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 with subtyping relation and qualifiers over shapes of operand stack for WebAssembly. Our type system has two kinds of qualifiers, and we show that the type inference algorithm gives principal types even for programs with stack polymorphic properties. We show that the new type system is in a precise relationship to the type system given in the WebAssembly specification. In addition, we give a denotational semantics based on the new type system.

Introduction

WebAssembly (Wasm) is a type-safe, portable, low-level programming language. Wasm is a stack-oriented language, and its type system reflects shapes of operand stacks. The Wasm specification [1] gives a formal type system for Wasm programs. In general, this type system does not give an instruction sequence a unique type but allows it to have different valid types. In particular, the typing of unreachable code, which is characterized by the term "stack polymorphic" in the Wasm specification, expands the range of valid types while retaining subtle restrictions. We recapitulate the range of possible valid types by two type qualifiers which we call *uni* and *bi*, and give a type system based on subtyping with these qualifiers. We prove our type system to be equivalent to the one in the specification. We give a type inference algorithm for the new type system and prove that the algorithm is sound and infers the principal type for a given program. We also provide a denotational semantics based on the new type system.

Wasm's stack polymorphism

A notable feature of programs in Wasm is the hierarchical block structure and the control flow that follows the structure. The block instructions block and loop have a structured syntax where an inner instruction sequence is followed by the keyword end. When a block is entered, a label is created that can be used as a branch target. At branching via a br or brif instruction, as many arguments are taken from the operand stack as the target label requires and execution continues according to the label. If control reaches the end of the block, the label created by the block is discarded and execution of subsequent instructions outside the block continues. Each label has a type taken from the type annotation of the bloc or loop.

The type system of the specification uses a mixed style where some constructs are given all valid types while others are only given "principal" types (although there is no formal notion of subtyping or type schemes with instances). Instructions (except br and some similar instructions such as unreachable, br_table and return) are only given principal types, while sequences of instructions are given all valid types. The typing rules for the empty sequence and sequencing incorporate the necessary amount of "subsumption" for constituent instructions.

The typing rule for the br instruction gives it all valid types, not a principal type—because it has no principal type in the type language of the specification. This is called "stack polymorphism" in the specification. This is somewhat misleading; what is meant is that br gets many monomorphic types.

A new type system based on subtyping with *uni* and *bi*

We extend Wasm's type language of function types with qualifiers *uni* and *bi* and subtyping. In Wasm, a function type is a pair of argument and result stack shapes where a stack shape is a list of value types. In this presentation, we make the simplifying assumption that there is exactly one value type, then stack shapes become just natural numbers. If an instruction is given a type $rs \vdash a \to_{uni} r$, then it means its evaluation takes a local stack of shape $a + d$ to a local stack of shape $r + d$ for any d (where by the local stack we mean the portion of the stack belonging to the current frame) unless it jumps to label ℓ in which case the resulting local stack must be of shape $rs \mathbin{!!} \ell + g$ for some g. The point is that there may be a passive part d of the local stack that is untouched. If an instruction is given a type $rs \vdash a \to_{bi} r$, then it means it can take a local stack of shape $a + d$ to a local stack of shape $r + e$ for any d, e unless it jumps. Here the passive parts of the stack before and after are completely unrelated.

A few examples of principal typing with *uni* and *bi* qualifiers are shown below. In the typing derivation (1), which derives the principal type of $\mathsf{brif}\ \ell; \mathsf{const}\ z$, it is used that $\mathsf{brif}\ \ell$ has type $1+(rs!!\ell) \to_{uni} rs!!\ell$, which is its principal type. This is because it pops a value from the operand stack, and then, if the value is true, jumps to ℓ-level outer block taking $rs \mathbin{!!} \ell$ from operand stack and, otherwise, continues with the subsequent instruction sequence of the current block without consuming $rs \mathbin{!!} \ell$. Similarly, the principal type of the instruction $\mathsf{const}\ z$ is $0 \to_{uni} 1$ because it does not consume the operand stack at all and produces a value on the top. To make the composition $\mathsf{brif}\ \ell; \mathsf{const}\ z$ well-typed, $\mathsf{const}\ z$ is cast to the type $rs \mathbin{!!} \ell \to_{uni} 1 + (rs \mathbin{!!} \ell)$ where subsumption is performed in the *uni* mode—adding the same amount $rs \mathbin{!!} \ell$ to both the argument type and result type.

$$\frac{rs \vdash \mathsf{brif}\ \ell : 1 + (rs \mathbin{!!} \ell) \to_{uni} rs \mathbin{!!} \ell \quad \dfrac{rs \vdash \mathsf{const}\ z : 0 \to_{uni} 1}{rs \vdash \mathsf{const}\ z : rs \mathbin{!!} \ell \to_{uni} 1 + (rs \mathbin{!!} i)}}{rs \vdash \mathsf{brif}\ \ell; \mathsf{const}\ z : 1 + (rs \mathbin{!!} \ell) \to_{uni} 1 + (rs \mathbin{!!} \ell)} \tag{1}$$

In the type derivation (2), the $\mathsf{br}\ \ell$ instruction gets a principal type $rs \mathbin{!!} \ell \to_{bi} 0$. The control jumps to the outside of the block taking the argument corresponding to the ℓ-th label unconditionally. This means subsequent instructions after $\mathsf{br}\ \ell$ in the current block will never be executed. There is no impact on runtime safety if we suppose $\mathsf{br}\ \ell$ produces any number of values on the operand stack on the current block. Subsumption of $\mathsf{br}\ \ell$ to $rs \mathbin{!!} \ell \to_{bi} 2$ is valid because it has the *bi* quantifier. Even though the composition $\mathsf{br}\ \ell; \mathsf{add}$ never terminates normally, its type is subtly constrained—the result type is at least 1. The Wasm typing discipline is that even unreachable code has to be well-typed.

$$\frac{\dfrac{rs \vdash \mathsf{br}\ \ell : rs \mathbin{!!} \ell \to_{bi} 0}{rs \vdash \mathsf{br}\ \ell : rs \mathbin{!!} \ell \to_{bi} 2} \quad rs \vdash \mathsf{add} : 2 \to_{uni} 1}{rs \vdash \mathsf{br}\ \ell; \mathsf{add} : rs \mathbin{!!} \ell \to_{bi} 1} \tag{2}$$

Differently from the type system of the specification, our type system assigns to both instructions and sequences of instructions all valid types, but a separate type inference algorithm finds principal types. For any instruction or code admitting a type all, the principal type always exist and it is the smallest one among its valid types.

We have fully formalized our type system (for a core fragment of Wasm), principal type inference with its soundness and completeness, and the denotational semantics in Agda. We have also formalized the type system of the specification (for the same fragment) and its precise relationship to our type system.

Denotational semantics with an indexed graded monad

Our type language essentially corresponds to the partially ordered monoid underlying a certain family of graded monads. The monads in this family are graded by types of programs (which are qualified function types, i.e., argument and result type pairs). The family is indexed by typing contexts (which consists of the result types required by jump targets).

Each graded monad in the family is a combination of a state monad (to account for manipulation of the store and the stack) with an exceptions monad (to account for the possibility of terminating with a jump) and a version of the delay monad (to account for the possibility of nontermination from loops; a computation takes a natural number to control unwinding of loops and may fail).

We provide a denotational semantics using this indexed graded monad.

The denotational semantics of instructions and pieces of code is terms of Kleisli maps of the graded monad corresponding to the type context of current environment. If $rs \vdash is : a \to_q r$, then $[\![is]\!] : 1 \to T^{rs}_{a \to_q r} 1$ where

$$T^{r_0,\ldots,r_{n-1}}_{a \to_q r} = \mathbb{N} \to \textstyle\prod_{(a',r')} a \to_q r \leq a' \to_{uni} r' \to \mathbf{St}\, a' \to \mathbf{Maybe}\,(\mathbf{St}\, r' + \coprod_{\ell=0}^{n-1} \mathbf{St}\, r_\ell)$$

where a state is a triple of a store, a local stack of the appropriate shape (height) and a downcounter, $\mathbf{St}\, r = \mathbf{Sto} \times \mathbf{Stk}\, r$. In particular, the empty sequence of instructions is interpreted by the unit and sequential composition of an instruction and a sequence of instructions by the multiplication. The casts that interpret the subtyping relation correspond to functoriality of the graded monad in the grade argument.

In its treatment of jumps and the stack, our denotational semantics bears similarities to the big-step semantics of Watt et al. [2], but is systematically structured around the indexed graded monad.

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

References

[1] A. Rossberg, ed. WebAssembly Specification, Release 1.1 (draft 2021-10-01). https://webassembly.github.io/spec/core/

[2] C. Watt, P. Maksimovic, N. Krishnaswami, P. Gardner. A program logic for first-Order encapsulated WebAssembly. In A. F. Donaldson, ed., *Proc. of 33rd Europ. Conf. on Object-Oriented Programming, ECOOP 2019*, v. 134 of *Leibniz Int. Proc. in Inform.*, art. 9. Dagstuhl Publishing, 2019.