return to this pomonoid structure in the next section.

5 Typed big-step semantics based on Sub

We now demonstrate Sub in action by building on it a typed functional-style big-step semantics (a denotational semantics)[5] of simplified Wasm.

[5] For a discussion of the merits of functional-style rather than the usual relational-style big-step semantics in constructive programming theory and the precise relationship between the two, see e.g., [**?**].

The denotation of a typing derivation of a piece of code is eventually a function that takes

- a natural number as a bound on the number of backjumps that can be made within the loops the piece of code is encompassed in[6]
- and a list of integers as an initial local stack,

runs the code and returns either

- nothing if the bound on the number of backjumps was exceeded,
- or a final local stack from normal termination (in the case of a bi-type, this is not a possibility),
- or a portion of stack to transfer to the branch target from abnormal termination from a jump to a label index.

Denotations of derivable subtypings coerce between such functions.

The semantic function for code types is therefore defined by

$$[\![a \to_q r]\!]\ rs = \mathbb{N} \to \mathbb{Z}_{32}^{a} \to \mathsf{Maybe}\ (\mathsf{NT}_q(r) + \sum_{\ell < |rs|} \mathbb{Z}_{32}^{rs!!\ell})$$

where $\mathsf{NT}_{\mathsf{bi}}(r) = \mathbf{0}$ and $\mathsf{NT}_{\mathsf{uni}}(r) = \mathbb{Z}_{32}^{r}$.[7]

The semantic functions for derivable subtypings and typing derivations are defined in Figure 6. The definitions use functions $\mathsf{split}\, a : \mathbb{Z}_{32}^{a+d} \to \mathbb{Z}_{32}^{a} \times \mathbb{Z}_{32}^{d}$ that split a given local stack into two parts, with the first part containing the first a elements and the second containing the rest. Function $\mathsf{take}\ a$ only give the first part.

Importantly, despite the fact that denotations $\left[\!\!\left[rs \overset{\pi}{\vdash} c : t \right]\!\!\right]$ are defined on particular derivations, any two derivations of the same typing judgement $rs \vdash c : t$ have the same denotation. We prove this by relating the semantics to type inference: if there is a derivation of $rs \vdash c : t$, then, by completeness of type inference (Theorem 4), there exists a unique t_0 (depending only on c and rs) such that $\mathsf{infer}\ rs\ c = \mathsf{Just}\ t_0$ and $t_0 <: t$, and by soundness (Theorem 3) there is also a derivation of $rs \vdash c : t_0$. We relate the denotations of the derivations of $rs \vdash c : t$ and $rs \vdash c : t_0$.

Proposition 2 (Coherence of typed semantics). *If $rs \vdash c : t$, then*

$$\left[\!\!\left[rs \overset{\pi}{\vdash} c : t \right]\!\!\right] = [\![t_0 <: t]\!]\ rs\ \left[\!\!\left[rs \overset{\pi_0}{\vdash} c : t_0 \right]\!\!\right]$$

[6] To avoid coinduction in the formalization of our constructive mathematical development, we make sure that all program executions terminate by limiting the number of backjumps—the only source of nontermination in simplified Wasm. This is poor man's domain theory that works well for our purposes; what we are using is a certain version of the delay monad [?].

[7] Notice that $[\![a \to_{\mathsf{bi}} r]\!]$ does not really depend on r. This suggests that bi-types should perhaps not have a posttype at all. Such a design is possible, we look at this in Appendix B. This gives a simpler type system that accepts more programs but still provides safety.

$$\llbracket t <: t' \rrbracket\ rs : \llbracket t \rrbracket\ rs \to \llbracket t' \rrbracket\ rs$$

$$\begin{aligned}
&\llbracket a \to_q r <: a + d \to_{q'} r + e \rrbracket\ f\ n\ stk\ \ = \\
&\quad \mathsf{let}\ (astk, pstk) = \mathsf{split}\ a\ stk\ \mathsf{in\ case}\ f\ n\ astk\ \mathsf{of} \\
&\qquad \mathsf{Nothing} \mapsto \mathsf{Nothing} \\
&\qquad \mathsf{Just}(\mathsf{Left}\ stk') \mapsto \mathsf{Just}(\mathsf{Left}(stk' \mathbin{+\!\!+} pstk)) \\
&\qquad \mathsf{Just}(\mathsf{Right}(\ell, stk')) \mapsto \mathsf{Just}(\mathsf{Right}(\ell, stk'))
\end{aligned}$$

$$\left\llbracket rs \overset{\pi}{\vdash} c : t \right\rrbracket : \llbracket t \rrbracket\ rs$$

$$\begin{aligned}
&\left\llbracket \frac{}{rs \vdash \mathbf{const}\ z : 0 \to_{\mathsf{uni}} 1} \right\rrbracket\ n\ stk \\
&\qquad = \mathsf{Just}(\mathsf{Left}(z :: stk)) \\
&\left\llbracket \frac{}{rs \vdash uop : 1 \to_{\mathsf{uni}} 1} \right\rrbracket\ n\ (z :: stk) \\
&\qquad = \mathsf{Just}(\mathsf{Left}(\llbracket uop \rrbracket z :: stk)) \\
&\left\llbracket \frac{}{rs \vdash bop : 2 \to_{\mathsf{uni}} 1} \right\rrbracket\ n\ (z_1 :: z_2 :: stk) \\
&\qquad = \mathsf{Just}(\mathsf{Left}(\llbracket bop \rrbracket z_1 z_2 :: stk)) \\
&\left\llbracket \frac{r :: rs \vdash \overset{\pi}{is} : a \to r}{rs \vdash \mathbf{block}_{a \to r}\ is\ \mathbf{end} : a \to_{\mathsf{uni}} r} \right\rrbracket\ n\ stk \\
&\qquad = \mathsf{case}\ \left\llbracket r :: rs \vdash \overset{\pi}{is} : a \to r \right\rrbracket\ n\ stk\ \mathsf{of} \\
&\qquad\quad \mathsf{Nothing} \mapsto \mathsf{Nothing} \\
&\qquad\quad \mathsf{Just}(\mathsf{Left}\ stk') \mapsto \mathsf{Just}(\mathsf{Left}\ stk') \\
&\qquad\quad \mathsf{Just}(\mathsf{Right}(0, stk')) \mapsto \mathsf{Just}(\mathsf{Left}\ stk') \\
&\qquad\quad \mathsf{Just}(\mathsf{Right}(\ell + 1, stk')) \mapsto \mathsf{Just}(\mathsf{Right}(\ell, stk')) \\
&\left\llbracket \pi' \left\{ \frac{a :: rs \vdash \overset{\pi}{is} : a \to r}{rs \vdash \mathbf{loop}_{a \to r}\ is\ \mathbf{end} : a \to_{\mathsf{uni}} r} \right. \right\rrbracket\ n\ stk \\
&\qquad = \mathsf{case}\ \left\llbracket a :: rs \vdash \overset{\pi}{is} : a \to r \right\rrbracket\ n\ stk\ \mathsf{of} \\
&\qquad\quad \mathsf{Nothing} \mapsto \mathsf{Nothing} \\
&\qquad\quad \mathsf{Just}(\mathsf{Left}\ stk') \mapsto \mathsf{Just}(\mathsf{Left}\ stk') \\
&\qquad\quad \mathsf{Just}(\mathsf{Right}(0, stk')) \mapsto \mathsf{if}\ n = 0 \\
&\qquad\qquad \mathsf{then\ Nothing} \\
&\qquad\qquad \mathsf{else}\ \left\llbracket rs \vdash \mathbf{loop}_{a \to r}\ \overset{\pi'}{is}\ \mathbf{end} : a \to_{\mathsf{uni}} r \right\rrbracket\ (n-1)\ stk' \\
&\qquad\quad \mathsf{Just}(\mathsf{Right}(\ell + 1, stk')) \mapsto \mathsf{Just}(\mathsf{Right}(\ell, stk')) \\
&\left\llbracket \frac{}{rs \vdash \mathbf{br_if}\ \ell : r \to_{\mathsf{uni}} 1 + r} \right\rrbracket\ n\ (z :: stk) \\
&\qquad = \mathsf{if}\ z \neq 0\ \mathsf{then\ Just}(\mathsf{Right}(\ell, \mathsf{take}\ (rs\ !!\ \ell)\ stk))\ \mathsf{else\ Just}(\mathsf{Left}\ stk) \\
&\left\llbracket \frac{}{rs \vdash \mathbf{br}\ \ell : r \to_{\mathsf{bi}} 0} \right\rrbracket\ n\ stk \\
&\qquad = \mathsf{Just}(\mathsf{Right}(\ell, \mathsf{take}\ (rs\ !!\ \ell)\ stk)) \\
&\left\llbracket \frac{}{rs \vdash \varepsilon : 0 \to_{\mathsf{uni}} 0} \right\rrbracket\ n\ stk \\
&\qquad = \mathsf{Just}(\mathsf{Left}(stk)) \\
&\left\llbracket \frac{rs \vdash \overset{\pi}{is} : a \to_q m \quad rs \vdash \overset{\pi'}{i} : m \to_{q'} r}{rs \vdash is\ i : a \to_{q \sqcap q'} r} \right\rrbracket\ n\ stk \\
&\qquad = \mathsf{case}\ \left\llbracket rs \vdash is \overset{\pi}{:} a \to_q m \right\rrbracket\ n\ stk\ \mathsf{of} \\
&\qquad\quad \mathsf{Nothing} \mapsto \mathsf{Nothing} \\
&\qquad\quad \mathsf{Just}(\mathsf{Left}\ stk') \mapsto \left\llbracket rs \vdash i \overset{\pi'}{:} m \to_{q'} r \right\rrbracket\ n\ stk' \\
&\qquad\quad \mathsf{Just}(\mathsf{Right}(\ell, stk')) \mapsto \mathsf{Just}(\mathsf{Right}(\ell, stk')) \\
&\left\llbracket \frac{rs \overset{\pi}{\vdash} c : t \quad t <: t'}{rs \vdash c : t'} \right\rrbracket\ n \\
&\qquad = \llbracket t <: t' \rrbracket^{rs} \left(\left\llbracket rs \overset{\pi}{\vdash} c : t \right\rrbracket\ n \right)
\end{aligned}$$

Fig. 6: Typed big-step semantics based on Sub

where the unique t_0 *such that* $\mathsf{infer}\ rs\ c = \mathsf{Just}\ t_0$ *is from Theorem 4 and* $\overset{\pi_0}{rs \vdash c : t_0}$ *is from Theorem 3. Hence any two derivations of* $rs \vdash c : t$ *have the same denotation.*

We also define an untyped semantics, which we relate to the typed semantics to characterize the safety the latter gives. In the untyped semantics, two kinds of runtime errors can occur in addition to exceeding the bound on backjumps: stack underflow and branching outside. The untyped semantics is defined in Figure 7 where we write $\mathsf{StackUnderflow}$, $\mathsf{BranchingOutside}$, and Ok for the three coprojections of the disjoint sum involved.

We define two kinds of type erasure to relate the untyped semantics to the typed semantics. One is an injection from typed initial stacks (fixed-length lists) to untyped initial stacks (arbitrary-length lists). The other is an injection from typed outcomes to untyped outcomes. Let erase_a be the inclusion $\mathbb{Z}_{32}^a \hookrightarrow \mathsf{List}\,\mathbb{Z}_{32}$ and $\mathsf{erase}_{rs,q,r}$ be the inclusion $\mathsf{Maybe}(\mathsf{NT}_q(r) + \sum_{\ell<|rs|} \mathbb{Z}_{32}^{rs!!\ell}) \hookrightarrow \mathsf{Maybe}(\mathsf{List}\ \mathbb{Z}_{32} + \mathbb{N} \times \mathsf{List}\ \mathbb{Z}_{32})$ (which hinges in particular on the inclusion $0 \to \mathsf{List}\ \mathbb{Z}_{32}$ in the case $q = \mathsf{bi}$). For every well-typed instruction sequence, the untyped semantics is identical to the type erasure of the typed semantics as follows.

Theorem 5 (Untyped vs. typed semantics). *If* $rs \vdash c : a \to_q r$*, then*

$$(\!|c|\!)\ rs\ n\ (\mathsf{erase}_a\ stk) = \mathsf{Ok}(\mathsf{erase}_{rs,q,r}\ ([\![rs \vdash c : a \to_q r]\!]\ n\ stk))$$

for all n *and* $stk \in \mathbb{Z}_{32}^a$*.*

Proof. We prove that whenever $a \to_q r <: a' \to_{q'} r'$, we have

$$\begin{aligned} &(\!|c|\!)\ rs\ n\ (\mathsf{erase}_{a'}\ stk) \\ &\quad = \mathsf{Ok}(\mathsf{erase}_{rs,q',r'}([\![a \to_q r <: a' \to_{q'} r']\!]\, rs\, [\![rs \vdash c : a \to_q r]\!]\, n\, stk)) \end{aligned}$$

for all $n \in \mathbb{N}$ and $stk \in \mathbb{Z}_{32}^{a'}$, by induction on the derivation of $rs \vdash c : a \to_q r$. The result follows because $[\![a \to_q r <: a \to_q r]\!]\ rs$ is the identity function.

In particular, Theorem 5 implies no well-typed instruction sequence *is* causes $\mathsf{StackUnderflow}$ or $\mathsf{BranchingOutside}$, assuming the stack has at least a elements.

Graded monad

We further justify the denotational semantics of Sub by noting that underpinning it there is an indexed *graded monad* [?,?,?] (on the category of sets and functions). The indexed graded monad consists of sets of computations, indexed by stack types rs and graded by code types $a \to_q r$, and describes composition of functions from values to computations. It is a graded version of a combination of a state monad (for stack manipulation), an exception monad (for jumps) and the delay monad (to avoid nontermination).

$$(\!|c|\!)\ rs : \mathbb{N} \to \mathsf{List}\,\mathbb{Z}_{32} \to \mathbf{2} + \mathsf{Maybe}\,(\mathsf{List}\,\mathbb{Z}_{32} + (\mathbb{N} \times \mathsf{List}\,\mathbb{Z}_{32}))$$

$$\begin{array}{l}
(\!|\textbf{const}\ z|\!)\ rs\ n\ stk = \mathsf{Ok}(\mathsf{Just}(\mathsf{Left}(z :: stk)))\\
(\!|unop|\!)\ rs\ n\ stk = \mathsf{case}\ stk\ \mathsf{of}\\
\qquad z :: stk' \mapsto \mathsf{Ok}(\mathsf{Just}(\mathsf{Left}((\llbracket unop \rrbracket z) :: stk')))\\
\qquad _ \mapsto \mathsf{StackUnderflow}\\
(\!|binop|\!)\ rs\ n\ stk = \mathsf{case}\ stk\ \mathsf{of}\\
\qquad z_1 :: z_2 :: stk' \mapsto \mathsf{Ok}(\mathsf{Just}(\mathsf{Left}((\llbracket binop \rrbracket z_1 z_2) :: stk')))\\
\qquad _ \mapsto \mathsf{StackUnderflow}\\
(\!|\textbf{block}_{a \to r}\ is\ \textbf{end}|\!)\ rs\ n\ stk = \mathsf{if}\ a > |stk|\ \mathsf{then\ StackUnderflow\ else}\\
\quad \mathsf{let}\ (astk, pstk) = \mathsf{split}\ a\ stk\ \mathsf{in\ case}\ (\!|is|\!)\ (r :: rs)\ n\ astk\ \mathsf{of}\\
\qquad \mathsf{Nothing} \mapsto \mathsf{Ok\ Nothing}\\
\qquad \mathsf{Just}(\mathsf{Left}\ stk') \mapsto \mathsf{Ok}(\mathsf{Just}(\mathsf{Left}(stk' +\!\!+\ pstk)))\\
\qquad \mathsf{Just}(\mathsf{Right}(0, stk')) \mapsto \mathsf{Ok}(\mathsf{Just}(\mathsf{Left}(stk' +\!\!+\ pstk)))\\
\qquad \mathsf{Just}(\mathsf{Right}(\ell + 1, stk')) \mapsto \mathsf{Ok}(\mathsf{Just}(\mathsf{Right}(\ell, stk')))\\
(\!|\textbf{loop}_{a \to r}\ is\ \textbf{end}|\!)\ rs\ n\ stk = \mathsf{if}\ a > |stk|\ \mathsf{then\ StackUnderflow\ else}\\
\quad \mathsf{let}\ (astk, pstk) = \mathsf{split}\ a\ stk\ \mathsf{in\ case}\ (\!|is|\!)\ (a :: rs)\ n\ astk\ \mathsf{of}\\
\qquad \mathsf{Nothing} \mapsto \mathsf{Ok\ Nothing}\\
\qquad \mathsf{Just}(\mathsf{Left}\ stk') \mapsto \mathsf{Ok}(\mathsf{Just}(\mathsf{Left}(stk' +\!\!+\ pstk)))\\
\qquad \mathsf{Just}(\mathsf{Right}(0, stk')) \mapsto\\
\qquad\qquad \mathsf{if}\ n = 0\ \mathsf{then\ Ok\ Nothing\ else}\\
\qquad\qquad\quad (\!|\textbf{loop}_{a \to r}\ is\ \textbf{end}|\!)\ rs\ (n-1)\ (stk' +\!\!+\ pstk)\\
\qquad \mathsf{Just}(\mathsf{Right}(\ell + 1, stk')) \mapsto \mathsf{Ok}(\mathsf{Just}(\mathsf{Right}(\ell, stk')))\\
(\!|\textbf{br}\ \ell|\!)\ rs\ n\ stk = \mathsf{if}\ \ell \geq |rs|\ \mathsf{then\ BranchingOutside\ else}\\
\quad \mathsf{if}\ (rs\ !!\ \ell) > |stk|\ \mathsf{then\ StackUnderflow\ else}\\
\qquad \mathsf{Ok}(\mathsf{Just}(\mathsf{Right}(\ell, \mathsf{take}\ (rs\ !!\ \ell)\ stk)))\\
(\!|\textbf{br_if}\ \ell|\!)\ rs\ n\ stk = \mathsf{case}\ stk\ \mathsf{of}\\
\qquad 0 :: stk' \mapsto \mathsf{Ok}(\mathsf{Just}(\mathsf{Left}\ stk'))\\
\qquad _ :: stk' \mapsto \mathsf{if}\ \ell \geq |rs|\ \mathsf{then\ BranchingOutside\ else}\\
\qquad\qquad\qquad \mathsf{if}\ rs\ !!\ \ell > |stk'|\ \mathsf{then\ StackUnderflow\ else}\\
\qquad\qquad\qquad\quad \mathsf{Ok}(\mathsf{Just}(\mathsf{Right}(\ell, \mathsf{take}\ (rs\ !!\ \ell)\ stk')))\\
\qquad _ \mapsto \mathsf{StackUnderflow}
\end{array}$$

$$\begin{array}{l}
(\!|\varepsilon|\!)\ rs\ n\ stk = \mathsf{Ok}(\mathsf{Just}(\mathsf{Left}\ stk))\\
(\!|is\ i|\!)\ rs\ n\ stk = \mathsf{case}\ (\!|is|\!)\ rs\ n\ stk\ \mathsf{of}\\
\qquad \mathsf{Nothing} \mapsto \mathsf{Ok\ Nothing}\\
\qquad \mathsf{Just}(\mathsf{Left}\ stk') \mapsto (\!|i|\!)\ rs\ n\ stk'\\
\qquad \mathsf{Just}(\mathsf{Right}(\ell, stk')) \mapsto \mathsf{Ok}(\mathsf{Just}(\mathsf{Right}(\ell, stk')))
\end{array}$$

Fig. 7: Untyped big-step semantics

$$T^{rs}_{a \to_q r} X = \mathbb{N} \to \mathbb{Z}^a_{32} \to \mathsf{Maybe}((X \times \mathsf{NT}_q(r)) + \sum_{i < |rs|} \mathbb{Z}^{rs!!i}_{32})$$

$$\eta^{rs}_X : X \to T^{rs}_{0 \to_{\mathsf{uni}} 0} X$$
$$\eta^{rs}_X \; x \; n \; stk = (x, \mathsf{Just}(\mathsf{Left} \; stk))$$

$$\mu^{rs}_{t,t',X} : T^{rs}_t (T^{rs}_{t'} X) \to T^{rs}_{t \oplus t'} X$$
$$\begin{array}{l} \mu^{rs}_{a \to_q m, m' \to_{q'} r, X} \; f \; n \; stk \; = \\ \quad \mathsf{let} \; (astk, pstk) = \mathsf{split} \; a \; stk \; \mathsf{in} \; \mathsf{case} \; f \; n \; astk \; \mathsf{of} \\ \quad\quad \mathsf{Nothing} \mapsto \mathsf{Nothing} \\ \quad\quad \mathsf{Just}(\mathsf{Left}(f', stk')) \mapsto \mathsf{let} \; (astk', pstk') = \mathsf{split} \; m' \; (stk' +\!\!+ \; pstk) \; \mathsf{in} \; \mathsf{case} \; f' \; n \; astk' \; \mathsf{of} \\ \quad\quad\quad \mathsf{Nothing} \mapsto \mathsf{Nothing} \\ \quad\quad\quad \mathsf{Just}(\mathsf{Left}(x', stk'')) \mapsto \mathsf{Just}(\mathsf{Left}(x', stk'' +\!\!+ \; pstk')) \\ \quad\quad\quad \mathsf{Just}(\mathsf{Right}(\ell, stk'')) \mapsto \mathsf{Just}(\mathsf{Right}(\ell, stk'')) \\ \quad\quad \mathsf{Just}(\mathsf{Right}(\ell, stk')) \mapsto \mathsf{Just}(\mathsf{Right}(\ell, stk')) \end{array}$$

$$T^{rs}_{t <: t', X} : T^{rs}_t X \to T^{rs}_{t'} X$$
$$\begin{array}{l} T^{rs}_{a \to_q r <: a+d \to_{q'} r+e, X} \; f \; n \; stk = \\ \quad \mathsf{let} \; (astk, pstk) = \mathsf{split} \; a \; stk \; \mathsf{in} \; \mathsf{case} \; f \; n \; astk \; \mathsf{of} \\ \quad\quad \mathsf{Nothing} \mapsto \mathsf{Nothing} \\ \quad\quad \mathsf{Just}(\mathsf{Left}(x, stk')) \mapsto \mathsf{Just}(\mathsf{Left}(x, stk' +\!\!+ \; pstk)) \\ \quad\quad \mathsf{Just}(\mathsf{Right}(\ell, stk')) \mapsto \mathsf{Just}(\mathsf{Right}(\ell, stk')) \end{array}$$

Fig. 8: Indexed graded monad T

Recall that the set of Sub's code types forms a pomonoid, with order $<:$, unit $0 \to_{\mathsf{uni}} 0$ and multiplication $\oplus$. The pomonoid structure is used in the types of the data of the indexed graded monad that we define in Figure 8. For each context rs, code type $a \to_q r$, and set X, there is a set $T^{rs}_{a \to_q r} X$ of computations that produce values in the set X. The sets $T^{rs}_{a \to_q r} X$ are functorial in X in the obvious way. The unit η_X of the graded monad sends each result $x \in X$ to the computation that immediately returns x, and the multiplication μ_X provides composition of functions from values to computations via flattening of computations of computations into computations. Finally, the coercion functions $T^{rs}_{t <: t'}$ provide subsumption.

This is indeed the structure that we use in the denotational semantics of Sub: the set $[\![a \to_q r]\!]^{rs}$ is just $\mathbf{1} \to T^{rs}_{a \to_q r} \mathbf{1}$, i.e., a special case of a general Kleisli map $X \to T^{rs}_{a \to_q r} Y$, while the denotations of ε, $is \; i$ and subsumptions can be written using the unit, multiplication resp. coercion of the indexed graded monad.

6 Conclusions and future work

We have shown two refinements of the type system of WebAssembly, explained on a minimal fragment of the language that only has the features of interest.

WebAssembly's system has the discrepancy that, while instruction sequences get assigned all valid types (for some definition of validity), instructions other than the exceptional **br** only get assigned their "tightest" (most informative) types. Thus instruction sequences are typed as one would expect from a *declarative* type system, but instructions are typed more in the spirit of a type inference *algorithm*. Our first type system Dir removes this discrepancy: both instructions and instruction sequences get all of their valid types, so Dir is properly declarative, one could say. Our second type system Sub improves on Dir by equipping all instructions and instruction sequences (specifically **br** and instruction sequences containing **br**) with principal types. This is achieved by introducing a code type qualifier to mark what we have here called bivariate stack polymorphism—an unusual form of stack polymorphism that only instructions and instruction sequences that surely fail to terminate normally enjoy.

We have argued that our type system design is systematic. Importantly, qualified code types form a pomonoid, leading to a denotational (functional big-step) semantics based on an indexed graded monad indexed by type contexts and graded by this pomonoid. This systematic design demonstrates, in particular, that the WebAssembly type system maybe considered to be too pedantic about surely non-returning programs. Such programs could be typed as a having a special bottom result type; as a result, more programs would become typable without compromising safety. Our type inference explicitly relies on the presence of specifically marked types for surely non-returning programs. WebAssembly's type system does not record such information in types, but its type-checking algorithm calculates it!

Our semantics shows that WebAssembly, despite being profiled as low-level, is very well suited for big-step reasoning, thanks, of course, to the language having structured control in a form characteristic to high-level languages; small-step reasoning is not necessary. We should also highlight that continuation-passing is not necessary either; direct style is enough, one can use exceptions to describe the semantics of branching. Finally, the semantics is fully compositional also in regards to how the stack is treated: one only ever needs to talk about the local portion of the stack that the instruction or instruction sequence under analysis has access to; there is no need to pass around the global stack and information about which portion is owned by which parent block-like structure.

In future work, we will formally prove that the big-step semantics agrees with the small-step semantics from the specification. The big-step semantics readily suggests a design for a Hoare-style program logic that we will prove sound and complete wrt. the big-step semantics; adequacy for the small-step semantics will then be a corollary. (Cf. the work on a Hoare logic for Wasm by Watt et al. [**?**].) The short distance between big-step semantics and Hoare-style program logics is another good reason to work with big-step reasoning. Finally, we want to study some source-level stack-based program analyses, define them compositionally and show them correct wrt. the big-step semantics. (See for example [**?**].)

Acknowledgements This work was supported by the Icelandic Research Fund grant no. 196323-053.

THIS APPENDIX IS NOT PART OF THE PAPER AND IS PROVIDED ONLY AS ADDITIONAL INFORMATION.

A Some proofs

Proof of Lemma 1. By induction on c (by which we mean mutual induction on c for the two cases i and is of c). We only show some cases. Remember that $d > 0 \vee e > 0$.

1. Cases $i = uop$, $i = bop$: Vacuously true because $rs \vdash_{\mathsf{Dir}} i : a + d \to r$ and $rs \vdash_{\mathsf{Dir}} i : a \to r + e$ never hold at the same time.
2. Case $i = \mathbf{block}_{a' \to r'}\ is\ \mathbf{end}$: Vacuously true because $rs \vdash_{\mathsf{Dir}} \mathbf{block}_{a' \to r'}\ is\ \mathbf{end} : a + d \to r$ and $rs \vdash_{\mathsf{Dir}} \mathbf{block}_{a' \to r'}\ is\ \mathbf{end} : a \to r + e$ never hold at the same time.
3. Case $is = is\ i$: Suppose $rs \vdash_{\mathsf{Dir}} is\ i : a + d \to r$ and $rs \vdash_{\mathsf{Dir}} is\ i : a \to r + e$. From inversion of $\textsc{Seq}_{\mathsf{Dir}}$, we learn that

$$rs \vdash_{\mathsf{Dir}} is : a + d \to m' \tag{1}$$
$$rs \vdash_{\mathsf{Dir}} i : m' \to r \tag{2}$$
$$rs \vdash_{\mathsf{Dir}} is : a \to m \tag{3}$$
$$rs \vdash_{\mathsf{Dir}} i : m \to r + e \tag{4}$$

for some m and m'. We proceed by case analysis on how m and m' compare,
(a) Case $m > m'$: Let $d = m - m' > 0$. We see that $rs \vdash_{\mathsf{Dir}} is : a \to m' + d$ from (3), $rs \vdash_{\mathsf{Dir}} is : a + d \to m'$ (1) , $d > 0$. From the induction hypothesis, we get $rs \vdash_{\mathsf{Sub}} is : a \to_{\mathsf{bi}} m'$. We also get $rs \vdash_{\mathsf{Sub}} i : m' \to_{\mathsf{uni}} r$ from Proposition 1 and (2).
By the rule $\textsc{Seq}_{\mathsf{Sub}}$, we get $rs \vdash_{\mathsf{Sub}} is\ i : a \to_{\mathsf{bi}} r$.
(b) Case $m < m'$: Symmetric.
(c) Case $m = m'$: We consider the cases $d > 0$ or $e > 0$ separately.
i. Case $d > 0$: We see that $rs \vdash_{\mathsf{Dir}} is : a \to m' + 0$ from (3), $rs \vdash_{\mathsf{Dir}} is : a + d \to m'$ (1) and $d > 0$. From the induction hypothesis, we get $rs \vdash_{\mathsf{Sub}} is : a \to_{\mathsf{bi}} m'$. We also get $rs \vdash_{\mathsf{Sub}} i : m' \to_{\mathsf{uni}} r$ from Proposition 1 and (2). By the rule $\textsc{Seq}_{\mathsf{Sub}}$, we get $rs \vdash_{\mathsf{Sub}} is\ i : a \to_{\mathsf{bi}} r$.
ii. Case $e > 0$: Symmetric.

Proof of Theorem 2.
$(\Longrightarrow)$

- Case $q = \mathsf{uni}$: From $rs \vdash_{\mathsf{Sub}} c : a_0 \to_q r_0$ and $a_0 \to_q r_0 <: a \to_{\mathsf{uni}} r$, the rule $\textsc{Subs}_{\mathsf{Sub}}$ yields $rs \vdash_{\mathsf{Sub}} c : a \to r$. From this, $rs \vdash_{\mathsf{Dir}} c : a \to r$ follows by Proposition 1.
- Case $q = \mathsf{bi}$: From $rs \vdash_{\mathsf{Sub}} c : a_0 \to_{\mathsf{bi}} r_0$, the rules $\textsc{SubT}_{\mathsf{bi}}$ and $\textsc{Subs}_{\mathsf{Sub}}$ give $rs \vdash_{\mathsf{Sub}} c : a_0 \to_{\mathsf{uni}} r_0$. We just proved in the previous case that $rs \vdash_{\mathsf{Sub}} c : a_0 \to_{\mathsf{uni}} r_0$ implies $\forall a, r.\ a_0 \to_{\mathsf{uni}} r_0 <: a \to_{\mathsf{uni}} r \implies rs \vdash_{\mathsf{Dir}} c : a \to r$.

$(\Longleftarrow)$

- Case $q = \mathsf{uni}$: Suppose $\forall a, r.\ a_0 \rightarrow_{\mathsf{uni}} r_0 <: a \rightarrow_{\mathsf{uni}} r \implies rs \vdash_{\mathsf{Dir}} c : a \rightarrow r$. Choosing (a_0, r_0) as (a, r), from $a_0 \rightarrow_{\mathsf{uni}} r_0 <: a_0 \rightarrow_{\mathsf{uni}} r_0$ we get $rs \vdash_{\mathsf{Dir}} C : a_0 \rightarrow r_0$. It follows that $rs \vdash_{\mathsf{Sub}} c : a_0 \rightarrow_{\mathsf{uni}} r_0$ from Proposition 1.
- Case $q = \mathsf{bi}$: Suppose $\forall a, r.\ a_0 \rightarrow_{\mathsf{bi}} r_0 <: a \rightarrow_{\mathsf{uni}} r \implies rs \vdash_{\mathsf{Dir}} c : a \rightarrow r$. We get two judgements $rs \vdash_{\mathsf{Dir}} c : a_0 + 1 \rightarrow r_0$ and $rs \vdash_{\mathsf{Dir}} c : a_0 \rightarrow r_0 + 1$ from the assumption by taking $(a_0 + 1, r_0)$ and $(a_0, r_0 + 1)$ for (a, r). By applying Lemma 1 to $d = 1 > 0$ and $e = 1 > 0$, we obtain $rs \vdash_{\mathsf{Sub}} a_0 \rightarrow_{\mathsf{bi}} r_0$.

B An improvement over Sub

The typed big-step semantics of Section 5 hints that there is no need for code types qualified with bi to have a posttype since they type pieces of code that surely fail to terminate normally—as they surely jump.

This suggests that we can improve on Sub by dropping posttypes from bi-types. Indeed, we can work with types $a \rightarrow r$ for pieces of code that may terminate normally and types $a \rightarrow$ for pieces of code that surely do not. The subtyping and typing rules of this improved type system are in Figure 9.

$$\frac{}{a \rightarrow\ <: a + d \rightarrow}\ \textsc{Subt}_{00} \qquad \frac{}{a \rightarrow\ <: a + d \rightarrow r + e}\ \textsc{Subt}_{01}$$

$$\frac{}{a \rightarrow r <: a + d \rightarrow r + d}\ \textsc{Subt}_{1}$$

$$\frac{}{rs \vdash \mathbf{const}\ z : 0 \rightarrow 1}\ \textsc{Const} \qquad \frac{}{rs \vdash uop : 1 \rightarrow 1}\ \textsc{Uop} \qquad \frac{}{rs \vdash bop : 2 \rightarrow 1}\ \textsc{Bop}$$

$$\frac{r :: rs \vdash is : a \rightarrow r}{rs \vdash \mathbf{block}_{a \rightarrow r}\ is\ \mathbf{end} : a \rightarrow r}\ \textsc{Block} \qquad \frac{a :: rs \vdash is : a \rightarrow r}{rs \vdash \mathbf{loop}_{a \rightarrow r}\ is\ \mathbf{end} : a \rightarrow r}\ \textsc{Loop}$$

$$\frac{rs\ !!\ \ell = r}{rs \vdash \mathbf{br\_if}\ \ell : 1 + r \rightarrow r}\ \textsc{Br\_if} \qquad \frac{rs\ !!\ \ell = r}{rs \vdash \mathbf{br}\ \ell : r \rightarrow}\ \textsc{Br}$$

$$\frac{}{rs \vdash \varepsilon : 0 \rightarrow 0}\ \textsc{Empty}$$

$$\frac{rs \vdash is : a \rightarrow \qquad rs \vdash i : m \rightarrow}{rs \vdash is\ i : a \rightarrow}\ \textsc{Seq}_{00} \qquad \frac{rs \vdash is : a \rightarrow \qquad rs \vdash i : m \rightarrow r}{rs \vdash is\ i : a \rightarrow}\ \textsc{Seq}_{01}$$

$$\frac{rs \vdash is : a \rightarrow m \qquad rs \vdash i : m \rightarrow}{rs \vdash is\ i : a \rightarrow}\ \textsc{Seq}_{10} \qquad \frac{rs \vdash is : a \rightarrow m \qquad rs \vdash i : m \rightarrow r}{rs \vdash is\ i : a \rightarrow r}\ \textsc{Seq}_{11}$$

$$\frac{rs \vdash c : t' \qquad t' <: t}{rs \vdash c : t}\ \textsc{Subs}$$

Fig. 9: Subtyping and typing rules of Sub′

Notice that Sub′ types more programs than Sub (and hence Spec). The instruction

$$\mathbf{block}_{0 \rightarrow 0}\ (\mathbf{br}\ 0)\ (\mathbf{const}\ 17)\ \mathbf{end}$$

for instance, is untypable in a context rs in Sub, but typable with principal type $0 \to 0$ in Sub'.

The reason is that, in Sub', although we have $0 :: rs \vdash \mathbf{const}\ 17 : 0 \to 1$, we are entitled to conclude $0 :: rs \vdash (\mathbf{br}\ 0)\ (\mathbf{const}\ 17) : 0 \to$ and further also $0 :: rs \vdash (\mathbf{br}\ 0)\ (\mathbf{const}\ 17) : 0 \to 0$, and hence $rs \vdash \mathbf{block}_{0 \to 0}\ (\mathbf{br}\ 0)\ (\mathbf{const}\ 17)\ \mathbf{end} : 0 \to 0$. In Sub, in contrast, the principal type of $(\mathbf{br}\ 0)\ (\mathbf{const}\ 17)$ in context $0 :: rs$ is $0 \to 1$, which does not subsume $0 \to 0$ and so renders $\mathbf{block}_{0 \to 0}\ (\mathbf{br}\ 0)\ (\mathbf{const}\ 17)\ \mathbf{end}$ untypable in rs.

Similarly to Sub, the type system Sub' enjoys principal types. The operation $\oplus$ of the type inference algorithm that computes the principal type of a sequence from the two given principal types is definable by

$$
\begin{aligned}
(a \to \ ) \oplus (m' \to \ ) &= a \to \\
(a \to \ ) \oplus (m' \to r) &= a \to \\
(a \to m) \oplus (m' \to \ ) &= a + (m' \mathbin{\dot{-}} m) \to \\
(a \to m) \oplus (m' \to r) &= a + (m' \mathbin{\dot{-}} m) \to r + (m \mathbin{\dot{-}} m')
\end{aligned}
$$

Now again, the code types of Sub' with their subtyping relation $<:$, the type $0 \to 0$ and the type operation $\oplus$ form a pomonoid. Moreover, there is an evident pomonoid homomorphism h from the pomonoid of code types of Sub, sending $a \to_{\mathsf{uni}} r$ to $a \to r$ and $a \to_{\mathsf{bi}} r$ to $a \to$. This function h has the properties that $t <: t'$ in Sub implies $h\ t <: h\ t'$ in Sub' and $rs \vdash c : t$ in Sub implies $rs \vdash c : h\ t$ in Sub', i.e., any subtyping or typing derivations in Sub can be translated into Sub'.

The type system Sub' admits a functional-style big-step semantics analogous to Sub in Section 5 and with the same property that the untyped denotations of typed programs agree with their typed denotations (in particular, they don't go wrong). In fact, the semantic functions for subtyping and typing derivations of Sub can be obtained by taking those for subtyping and typing derivations of Sub' and precomposing them with the translations from Sub to Sub'.