




A Type System with Subtyping for WebAssembly’s Stack Polymorphism

Dylan McDermott¹, Yasuaki Morita¹(✉), and Tarmo Uustalu^{1,2}

¹ Dept. of Computer Science, Reykjavik University, Reykjavik, Iceland

² Dept. of Software Science, Tallinn University of Technology, Tallinn, Estonia
{dylanm,yasuaki20,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 one cannot single out a canonical type for a typable instruction sequence satisfactorily. We introduce qualifiers on code types to distinguish between the two flavors of stack polymorphism that occur in WebAssembly 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 stack-manipulation related runtime errors. We have formalized our type system, inference algorithm, and semantics in Agda.

1 Introduction

WebAssembly (Wasm) [10] is a statically typed, stack-oriented bytecode language. Wasm has been designed with a formal semantics [2]. Watt [15] 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 [14], and Huang [3] and Watt et al. [17] 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, an argument type and a result type. 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 first introduce a 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. The two type systems **Dir** and **Sub** have a different status: **Dir** is a minor variant of **Spec**, which we introduce as a first step toward **Sub**; **Sub** is the main type system of our interest. 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 and type 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/wasm-types>.

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.)
$t ::= a \rightarrow r$	code types (called stack types in the spec.)
$\ell \in \mathbb{N}$	label indices
$z \in \mathbb{Z}_{32}$	32-bit integers
$uop ::= \mathbf{eqz} \mid \dots$	unary numeric operations
$bop ::= \mathbf{add} \mid \dots$	binary numeric operations
$i ::= \mathbf{const} \ z \mid uop \mid bop$	numeric instructions
$\mid \mathbf{block}_t \ is \ \mathbf{end} \mid \mathbf{loop}_t \ is \ \mathbf{end}$	block-like instructions
$\mid \mathbf{br_if} \ \ell \mid \mathbf{br} \ \ell$	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 height of the stack. With this simplification, issues such as values of the 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; we have verified that 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 [2,10]. 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 $a \rightarrow r$ 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 $a \rightarrow r$, then, before the jump, r resp. 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 $rs \vdash^1 i : a \rightarrow r$ and $rs \vdash^S is : a \rightarrow r$ where the code type $a \rightarrow r$ describes in both cases 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 types, records the result resp. argument types

$$\begin{array}{c}
\frac{}{rs \vdash^1 \mathbf{const} z : 0 \rightarrow 1} \text{CONST} \quad \frac{}{rs \vdash^1 uop : 1 \rightarrow 1} \text{UOP} \quad \frac{}{rs \vdash^1 bop : 2 \rightarrow 1} \text{BOP} \\
\frac{r :: rs \vdash^S is : a \rightarrow r}{rs \vdash^1 \mathbf{block}_{a \rightarrow r} is \mathbf{end} : a \rightarrow r} \text{BLOCK} \quad \frac{a :: rs \vdash^S is : a \rightarrow r}{rs \vdash^1 \mathbf{loop}_{a \rightarrow r} is \mathbf{end} : a \rightarrow r} \text{LOOP} \\
\frac{rs !! \ell = r}{rs \vdash^1 \mathbf{br.if} \ell : 1 + r \rightarrow r} \text{BR_IF} \quad \frac{rs !! \ell = r}{rs \vdash^1 \mathbf{br} \ell : r + d \rightarrow e} \text{BR} \\
\frac{}{rs \vdash^S \varepsilon : a \rightarrow a} \text{EMPTY} \quad \frac{rs \vdash^S is : a \rightarrow m + d \quad rs \vdash^1 i : m \rightarrow r}{rs \vdash^S is i : a \rightarrow r + d} \text{SEQ}
\end{array}$$

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

of the **block** or **loop** instructions encompassing i or is , in the inside-out order. We write $rs !! \ell$ 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: $a \rightarrow r$ reflects the numbers of arguments and results of the operation, 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 $a \rightarrow a$ for any natural number a . If we take 0 for a , then it becomes $0 \rightarrow 0$. This choice can be called 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 $1 \rightarrow 1$ no longer tells us that the value at the top of the stack remains unchanged. In such a sense, we would say $\varepsilon : 1 \rightarrow 1$ is a reasonable typing but loose in comparison to $\varepsilon : 0 \rightarrow 0$. 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, discussed below).³ 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.

³ We use the term ‘stack polymorphism’ in the sense of Morrisett et al. [6], viz. polymorphism of stack functions in the type of the untouched part of the 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 Spec does not have, but which is semantically justified). First, we think of the type $m + d \rightarrow r + d$ as a loosened version of the type $m \rightarrow r$ of i , although no typing rules allow us to give i this type officially. Since this 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., ε i) are not treated the same way in Spec. For example, **const** 17 as an instruction only has type $0 \rightarrow 1$ in any context, but as an instruction sequence it has the type $d \rightarrow 1 + d$ for any d (since ε admits the type $d \rightarrow d$).

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 [2], 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 not typable when $r = 0$ and is typable when $r \geq 1$, even though the instruction **const** 17 and the end of the **loop** are unreachable:

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

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

$$\begin{array}{c}
\frac{}{rs \vdash \mathbf{const} z : d \rightarrow 1 + d} \text{CONST} \quad \frac{}{rs \vdash uop : 1 + d \rightarrow 1 + d} \text{UOP} \quad \frac{}{rs \vdash bop : 2 + d \rightarrow 1 + d} \text{BOP} \\
\frac{r :: rs \vdash is : a \rightarrow r}{rs \vdash \mathbf{block}_{a \rightarrow r} is \mathbf{end} : a + d \rightarrow r + d} \text{BLOCK} \quad \frac{a :: rs \vdash is : a \rightarrow r}{rs \vdash \mathbf{loop}_{a \rightarrow r} is \mathbf{end} : a + d \rightarrow r + d} \text{LOOP} \\
\frac{rs !! \ell = r}{rs \vdash \mathbf{br_if} \ell : 1 + r + d \rightarrow r + d} \text{BR_IF} \quad \frac{rs !! \ell = r}{rs \vdash \mathbf{br} \ell : r + d \rightarrow e} \text{BR} \\
\frac{}{rs \vdash \varepsilon : r \rightarrow r} \text{EMPTY} \quad \frac{rs \vdash is : a \rightarrow m \quad rs \vdash i : m \rightarrow r}{rs \vdash is i : a \rightarrow r} \text{SEQ}
\end{array}$$

Fig. 3. Typing rules in the type system Dir

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

For an instruction to have a type in Dir, it must be typable also in **Spec**, but the argument and result type may be smaller by some same d . An instruction sequence has exactly the same types in Dir and **Spec**.

Theorem 1 (Dir vs. Spec).

$$\begin{aligned}
rs \vdash_{\text{Dir}} i : a \rightarrow r &\iff (\exists d, a', r'. a = a' + d \wedge r = r' + d \wedge rs \vdash_{\text{Spec}}^1 i : a' \rightarrow r') \\
rs \vdash_{\text{Dir}} is : a \rightarrow r &\iff rs \vdash_{\text{Spec}}^S is : a \rightarrow r
\end{aligned}$$

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

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

$$(\forall d. rs \vdash_{\text{Dir}} i : a + d \rightarrow r + d) \iff rs \vdash_{\text{Spec}}^1 i : a \rightarrow r$$

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

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 satisfactory canonical type among them in all cases. Instructions other

$$\begin{array}{c}
\frac{}{a \rightarrow_{\text{bi}} r <: a + d \rightarrow_q r + e} \text{SUBT}_{\text{bi}} \quad \frac{}{a \rightarrow_{\text{uni}} r <: a + d \rightarrow_{\text{uni}} r + d} \text{SUBT}_{\text{uni}} \\
\\
\frac{}{rs \vdash \mathbf{const} z : 0 \rightarrow_{\text{uni}} 1} \text{CONST} \quad \frac{}{rs \vdash uop : 1 \rightarrow_{\text{uni}} 1} \text{UOP} \quad \frac{}{rs \vdash bop : 2 \rightarrow_{\text{uni}} 1} \text{BOP} \\
\frac{r :: rs \vdash is : a \rightarrow_{\text{uni}} r}{rs \vdash \mathbf{block}_{a \rightarrow r} is \mathbf{end} : a \rightarrow_{\text{uni}} r} \text{BLOCK} \quad \frac{a :: rs \vdash is : a \rightarrow_{\text{uni}} r}{rs \vdash \mathbf{loop}_{a \rightarrow r} is \mathbf{end} : a \rightarrow_{\text{uni}} r} \text{LOOP} \\
\frac{rs \text{ !! } \ell = r}{rs \vdash \mathbf{br.if} \ell : 1 + r \rightarrow_{\text{uni}} r} \text{BR_IF} \quad \frac{rs \text{ !! } \ell = r}{rs \vdash \mathbf{br} \ell : r \rightarrow_{\text{bi}} 0} \text{BR} \\
\\
\frac{}{rs \vdash \varepsilon : 0 \rightarrow_{\text{uni}} 0} \text{EMPTY} \quad \frac{rs \vdash is : a \rightarrow_q m \quad rs \vdash i : m \rightarrow_{q'} r}{rs \vdash is i : a \rightarrow_{q \sqcap q'} r} \text{SEQ} \\
\\
\frac{rs \vdash c : t' \quad t' <: t}{rs \vdash c : t} \text{SUBS}
\end{array}$$

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

than **br** and the empty sequence do have principal types under the (unstated, but conceivable) subtyping relation induced by the inequation $a \rightarrow r <: a + d \rightarrow r + d$, which can be justified by the fact that in **Dir** $rs \vdash c : a \rightarrow r$ implies $rs \vdash c : a + d \rightarrow r + d$ for any piece of code c . But **br** and general instruction sequences (specifically those containing **br**) do not have such principal types. We will now improve on this and introduce a type system **Sub** where even **br** and instruction sequences have principal 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:

$$\overline{\text{bi}} \leq q \quad \overline{\text{uni}} \leq \overline{\text{uni}}$$

In the type system **Sub** code types have the form $a \rightarrow_q r$; 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 the 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 $\text{uni} \sqcap \text{uni} = \text{uni}$ and $q \sqcap q' = \text{bi}$ otherwise. All looseness of typing is introduced by a subsumption rule **SUBS** that applies to both instructions and instruction sequences.

The **uni**-types assigned to a piece of code by **Sub** are precisely the types assigned by **Dir**.

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

$$rs \vdash_{\text{Sub}} c : a \rightarrow_{\text{uni}} r \iff rs \vdash_{\text{Dir}} c : a \rightarrow r$$

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

(\impliedby) By induction on the derivation in $rs \vdash_{\text{Dir}} c : a \rightarrow r$.

To also describe the bi-types of a piece of code in **Sub** in terms of its types in **Dir**, we first show a lemma about bi-types.

Lemma 1.

$$rs \vdash_{\text{Dir}} c : a+d \rightarrow r \wedge rs \vdash_{\text{Dir}} c : a \rightarrow r + e \wedge (d > 0 \vee e > 0) \implies rs \vdash_{\text{Sub}} c : a \rightarrow_{\text{bi}} r$$

Proof. By induction on c .

For a piece of code to acquire a particular type in **Sub**, all of its uni-supertypes must type it in **Dir** (and so also in **Spec** in the case of an instruction sequence).

Theorem 2 (Sub vs. Dir).

$$rs \vdash_{\text{Sub}} c : a_0 \rightarrow_q r_0 \iff (\forall a, r. a_0 \rightarrow_q r_0 <: a \rightarrow_{\text{uni}} r \implies rs \vdash_{\text{Dir}} c : a \rightarrow r)$$

Proof. From [Proposition 1](#) and [Lemma 1](#).

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 that context at all, i.e., type inference 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 algorithm traverses c once, from left to right (depth-first left-first). `MaybeX` is the disjoint sum $1+X$, with coprojections `Nothing` and `Just`. For every instruction, and for the empty sequence, the inferred type is the type from the conclusion of the typing rule. This is not the case for 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 $a \rightarrow_q m$


```

infer c rs : Maybe CodeType

infer (const z) rs = Just(0 →uni 1)
infer uop rs = Just(1 →uni 1)
infer bop rs = Just(2 →uni 1)
infer (blocka→r is end) rs = do tis ← infer is (r :: rs)
                             if tis <: a →uni r then Just(a →uni r) else Nothing
infer (loopa→r is end) rs = do tis ← infer is (a :: rs)
                             if tis <: a →uni r then Just(a →uni r) else Nothing
infer (br.if ℓ) rs = if ℓ < |rs| then Just((1 + rs !! ℓ) →uni (rs !! ℓ)) else Nothing
infer (br ℓ) rs = if ℓ < |rs| then Just((rs !! ℓ) →bi 0) else Nothing

infer ε rs = Just(0 →uni 0)
infer (is :: i) rs = do tis ← infer is rs
                       ti ← infer i rs
                       Just(tis ⊕ ti)

```

Fig. 5. Type inference for Sub

and $m' \rightarrow_{q'} r$ 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 $rs \vdash is : a \rightarrow_{bi} m$, then we can achieve $rs \vdash is : a \rightarrow_{bi} \max(m, m')$, but if we have $rs \vdash is : a \rightarrow_{uni} m$, then we only get $rs \vdash is : a + (\max(m, m') - m) \rightarrow_{uni} \max(m, m')$. As a result of exactly the same thing happening for $rs \vdash i : m' \rightarrow_q r$, the operation \oplus can be defined uniformly in the four cases of q, q' using the “monus” operation $m \dot{-} m' = \max(m, m') - m$ and its qualified version $m \dot{-}_{uni} m' = m \dot{-} m'$, $m \dot{-}_{bi} m' = 0$. We define

$$(a \rightarrow_q m) \oplus (m' \rightarrow_{q'} r) = a + (m' \dot{-}_q m) \rightarrow_{q \sqcap q'} r + (m \dot{-}_{q'} m')$$

The algorithm is sound and complete, i.e., the algorithm infers a type for a piece of code precisely when it is typable, and the inferred type is principal (derivable and a subtype of any other derivable type).

Theorem 3 (Soundness of type inference of Sub).

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

Proof. By induction on c .

Theorem 4 (Completeness of type inference of Sub).

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

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

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.

Pomonoid

The set of code types of `Sub`, together with its subtyping relation $<:$, the element $0 \rightarrow_{\text{uni}} 0$ 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 [9] (see also [13]) and studied earlier in algebra as the *polycyclic monoid* (the inverse envelope of a free monoid) by Nivat and Perrin [8] (modulo the fact that we have replaced lists of value types as stack types by natural numbers, which gives the *bicyclic monoid*).⁵

That we get a pomonoid is very reasonable: it 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)⁶ of simplified `Wasm`.

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

- a natural number as a bound on the number of backjumps that can be made within the loops that this piece of code is encompassed by⁷
- 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.

⁵ In the bicyclic monoid, the partial operation \oplus is made total by adding a special zero element \top , the greatest in the partial order, for ‘possible untypability’.

⁶ For a discussion of the merits of functional-style rather than the usual relational-style big-step semantics in constructive programming theory, see e.g., [7].

⁷ 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 variation of Capretta’s *delay monad* [1].

The semantic function for code types is therefore defined by

$$\llbracket a \rightarrow_q r \rrbracket rs = \mathbb{N} \rightarrow \mathbb{Z}_{32}^a \rightarrow \mathbf{1} + \text{NT}_q(r) + \sum_{\ell < |rs|} \mathbb{Z}_{32}^{rs!!\ell}$$

where $\text{NT}_{\text{bi}}(r) = \mathbf{0}$ and $\text{NT}_{\text{uni}}(r) = \mathbb{Z}_{32}^r$ (NT for “normal termination”); here $\mathbf{0}$ stands for the empty set and \sum for an indexed disjoint sum. We write `Timeout`, `Norm` and `Jump` for the coprojections of the ternary disjoint sum above.⁸

The semantic functions for derivable subtypings and type derivations are defined in [Figure 6](#); the latter is defined by structural recursion on the type derivation. The definitions use auxiliary functions `split a` : $\mathbb{Z}_{32}^{a+d} \rightarrow \mathbb{Z}_{32}^a \times \mathbb{Z}_{32}^d$ that split a given local stack into two parts, with the first part containing the first a elements and the second containing the rest. The function `take a` : $\mathbb{Z}_{32}^{a+d} \rightarrow \mathbb{Z}_{32}^a$ gives only the first part.

The denotation of a derivable subtyping $a \rightarrow_q r <: a + d \rightarrow_{q'} r + e$ is a higher-order function that takes a function f sending any stack of height a to a stack of height r if it terminates normally and returns a function sending a given stack stk of height $a + d$ to a stack of height $r + e$ if f applied to the first a elements of stk terminates normally. It is important to realize that, if $q = \text{bi}$ (and $d \neq e$ in general), then normal termination (the case `Norm stk'`) cannot happen. If $q = \text{uni}$, then the last $d(= e)$ elements of stk that are split off from it before the first a elements are supplied to f are appended back to the result in this case.

Importantly, despite the fact that denotations $\llbracket rs \stackrel{\pi}{\vdash} c : t \rrbracket$ are defined for type derivations (indicated by π) and not just for derivable typing judgements, any two derivations of the same typing judgement $rs \vdash c : t$ still acquire 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 `infer rs c = Just t0` 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*

$$\llbracket rs \stackrel{\pi}{\vdash} c : t \rrbracket = \llbracket t_0 <: t \rrbracket rs \llbracket rs \stackrel{\pi_0}{\vdash} c : t_0 \rrbracket$$

where the unique t_0 such that `infer rs c = Just t0` is from [Theorem 4](#) and $rs \stackrel{\pi_0}{\vdash} c : t_0$ 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

⁸ Notice that $\llbracket a \rightarrow_{\text{bi}} r \rrbracket$ does not really depend on r . This suggests that `bi`-types should perhaps not have a result type at all. Such a design is possible, we look at this in [Section 6](#). The resulting type system accepts more programs but still provides safety.

$$\begin{aligned}
& \llbracket t <: t' \rrbracket rs : \llbracket t \rrbracket rs \rightarrow \llbracket t' \rrbracket rs \\
& \llbracket a \rightarrow_q r <: a + d \rightarrow_{q'} r + e \rrbracket f \ n \ stk = \text{let } (astk, pstk) = \text{split } a \ stk \text{ in case } f \ n \ astk \text{ of} \\
& \quad \text{Timeout} \mapsto \text{Timeout} \\
& \quad \text{Norm } stk' \mapsto \text{Norm}(stk' ++ pstk) \\
& \quad \text{Jump}(\ell, stk') \mapsto \text{Jump}(\ell, stk') \\
& \quad \left[\frac{\pi}{rs \vdash c : t} \right] : \llbracket t \rrbracket rs \\
& \left[\frac{}{rs \vdash \text{const } z : 0 \rightarrow_{\text{uni}} 1} \right] n \ stk = \text{Norm}(z :: stk) \\
& \left[\frac{}{rs \vdash uop : 1 \rightarrow_{\text{uni}} 1} \right] n \ (z :: stk) = \text{Norm}(\llbracket uop \rrbracket z :: stk) \\
& \left[\frac{}{rs \vdash bop : 2 \rightarrow_{\text{uni}} 1} \right] n \ (z_2 :: z_1 :: stk) = \text{Norm}(\llbracket bop \rrbracket z_1 z_2 :: stk) \\
& \left[\frac{\pi}{rs \vdash \text{block}_{a \rightarrow r} \ is \ \text{end} : a \rightarrow_{\text{uni}} r} \right] n \ stk = \text{case} \left[r :: rs \vdash \frac{\pi}{is : a \rightarrow r} \right] n \ stk \text{ of} \\
& \quad \text{Timeout} \mapsto \text{Timeout} \\
& \quad \text{Norm } stk' \mapsto \text{Norm } stk' \\
& \quad \text{Jump}(0, stk') \mapsto \text{Norm } stk' \\
& \quad \text{Jump}(\ell + 1, stk') \mapsto \text{Jump}(\ell, stk') \\
& \left[\frac{\pi' \left\{ a :: rs \vdash \frac{\pi}{is : a \rightarrow r} \right\}}{rs \vdash \text{loop}_{a \rightarrow r} \ is \ \text{end} : a \rightarrow_{\text{uni}} r} \right] n \ stk = \text{case} \left[a :: rs \vdash \frac{\pi}{is : a \rightarrow r} \right] n \ stk \text{ of} \\
& \quad \text{Timeout} \mapsto \text{Timeout} \\
& \quad \text{Norm } stk' \mapsto \text{Norm } stk' \\
& \quad \text{Jump}(0, stk') \mapsto \text{if } n = 0 \text{ then Timeout else } \left[\frac{\pi'}{rs \vdash \text{loop}_{a \rightarrow r} \ is \ \text{end} : a \rightarrow_{\text{uni}} r} \right] (n - 1) \ stk' \\
& \quad \text{Jump}(\ell + 1, stk') \mapsto \text{Jump}(\ell, stk') \\
& \left[\frac{}{rs \vdash \text{br } \ell : r \rightarrow_{\text{bi}} 0} \right] n \ stk = \text{Jump}(\ell, \text{take } (rs !! \ell) \ stk) \\
& \left[\frac{}{rs \vdash \text{br.if } \ell : r \rightarrow_{\text{uni}} 1 + r} \right] n \ (z :: stk) = \text{if } z \neq 0 \text{ then } \text{Jump}(\ell, \text{take } (rs !! \ell) \ stk) \text{ else Norm } stk \\
& \left[\frac{}{rs \vdash \varepsilon : 0 \rightarrow_{\text{uni}} 0} \right] n \ stk = \text{Norm } stk \\
& \left[\frac{\pi \quad \pi'}{rs \vdash is : a \rightarrow_q m \quad rs \vdash i : m \rightarrow_{q'} r} \right] n \ stk = \text{case} \left[rs \vdash is : a \rightarrow_q m \right] n \ stk \text{ of} \\
& \quad \text{Timeout} \mapsto \text{Timeout} \\
& \quad \text{Norm } stk' \mapsto \left[rs \vdash i : m \rightarrow_{q'} r \right] n \ stk' \\
& \quad \text{Jump}(\ell, stk') \mapsto \text{Jump}(\ell, stk') \\
& \left[\frac{\pi}{rs \vdash c : t \quad t <: t'} \right] n = \llbracket t <: t' \rrbracket^{rs} (\left[rs \vdash c : t \right] n)
\end{aligned}$$

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

of runtime errors can occur in addition to exceeding the bound on backjumps: jumps too far out and stack underflow. The untyped semantics is defined in [Figure 7](#) where we write `JumpOutside`, `StackUnderflow` and `Ok` for the coprojections of the outer ternary disjoint sum and `Timeout`, `Norm` and `Jump` for those of the inner one.

We define two kinds of type erasure to relate the untyped semantics to the typed semantics. One is an injection from typed initial stacks (specific-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 $\mathbb{Z}_{32}^a \hookrightarrow \text{List } \mathbb{Z}_{32}$ and $\text{erase}_{rs,q,r}$ be the inclusion $1 + \text{NT}_q(r) + \sum_{\ell < |rs|} \mathbb{Z}_{32}^{rs!!\ell} \hookrightarrow 1 + \text{List } \mathbb{Z}_{32} + \mathbb{N} \times \text{List } \mathbb{Z}_{32}$ (which hinges in particular on the inclusion $\mathbf{0} \hookrightarrow \text{List } \mathbb{Z}_{32}$ in the case $q = \text{bi}$). For every well-typed instruction sequence, the untyped denotation is identical to the type erasure of the typed denotation.

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

$$\llbracket c \rrbracket rs \ n \ (\text{erase}_a \ stk) = \text{Ok}(\text{erase}_{rs,q,r}(\llbracket rs \vdash c : a \rightarrow_q r \rrbracket \ n \ stk))$$

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

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

$$\begin{aligned} \llbracket c \rrbracket rs \ n \ (\text{erase}_{a'} \ stk) \\ = \text{Ok}(\text{erase}_{rs,q',r'}(\llbracket a \rightarrow_q r <: a' \rightarrow_{q'} r' \rrbracket \ rs \ \llbracket rs \vdash c : a \rightarrow_q r \rrbracket \ n \ stk)) \end{aligned}$$

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

In particular, [Theorem 5](#) implies that no well-typed piece of code c can cause `StackUnderflow` or `JumpOutside` when run on a good initial stack stk .

Graded Monad

We further justify the denotational semantics of `Sub` by noting that underpinning it there is an indexed *graded monad* [\[11,5,4\]](#) (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 $a \rightarrow_q r$, 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).

Recall that the set of `Sub`’s code types forms a pomonoid, with order $<:$, unit $0 \rightarrow_{\text{uni}} 0$ 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 $a \rightarrow_q r$, and set X , there is a set $T_{a \rightarrow_q r}^{rs} X$ of computations that produce values in the set X . The sets $T_{a \rightarrow_q r}^{rs} X$ 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

$$\begin{aligned}
& \llbracket c \rrbracket rs : \mathbb{N} \rightarrow \text{List } \mathbb{Z}_{32} \rightarrow 1 + 1 + (1 + \text{List } \mathbb{Z}_{32} + \mathbb{N} \times \text{List } \mathbb{Z}_{32}) \\
& \llbracket \text{const } z \rrbracket rs \ n \ stk = \text{Ok}(\text{Norm}(z :: stk)) \\
& \llbracket uop \rrbracket rs \ n \ stk = \text{case } stk \text{ of} \\
& \quad z :: stk' \mapsto \text{Ok}(\text{Norm}(\llbracket uop \rrbracket z :: stk')) \\
& \quad _ \mapsto \text{StackUnderflow} \\
& \llbracket bop \rrbracket rs \ n \ stk = \text{case } stk \text{ of} \\
& \quad z_2 :: z_1 :: stk' \mapsto \text{Ok}(\text{Norm}(\llbracket bop \rrbracket z_1 z_2 :: stk')) \\
& \quad _ \mapsto \text{StackUnderflow} \\
& \llbracket \text{block}_{a \rightarrow r} \text{ is end} \rrbracket rs \ n \ stk = \text{if } a > |stk| \text{ then StackUnderflow else} \\
& \quad \text{let } (astk, pstk) = \text{split } a \ stk \text{ in case } \llbracket is \rrbracket (r :: rs) \ n \ astk \text{ of} \\
& \quad \quad \text{Timeout} \mapsto \text{Ok Timeout} \\
& \quad \quad \text{Norm } stk' \mapsto \text{Ok}(\text{Norm}(stk' ++ pstk)) \\
& \quad \quad \text{Jump}(0, stk') \mapsto \text{Ok}(\text{Norm}(stk' ++ pstk)) \\
& \quad \quad \text{Jump}(\ell + 1, stk') \mapsto \text{Ok}(\text{Jump}(\ell, stk')) \\
& \llbracket \text{loop}_{a \rightarrow r} \text{ is end} \rrbracket rs \ n \ stk = \text{if } a > |stk| \text{ then StackUnderflow else} \\
& \quad \text{let } (astk, pstk) = \text{split } a \ stk \text{ in case } \llbracket is \rrbracket (a :: rs) \ n \ astk \text{ of} \\
& \quad \quad \text{Timeout} \mapsto \text{Ok Timeout} \\
& \quad \quad \text{Norm } stk' \mapsto \text{Ok}(\text{Norm}(stk' ++ pstk)) \\
& \quad \quad \text{Jump}(0, stk') \mapsto \text{if } n = 0 \text{ then Ok Timeout else} \\
& \quad \quad \quad \llbracket \text{loop}_{a \rightarrow r} \text{ is end} \rrbracket rs \ (n - 1) \ (stk' ++ pstk) \\
& \quad \quad \text{Jump}(\ell + 1, stk') \mapsto \text{Ok}(\text{Jump}(\ell, stk')) \\
& \llbracket \text{br } \ell \rrbracket rs \ n \ stk = \text{if } \ell \geq |rs| \text{ then JumpOutside else} \\
& \quad \text{if } rs \ \! \! \ell > |stk| \text{ then StackUnderflow else} \\
& \quad \quad \text{Ok}(\text{Jump}(\ell, \text{take } (rs \ \! \! \ell) \ stk)) \\
& \llbracket \text{br.if } \ell \rrbracket rs \ n \ stk = \text{case } stk \text{ of} \\
& \quad 0 :: stk' \mapsto \text{Ok}(\text{Norm } stk') \\
& \quad _ :: stk' \mapsto \text{if } \ell \geq |rs| \text{ then JumpOutside else} \\
& \quad \quad \text{if } rs \ \! \! \ell > |stk'| \text{ then StackUnderflow else} \\
& \quad \quad \quad \text{Ok}(\text{Jump}(\ell, \text{take } (rs \ \! \! \ell) \ stk')) \\
& \quad _ \mapsto \text{StackUnderflow} \\
& \llbracket \varepsilon \rrbracket rs \ n \ stk = \text{Ok}(\text{Norm } stk) \\
& \llbracket is \ i \rrbracket rs \ n \ stk = \text{case } \llbracket is \rrbracket rs \ n \ stk \text{ of} \\
& \quad \text{Timeout} \mapsto \text{Ok Timeout} \\
& \quad \text{Norm } stk' \mapsto \llbracket i \rrbracket rs \ n \ stk' \\
& \quad \text{Jump}(\ell, stk') \mapsto \text{Ok}(\text{Jump}(\ell, stk'))
\end{aligned}$$

Fig. 7. Untyped big-step semantics

$$\begin{aligned}
T_{a \rightarrow_q r}^{rs} X &= \mathbb{N} \rightarrow \mathbb{Z}_{32}^a \rightarrow 1 + X \times \text{NT}_q(r) + \sum_{i < |rs|} \mathbb{Z}_{32}^{rs!!i} \\
\eta_X^{rs} : X &\rightarrow T_{0 \rightarrow_{\text{uni}0}}^{rs} X \\
\eta_X^{rs} x \ n \ stk &= (x, \text{Norm } stk) \\
\mu_{t, t', X}^{rs} : T_t^{rs} (T_{t'}^{rs} X) &\rightarrow T_{t \oplus t'}^{rs} X \\
\mu_{a \rightarrow_q m, m' \rightarrow_q r, X}^{rs} f \ n \ stk &= \text{let } (astk, pstk) = \text{split } a \ stk \text{ in case } f \ n \ astk \text{ of} \\
&\quad \text{Timeout} \mapsto \text{Timeout} \\
&\quad \text{Norm}(f', stk') \mapsto \text{let } (astk', pstk') = \text{split } m' (stk' ++ pstk) \text{ in case } f' \ n \ astk' \text{ of} \\
&\quad \quad \text{Timeout} \mapsto \text{Timeout} \\
&\quad \quad \text{Norm}(x', stk'') \mapsto \text{Norm}(x', stk'' ++ pstk') \\
&\quad \quad \text{Jump}(\ell, stk'') \mapsto \text{Jump}(\ell, stk'') \\
&\quad \text{Jump}(\ell, stk') \mapsto \text{Jump}(\ell, stk') \\
T_{t <: t', X}^{rs} : T_t^{rs} X &\rightarrow T_{t'}^{rs} X \\
T_{a \rightarrow_q r <: a+d \rightarrow_q r+e, X}^{rs} f \ n \ stk &= \text{let } (astk, pstk) = \text{split } a \ stk \text{ in case } f \ n \ astk \text{ of} \\
&\quad \text{Timeout} \mapsto \text{Timeout} \\
&\quad \text{Norm}(x, stk') \mapsto \text{Norm}(x, stk' ++ pstk) \\
&\quad \text{Jump}(\ell, stk') \mapsto \text{Jump}(\ell, stk')
\end{aligned}$$

Fig. 8. Indexed graded monad T

provides composition of functions from values to computations via flattening of computations of computations into computations. Finally, the coercion functions $T_{t <: t'}^{rs}$ provide subsumption.

This is indeed the structure that we use in the denotational semantics of **Sub**: the set $\llbracket a \rightarrow_q r \rrbracket^{rs}$ is just $1 \rightarrow T_{a \rightarrow_q r}^{rs} 1$, i.e., a special case of a general Kleisli map $X \rightarrow T_{a \rightarrow_q r}^{rs} Y$, while the denotations of ε , is i and subsumptions can be written using the unit, multiplication resp. coercion of the indexed graded monad.

6 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 result type 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 result types 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.

Notice that **Sub'** types more programs than **Sub** (and hence **Spec**). The instruction **block** $_{0 \rightarrow 0}$ (**br** 0) (**const** 17) **end**, for instance, is untypable in **Sub**, but typable with principal type $0 \rightarrow 0$ in **Sub'**.

Similarly to **Sub**, the type system **Sub'** enjoys principal types, with the principal type of a sequence given by an operation \oplus' .

$$\begin{array}{c}
\frac{}{a \rightarrow <: a + d \rightarrow} \text{SUBT}_{00} \quad \frac{}{a \rightarrow <: a + d \rightarrow r + e} \text{SUBT}_{01} \quad \frac{}{a \rightarrow r <: a + d \rightarrow r + d} \text{SUBT}_1 \\
\frac{}{rs \vdash \mathbf{const} z : 0 \rightarrow 1} \text{CONST} \quad \frac{}{rs \vdash \mathbf{uop} : 1 \rightarrow 1} \text{UOP} \quad \frac{}{rs \vdash \mathbf{bop} : 2 \rightarrow 1} \text{BOP} \\
\frac{r :: rs \vdash is : a \rightarrow r}{rs \vdash \mathbf{block}_{a \rightarrow r} is \mathbf{end} : a \rightarrow r} \text{BLOCK} \quad \frac{a :: rs \vdash is : a \rightarrow r}{rs \vdash \mathbf{loop}_{a \rightarrow r} is \mathbf{end} : a \rightarrow r} \text{LOOP} \\
\frac{rs !! \ell = r}{rs \vdash \mathbf{br.if} \ell : 1 + r \rightarrow r} \text{BR_IF} \quad \frac{rs !! \ell = r}{rs \vdash \mathbf{br} \ell : r \rightarrow} \text{BR} \quad \frac{}{rs \vdash \varepsilon : 0 \rightarrow 0} \text{EMPTY} \\
\frac{rs \vdash is : a \rightarrow \quad rs \vdash i : m \rightarrow}{rs \vdash is i : a \rightarrow} \text{SEQ}_{00} \quad \frac{rs \vdash is : a \rightarrow \quad rs \vdash i : m \rightarrow r}{rs \vdash is i : a \rightarrow} \text{SEQ}_{01} \\
\frac{rs \vdash is : a \rightarrow m \quad rs \vdash i : m \rightarrow}{rs \vdash is i : a \rightarrow} \text{SEQ}_{10} \quad \frac{rs \vdash is : a \rightarrow m \quad rs \vdash i : m \rightarrow r}{rs \vdash is i : a \rightarrow r} \text{SEQ}_{11} \\
\frac{rs \vdash c : t' \quad t' <: t}{rs \vdash c : t} \text{SUBS}
\end{array}$$

Fig. 9. Subtyping and typing rules of Sub'

The code types of Sub' with their subtyping relation $<:$, the type $0 \rightarrow 0$ and the type operation \oplus' again form a pomonoid.⁹ Moreover, there is an evident pomonoid homomorphism h from the pomonoid of code types of Sub , sending $a \rightarrow_{\text{uni}} r$ to $a \rightarrow r$ and $a \rightarrow_{\text{bi}} r$ to $a \rightarrow$. 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., the subtyping and typing derivations in Sub translate 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 do not 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' .

7 Conclusions and Future Work

We have shown two refinements of the type system of Wasm, explained on a minimal fragment of the language that only has the features of interest. Wasm's type system has the discrepancy that, while instruction sequences get assigned all valid types (for some definition of validity), instructions other than the exceptional \mathbf{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

⁹ But with lists instead of natural numbers as stack types, normal associativity of \oplus' is lost, as $((a_0 \rightarrow) \oplus' (a \rightarrow m)) \oplus' (m' \rightarrow r) = (a_0 \rightarrow)$ while $(a \rightarrow m) \oplus' (m' \rightarrow r)$ is undefined when neither m nor m' is a prefix of the other. Totalizing \oplus' with a zero greatest element \top gives a skew pomonoid: associativity holds as an inequation.

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 specifically 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 design demonstrates, in particular, that the Wasm type system may be considered to be too pedantic about surely non-returning programs. Such programs could be typed as a having no result type; then more programs would become typable without compromising safety, cf. system `Sub'` in Section 6. The systems `Dir` and `Sub` are (on purpose) conservative over the type system of the Wasm specification in that they type exactly the same programs. `Sub` has principal types because it has specifically marked types for surely non-returning programs. The type system of the specification does not record such information in types, but its type-checking algorithm calculates it nonetheless.¹⁰

Our semantics shows that Wasm, 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. [16].) 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 [12].)

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

¹⁰ One could argue that all of this is splitting hairs over typing code fragments that can be seen to be unreachable by a very simple analysis and that a good compiler from a higher-level language to Wasm should not produce this kind of unreachable code. The latter might be true, but a type system for Wasm must still handle all Wasm programs, in particular also programs containing this kind of unreachable code, in some adequate way, unless we declare these programs syntactically ill-formed.

References

1. Chapman, J., Uustalu, T., Veltri, N.: Quotienting the delay monad by weak bisimilarity. *Math. Struct. Comput. Sci.* **29**(1), 67–92 (2019). <https://doi.org/10.1017/s0960129517000184>
2. Haas, A., Rossberg, A., Schuff, D.L., Titzer, B.L., Holman, M., Gohman, D., Wagner, L., Zakai, A., Bastien, J.F.: Bringing the Web up to speed with WebAssembly. In: *Proc. of 38th ACM SIGPLAN Conf. on Programming Language Design and Implementation, PLDI 2017*, pp. 185–200. ACM Press, New York (2017). <https://doi.org/10.1145/3062341.3062363>
3. Huang, X.: A Mechanized Formalization of the WebAssembly Specification in Coq. Master’s thesis, Rochester Inst. of Technol. (2019), https://www.cs.rit.edu/~mtf/student-resources/20191_huang_mscourse.pdf
4. Katsumata, S.: Parametric effect monads and semantics of effect systems. In: *Proc. of 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL ’14*, pp. 633–645. ACM Press, New York (2014). <https://doi.org/10.1145/2535838.2535846>
5. Melliès, P.A.: Parametric monads and enriched adjunctions. Manuscript (2012), <https://www.irif.fr/~mellies/tensorial-logic/8-parametric-monads-and-enriched-adjunctions.pdf>
6. Morrisett, G., Crary, K., Glew, N., Walker, D.: Stack-based typed assembly language. *J. Funct. Program.* **13**(5), 957–959 (2003). <https://doi.org/10.1017/s0956796801004178>
7. Nakata, K., Uustalu, T.: Trace-based coinductive operational semantics for while: big-step and small-step, relational and functional styles. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) *TPHOLs 2009*, LNCS, vol. 5674, pp. 375–390. Springer, Berlin, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_26
8. Nivat, M., Perrot, J.F.: Une généralisation du monoïde bicyclique. *Comptes Rendus Acad. Sci. Paris, Ser. A* **271**, 824–827 (1970), <https://gallica.bnf.fr/ark:/12148/bpt6k480299v/f830>
9. Pöial, J.: Algebraic specification of stack-effects for Forth-programs. In: *1990 FORML Conf. Proc.*, pp. 282–290. Forth Interest Group (1991), https://www.kodu.ee/~jpoial/teadus/EuroForth90_Algebraic.pdf
10. Rossberg, A.: WebAssembly core specification, version 1.1. Editor’s Draft, 18 Dec. 2021 (2021), <https://webassembly.github.io/spec/core/>
11. Smirnov, A.: Graded monads and rings of polynomials. *J. Math. Sci.* **151**(3), 3032–3051 (2008). <https://doi.org/10.1007/s10958-008-9013-7>
12. Stiévenart, Q., De Roover, C.: Compositional information flow analysis for WebAssembly programs. In: *Proc. of IEEE 20th Int. Working Conf. on Source Code Analysis and Manipulation, SCAM ’20*, pp. 13–24. IEEE, Los Alamitos, CA (2020). <https://doi.org/10.1109/scam51674.2020.00007>
13. Stoddart, B., Knaggs, P.J.: Type inference in stack based languages. *Formal Aspects Comput.* **5**(4), 289–298 (1993). <https://doi.org/10.1007/bf01212404>
14. W3C: WebAssembly core specification, version 1.0. W3C Recommendation, 5 Dec. 2019 (2019), <https://www.w3.org/TR/wasm-core-1>
15. Watt, C.: Mechanising and verifying the WebAssembly specification. In: *Proc. of 7th ACM SIGPLAN Int. Conf. on Certified Programs and Proofs, CPP ’18*, pp. 53–65. ACM Press, New York (2018). <https://doi.org/10.1145/3167082>
16. Watt, C., Maksimovic, P., Krishnaswami, N.R., Gardner, P.: A program logic for first-order encapsulated WebAssembly. In: Donaldson, A.F. (ed.) *Proc. of 33rd Europ. Conf. on Object-Oriented Programming, ECOOP 2019*, Leibniz Int. Proc. in

Inform., vol. 134, pp. 9:1–9:30. Dagstuhl Publishing, Saarbrücken/Wadern (2019).
<https://doi.org/10.4230/lipics.ecoop.2019.9>

17. Watt, C., Rao, X., Pichon-Pharabod, J., Bodin, M., Gardner, P.: Two mechanisations of WebAssembly 1.0. In: Huisman, M., Pasareanu, C.S., Zhan, N. (eds.) FM 2021, LNCS, vol. 13047, pp. 61–79. Springer, Cham (2021).
https://doi.org/10.1007/978-3-030-90870-6_4