An Operational Semantics for Yul

Vasileios Koutavas[1,3], Yu-Yang Lin[1], and Nikos Tzevelekos[2](✉)

[1] Trinity College Dublin, Dublin, Ireland {Vasileios.Koutavas,linhouy}@tcd.ie
[2] Queen Mary University of London, London, UK nikos.tzevelekos@qmul.ac.uk
[3] Lero - Science Foundation Ireland Research Centre for Software, Limerick, Ireland

Abstract. We present a big-step and small-step operational semantics for Yul — the intermediate language used by the Solidity compiler to produce EVM bytecode — in a mathematical notation that is congruous with the literature of programming languages, lends itself to language proofs, and can serve as a precise, widely accessible specification for the language. Our two semantics stay faithful to the original, informal specification of the language but also clarify under-specified cases such as void function calls. Our presentation allows us to prove the equivalence between the two semantics. We also implement the small-step semantics in an interpreter for Yul which avails of optimisations that are provably correct. We have tested the interpreter using tests from the Solidity compiler and our own. We envisage that this work will enable the development of verification and symbolic execution technology directly in Yul, contributing to the Ethereum security ecosystem, as well as aid the development of a provably sound future type system.

Keywords: Yul, Ethereum, Operational semantics, Programming languages, Formal methods

1 Introduction

Smart contracts are programs stored on a blockchain, executed within transactions, which read and write data on said blockchain. They form an underpinning technology for decentralised finance applications by enabling the programming of autonomous services offering lending, auction, new cryptocurrency creation, and virtually any type of transactional facilities. On Ethereum—the largest blockchain with the ability to execute them—smart contracts are typically written in a high-level programming language, predominantly Solidity, and then compiled down to Ethereum Virtual Machine (EVM) instructions. Smart contracts already control significant amounts of cryptocurrency[4], making them security-critical software. Numerous documented exploits and bugs have resulted in the loss of billions of dollars worth of cryptocurrency for users [1, 15, 37], making smart contract correctness and security a problem of critical importance, especially since smart contracts are immutable once deployed on the blockchain.

[4] As of 24 June 2024, DefiLlama (https://defillama.com/chain/Ethereum) lists a Total Value Locked (TVL) of USD 59.937bn, equal to ETH 17.31m, on Ethereum.

The Solidity compiler team recently introduced Yul, an intermediate representation (IR) language sitting between Solidity and EVM bytecode, with one of its stated goals being to help with the implementation of formal verification and optimisation techniques [38]. Although such tools for Ethereum smart contracts exist, they either work with Solidity (e.g. SOLHINT [9], SLITHER [8,19], ECHIDNA [5,22], GAMBIT [6]) or EVM bytecode (e.g. MYTHRIL [7,36], OYENTE [2,30], MANTICORE [4,33]), making them hard to develop and maintain because of the feature-rich, often evolving syntax of the former, or the low-level stack- and jump-based design of the latter. Thus, Yul is designed on a small number of high-level constructs—loops, functions, conditional/switch and local variables—avoiding stack and jump complexity, and includes flexible, dialect-specific built-ins and data types.[5] All modern Solidity code can be compiled to Yul through the Solidity compiler, and almost all existing EVM bytecode can be decompiled to it [21].

In order for Yul to serve its intended purpose, it needs to have precise formal semantics on which formal verification techniques can be developed and optimisations can be proven correct. The official documentation of Yul [38][6] only provides an informal description of the language, its grammar, scoping rules, and its operational semantics in terms of an evaluation function written in pseudocode. Although sufficient to understand the intent of the language constructs, the documentation is not a rigorous enough specification of Yul on which to base formal proofs. As an example, the operational semantics in the documentation does not explicitly handle the case for calling functions with no return variables, which must be treated as statements rather than expressions (cf. Sec. 3.5 and Rem. 10).

A number of mechanisations of the intended semantics of Yul in formal tools have appeared online. These include a shallow embedding of Yul into Dafny [11,12], and deep embeddings into the K-framework [18], Isabelle/HOL [3], Lean [34], and ACL2 [25]. Although these contribute significantly towards the goal of giving a solid mathematical footing for the semantics of Yul, they have inherent shortcomings. Besides not being peer-reviewed, their main drawback is that in order to understand these mechanisations, one is required to have working knowledge of the formal tools used, something that, although increasingly more common, is not universal among those who have received training in programming languages. Moreover, the tools used lend themselves more readily to mechanically formalising the details of various semantics rather than communicating them to experts. For example, the mechanisations of Yul contain explicit implementation of mathematical details that are conceptually straightforward (e.g. pattern-matching and manipulation of low-level dependencies such as sets, mappings, lists, etc), making them too lengthy and complex to serve as a high-level model of the language. Moreover, formalisation tools often incur engineering considerations that arise from their specific framework or underlying theories, which may require reshaping the presentation into a less standard form. Finally, to even write a

[5] Although currently Yul is used only with EVM primitives, it is designed to be parametric to a *dialect* specifying the data types and machine instructions.

[6] Current latest version and the one accessed is v0.8.26 of the Solidity compiler

mechanisation of the Yul semantics one must first have a conceptual mathematical model of the language, which to our knowledge does not yet publicly exist. All this makes the existing Yul mechanisations hard to understand or trust as being equivalent to each other, which limits their suitability as definitive presentations of the semantics of Yul. For these reasons, we believe that the development of a diverse tool ecosystem for Yul would benefit from a more abstract, widely understandable standard mathematical model thereof.

In this paper we contribute to this direction by providing two mathematical semantics of the language in textbook notation that, staying faithful to the intent of the original informal specification, can serve as a useful, widely understood, precise reference document. We hope our work will aid future formal verification research and implementations for Yul, contribute to the evolution of the language—such as when the community decides to introduce a type system and prove its soundness—and assist in validating existing mechanisations by providing a more abstract model to compare them to.

We present here an abstract syntax of Yul (Sec. 2), together with both a big-step and small-step operational semantics (Sec. 3), and prove their correspondence (Sec. 4). We choose to present both styles of semantics because a big-step semantics is typically more intuitive and human-readable, and more readily relates to the informal specification of Yul, while a small-step semantics is easier to implement, especially in interpreters and symbolic execution tools. Note that we present only the semantics for Yul, which is given separately from the semantics of EVM bytecode and dialect-specific notation such as objects. As mentioned earlier, this is because Yul is intended to be parametric to the specific dialect implemented. We also present a modular implementation of the semantics of Yul in the form of an interpreter that is parametric on a given dialect and avails of optimisations that are provably correct (Sec. 5). At the moment, this interpreter includes a subset of the semantics of the Shanghai upgrade of EVM as its only dialect, which is sufficient to evaluate the correctness of our semantics experimentally. As such, our prototype interpreter only executes *closed* programs that make use of a subset of the instructions available to the EVM. We evaluate our implementation with sample implementations of standard algorithms and tests from the Solidity compiler. Finally, we discuss related work (Sec. 6) and conclusions (Sec. 7). Our work has been motivated by our ongoing efforts to implement a complete symbolic execution engine of EVM-dialect Yul, which would not have been possible without the semantics presented herein.

2 Yul Syntax

Yul is designed to be a highly readable language with simple syntax and semantics that is as direct to transform into bytecode as possible. Yul achieves these goals by providing high-level syntax that replaces EVM's (the primary target of the language) low-level jumps for control flow, and explicit stack operations for variable management.

We define the *source-level* abstract syntax of Yul; that is, the syntax of Yul programs, in Fig. 1. The grammar in this figure ranges over (dialect-specific) values or constants with u, v, c, variables/identifiers with x, y, z, statements with

$$\begin{array}{lll} \textsf{Stmt:} & S ::= \{S^*\} \mid \textsf{function}\, x\,(\vec{x}) \to \vec{x}\,\{S^*\} \mid \textsf{let}\ \vec{x} := M \mid \vec{x} := M \mid M \mid \textsf{if}\ M\{S^*\} \\ & \quad \mid \textsf{switch}\ M\,(\textsf{case}\ v\{S^*\})^*\textsf{default}\{S^*\} \mid \textsf{for}\{S^*\}M\{S^*\}\{S^*\} \\ & \quad \mid \textsf{break} \mid \textsf{continue} \mid \textsf{leave} \\ \textsf{Exp:} & M ::= x\,(\vec{M}) \mid op\,(\vec{M}) \mid x \mid v \qquad \textsf{Val:}\ u, v, c \qquad \textsf{Var:}\ x, y, z, f \end{array}$$

Fig. 1. Yul source-level syntax.

S and expressions with M, and variants thereof. We use vector syntax (e.g. $\vec{x}$) to mean a comma-separated, consecutively-indexed list of syntax $(x_1, x_2, \ldots, x_n)$, and the Kleene star (e.g. S^*) to mean consecutively-indexed juxtaposition of syntax without commas $(S_1 S_2 \ldots S_m)$. We next explain the syntactical constructs and their intention.

– **Block statement:** $\{S^*\}$. These are sequences of statements that are to be executed one after the other unless a halting statement is encountered.

– **Function definition:** $\textsf{function}\, x\,(\vec{x}_a) \to \vec{x}_r\{S^*\}$.

These contain the function name x, vectors of argument, $\vec{x}_a$, and return, $\vec{x}_r$, variables, and a function body block $\{S^*\}$. Functions can return zero (hereafter referred to as void functions), one, or a tuple of values (multi-value functions). Blocks are leveraged to define the scope of variables in a function call. Actual arguments are assigned to $\vec{x}_a$ when entering, and return values are those assigned to $\vec{x}_r$ when exiting, the function block. Nested function declarations are allowed by Yul's specification [38].

– **Variable declaration/assignment:** $\textsf{let}\ \vec{x} := M$ / $\vec{x} := M$. Their behaviour is identical except for the conditions involved: $\vec{x}$ cannot be in scope if declared with a let-statement and must be in scope for standard assignment. The syntax allows for multi-value assignment, with M being an expression which must match the arity of $\vec{x}$. As tuples are not part of the source-level syntax, they can only occur as a result of calling a multi-value function. Tuples do not appear in Fig. 1 but will be introduced in the runtime syntax in Sec. 3. The Yul specification [38] allows for uninitialised declarations. For simplicity we consider these to be syntactic sugar for function calls returning the correct arity of zeros.

– **Conditional statement:** $\textsf{if}\ M\{S^*\}$. M must evaluate to a constant that represents either tt or ff, and $\{S^*\}$ is a block for scoping reasons; there are no else-branches in conditionals.

– **Switch statement:** $\textsf{switch}\, M(\textsf{case}\, v\{S^*\})^*\textsf{default}\{S_d^*\}$. If M evaluates to constant v_i then block $\{S_i^*\}$ executes, otherwise $\{S_d^*\}$ does.

– **Loop:** $\textsf{for}\{S_i^*\}M\{S_p^*\}\{S_b^*\}$. Loops comprise an initialisation block $\{S_i^*\}$, a conditional expression M which evaluates to tt or ff at each iteration, a post-iteration block $\{S_p^*\}$ for updating loop variables, and a loop body $\{S_b^*\}$.

– **Halting statement:** break, continue, leave. These are control-flow statements; break and continue may only occur in the body of the innermost loop that contains them, and outside of any function defined in that loop body. They are used to exit (break) said loop or skip the rest of the loop body and continue to

the post-iteration block (continue). Statement leave only occurs inside function bodies and exits the current function.

– **Function call expression:** $x(\vec{M})$. A function call is normally a part of an expression and returns a single value. However there are two special cases of note: (1) a function call can appear as the expression of a multi-value declaration/assignment and must return a tuple of values of the correct arity; (2) a void function call that appears at the place of a statement, in which case it must return regular (i.e. similar to void or unit values in other languages—see Sec. 3); this is the only allowed expression that can appear at the place of a statement. Arguments are evaluated *right-to-left*.

– **Opcode execution expression:** $op\ (\vec{M})$. These are calls to dialect-specific instructions external to Yul. Like function calls, arguments are evaluated right-to-left and return zero, one, or multiple values. Unlike function calls, opcodes may return exceptional values which form part of the dialect, not named by the Yul specification. We will treat them as abnormal program exit values in our small-step semantics in the following section.

Example 1. To illustrate the source-level language, consider a naive recursive implementation of a Fibonacci term generator. Note that in Yul double forward slashes are comments.

```
{ // Function that computes Fibonacci via naive recursion
  function fibonacci_rec(n) -> result {
    if lt(n,3) {
        result := 1
    }
    if gt(n,2) {
        result := add(fibonacci_rec(sub(n,1)),fibonacci_rec(sub(n,2)))
    }
  } // Call Fibonacci with 10
  let fib_10 := fibonacci_rec(10)
  mstore(0x00, fib_10)
}
```

In the program above, the function `fibonacci_rec` computes the n-th Fibonacci number, where `lt`/`gt`, `add`/`sub` and `mstore` are comparison, arithmetic and memory instructions respectively. These are opcodes in the EVM dialect of Yul. Functions do not require a return statement. Instead, the defined output variable `result` is assigned and its value returned at the end. Nesting function calls within opcode/function calls is allowed, and is only constrained by the stack limit of the underlying dialect (the EVM stack limit is 1024 words). The program computes the 10th Fibonacci term and stores it at position `0x00` in EVM memory. ■

Remark 2 (Typing). Although simple checks are in place in the Solidity compiler, such as for the arity of function arguments/returns, Yul at the moment provides no formal specification for a type system. Its default dialect involves only EVM bytecode operations with a single value type of 256-bit unsigned integers (U256). While the Yul source language involves literals of various kinds (such as hex literals), there is an implicit assumption that these are appropriately translated into values of the underlying dialect—e.g. U256 for the EVM dialect.

Remark 3 (Dialects). As mentioned in the Introduction, because Yul is meant to be target-independent, it is designed to be modular with regards to multiple so-called dialects. Each dialect in Yul would in principle be defined in relation to the specific back-end bytecode that it is meant to compile into. For this purpose, dialects amount to a collection of machine opcodes that remain after excluding those associated with control flow and variable management, and the respective value types used by said opcodes. Thus, while we use tt and ff values in our semantics, these do not necessarily represent boolean values; instead they are placeholders for whichever values are considered true and false in a given dialect. For instance, in the EVM dialect, zero represents ff while non-zero values are considered tt. This is a conceptually simple mapping that entails a variety of subtle behaviours one has to be careful about in the EVM (e.g. negating a "true" value does not make it a "false" one, as negation is bit-wise and could produce another non-zero value).

3 Yul Semantics

In this section we present big- and small-step semantics for Yul. We present them together, grouped thematically under rules for blocks, variable manipulation, branching statements, loops, and expressions. For small-step semantics specifically, we additionally define top-level rules that carry around the evaluation contexts and handle external opcode instructions. We prove the correspondence of the two semantics in Sec. 4. For both semantics, we shall use configurations of the form:

$$\langle S \mid G\,; L\,; \mathcal{N} \rangle$$

with the following components:

– **Statement** S (or *term*) being evaluated. Statements are drawn from an extended *operational syntax* defined in Fig. 2 (top), and discussed further below.

– **Global environment** G that is external to the semantics of Yul itself, and in the context of smart contracts is meant to contain the state of the blockchain, the data of external calls to a smart contract, etc.; G will be passed as state by the semantics of Yul and may change only by calls to dialect opcodes.

– **Local state** L of all variables. This is a dictionary $L : x \mapsto v$. All variables in scope for the term being evaluated will be in L.

– **Namespace** $\mathcal{N}$ of all functions visible to the term being evaluated. This is like a function repository $\mathcal{N} : x \mapsto (\vec{x}_a, \vec{x}_r, S_b)$ for all the functions in scope.

Next, we describe the the extended syntax for statements which adds regular, representing the successful completion of a statement, and the following statements that arise from certain small-step reductions (cf. Def. 6):

– **Block scoping:** $\{\vec{S}\}^{\mathcal{N}}_{L}$. This is used to implement the scoping rules for blocks and are distinct from source-level blocks $\{S^*\}$. Here $\mathcal{N}$ records functions and L variables in scope before entering a block, needed when reinstating this scope after exiting the block.

– **Break/continue catching:** $[\![S]\!]_{\mathsf{brk}}$ / $[\![S]\!]_{\mathsf{cnt}}$. These catch break and continue statements respectively within loop body blocks.

$$\begin{array}{rrcl}
\textsf{Stmt:} & S & ::= & \ldots \mid \textsf{regular} \mid \mathcal{M} \mid \{\vec{S}\}_L^{\mathcal{N}} \mid [\![S]\!]_{\textsf{brk}} \mid [\![S]\!]_{\textsf{cnt}} \mid (\!|S|\!)_L^{\vec{x}} \\
\textsf{Exp:} & M & ::= & \ldots \mid <\vec{v}> \\
\textsf{Mode:} & \mathbb{M} & ::= & \textsf{break} \mid \textsf{continue} \mid \textsf{leave} \mid \textsf{regular} \mid \mathcal{M} \\
\textsf{ExMode:} & \mathcal{M} & & \\
\end{array}$$

$$\begin{array}{rl}
\textsf{SCxt:} & E ::= [\cdot] \mid \{E, \vec{S}\}_L^{\mathcal{N}} \mid \textsf{let } \vec{x} := E \mid \vec{x} := E \mid E_{\textsf{Exp}} \mid [\![E]\!]_{\textsf{cnt}} \mid [\![E]\!]_{\textsf{brk}} \mid \textsf{if } E_{\textsf{Exp}} \{S^*\} \\
& \quad \mid \textsf{switch } E_{\textsf{Exp}} (\textsf{case } v \{S^*\})^* \textsf{default} \{S^*\} \\
\textsf{ECxt:} & E_{\textsf{Exp}} ::= [\cdot] \mid x(\vec{M}, E_{\textsf{Exp}}, \vec{v}) \mid op(\vec{M}, E_{\textsf{Exp}}, \vec{v}) \mid (\!|E|\!)_L^{\vec{x}}
\end{array}$$

Fig. 2. Yul operational syntax (omitting constructs shown in Fig. 1).

– **Function call framing:** $(\!|S|\!)_L^{\vec{x}}$. These encode function call frames by remembering the L at call site and the return variables $\vec{x}$. At return, L is used to restrict the domain and ensure correct scoping (similar to blocks) and $\vec{x}$ is used to retrieve the function return values. Function calls do not need to reinstate $\mathcal{N}$; this is done indirectly via the block of the function body (see Sec. 3.5 and Ex. 9).

We also extend expressions M with tuples $<\vec{v}>$, needed for multi-value assignments and function returns, which are defined when $|\vec{v}| > 1$. Finally, we introduce *modes* $\mathbb{M}$, as in the Yul specification, which can signal regular termination (regular), e.g. of a void function call, break/continue modes for for-loops, and leave mode for returning early from functions. Modes also include *external irregular modes* $\mathcal{M}$ that may be returned by opcode evaluation. These are parametric to the Yul semantics and depend on the specific dialect implemented.

In Fig. 2 (bottom) we define statement (SCxt) and expression (ECxt) evaluation contexts for our small-step semantics. Most evaluation contexts are derived in a standard way from the syntax of the language and implement correct evaluation order in small-step semantics. For example block statements are evaluated left-to-right, whereas function and opcode call arguments are evaluated right-to-left. The inclusion of function call contexts $(\!|E|\!)_L^{\vec{x}}$ encodes a call stack in our small-step semantics. As functions have a fresh variable scope when they are called, the surrounding scope L of the caller must be recorded and reinstated at function return. The return variables must also be recorded to be returned to the caller.

In the following subsections we present the rules of the big- and small-step semantics of Yul, grouped by the kind of term they handle. These rules precisely define the predicates of the next two definitions.

Definition 4 (Big-step semantics). *We define a big-step semantics on successful executions of statements that do not result in external irregular modes:*

$$\langle S \mid G ; L ; \mathcal{N} \rangle \Downarrow \langle \mathbb{M} \mid G' ; L' ; \mathcal{N}' \rangle$$

We additionally define variants thereof for the evaluation of:

- *sequences of statements within blocks,* $\langle \vec{S} \mid G ; L ; \mathcal{N} \rangle \Downarrow_{\textsf{seq}} \langle \mathbb{M} \mid G' ; L' ; \mathcal{N}' \rangle$
- *expressions,* $\langle M \mid G ; L ; \mathcal{N} \rangle \Downarrow_{\textsf{exp}} \langle S' \mid G' ; L' ; \mathcal{N}' \rangle$, *where* $S' \in \{v, <\vec{v}>, \textsf{regular}\}$

The above are concretely defined by the top rules in Fig. 3 to Fig. 7. We also assume a dialect-specific reduction relation for opcodes denoted by $\Downarrow_{\textsf{opc}}$.

Remark 5. Although it would have been possible to define big-step rules to deal with propagation of external irregular modes, we follow standard practice and define $\Downarrow$ only on successful computation, excluding irregular modes. We do handle these modes in the small-step semantics, which more closely specifies an interpreter for the language.

Definition 6 (Small-step semantics). *We write*

$$\langle S \mid G; L\,; \mathcal{N}\rangle \to \langle S' \mid G'; L'\,; \mathcal{N}'\rangle$$

for a small-step transition in our small-step semantics. When a transition does not change G (as only opcode execution does) we omit G from the configurations. The reduction rules are shown in the bottom part of Fig. 3 to Fig. 7, with the additional top-level rules:

(1) $\langle E[S] \mid G; L\,; \mathcal{N}\rangle \to \langle E[S'] \mid G'; L'\,; \mathcal{N}'\rangle$ *if* $\langle S \mid G; L\,; \mathcal{N}\rangle \to \langle S' \mid G'; L'\,; \mathcal{N}'\rangle$
(2) $\langle E[\mathcal{M}] \mid G; L\,; \mathcal{N}\rangle \to \langle \mathcal{M} \mid G; L\,; \mathcal{N}\rangle$

The top-level rules decompose the term to an evaluation context with an inner statement in its hole that can either: (1) perform a Yul reduction, or (2) is an irregular return mode that terminates the program. We next present the rules for small- and big-step semantics for each type of term.

3.1 Block Statements

The first rules are those for blocks, shown in Fig. 3. Blocks sequentially execute statements and define a local namespace for variables and functions. The big-step semantics handles function scoping in the Block rule, where the existing function namespace $\mathcal{N}$ is extended by the functions defined in the block (given by $\mathcal{N}_1 = \mathsf{funs}(\{S_1..S_n\})$), and block statements are evaluated under $\mathcal{N} \uplus \mathcal{N}_1$. Moreover, any new variables defined within the scope (discussed in the following subsection), are pruned in the resulting configuration of the rule by $L_1 \restriction L$, which restricts the variable environment L_1 to only the variables in L. Rules SeqEmpty, SeqR (regular case) and SeqI (irregular case) handle the sequencing of statement execution with the last one skipping all remaining statements in a sequence if a Yul mode other than regular is encountered, which causes early exit from a loop iteration, an entire loop, or a function body. Function declarations are ignored by rule FunDecl, as these are put in scope by the Block rule.

On the small-step side, we make use of a context $\{\ldots\}^{\mathcal{N}}_{L}$ that remembers the variable and function namespaces to handle scoping. This context is introduced when a block is entered and removed (with the appropriate scope restriction) when exiting the block. Note that, in this semantics and the Yul specification, functions do not need to be declared before being called, as long as they are in the same, or surrounding, block of the call.

Remark 7 (Block optimisation). It is sound to simplify any $\{\ldots\{\{\cdot\}^{\mathcal{N}}_{L_1}\}^{\mathcal{N}}_{L_2}\ldots\}^{\mathcal{N}}_{L_n}$ to $\{\cdot\}^{\mathcal{N}}_{L}$, provided $\mathsf{dom}(L_1) = \ldots = \mathsf{dom}(L_n)$, avoiding redundant block-exit reductions. However, this may not provide a performance gain as calculating equality of domains can be costlier than the additional reductions.

$$\frac{\mathcal{N}_1 = \mathsf{funs}(\{S_1..S_n\}) \qquad \langle S_1,..,S_n \mid G\,;L\,;\mathcal{N}\uplus\mathcal{N}_1\rangle \Downarrow_{\mathsf{seq}} \langle \mathbb{M} \mid G_1\,;L_1\,;\mathcal{N}\uplus\mathcal{N}_1\rangle}{\langle \{S_1..S_n\} \mid G\,;L\,;\mathcal{N}\rangle \Downarrow \langle \mathbb{M} \mid G_1\,;L_1{\restriction}L\,;\mathcal{N}\rangle}\ \textsc{Block}$$

$$\frac{}{\langle \varepsilon \mid G\,;L\,;\mathcal{N}\rangle \Downarrow_{\mathsf{seq}} \langle \mathsf{regular} \mid G\,;L\,;\mathcal{N}\rangle}\ \textsc{SeqEmpty}$$

$$\frac{\langle S_1 \mid G\,;L\,;\mathcal{N}\rangle \Downarrow \langle \mathsf{regular} \mid G_1\,;L_1\,;\mathcal{N}\rangle \quad \langle \vec{S} \mid G_1\,;L_1\,;\mathcal{N}\rangle \Downarrow_{\mathsf{seq}} \langle \mathbb{M} \mid G_2\,;L_2\,;\mathcal{N}\rangle}{\langle S_1,\vec{S} \mid G\,;L\,;\mathcal{N}\rangle \Downarrow_{\mathsf{seq}} \langle \mathbb{M} \mid G_2\,;L_2\,;\mathcal{N}\rangle}\ \textsc{SeqR}$$

$$\frac{\langle S_1 \mid G\,;L\,;\mathcal{N}\rangle \Downarrow \langle \mathbb{M} \mid G_1\,;L_1\,;\mathcal{N}\rangle \qquad \mathbb{M} \in \{\mathsf{break},\mathsf{continue},\mathsf{leave}\}}{\langle S_1,\vec{S} \mid G\,;L\,;\mathcal{N}\rangle \Downarrow_{\mathsf{seq}} \langle \mathbb{M} \mid G_1\,;L_1\,;\mathcal{N}\rangle}\ \textsc{SeqI}$$

$$\frac{}{\langle \mathsf{function}\,x\,(\vec{y})\to\vec{z}\,S \mid G\,;L\,;\mathcal{N}\rangle \Downarrow \langle \mathsf{regular} \mid G\,;L\,;\mathcal{N}\rangle}\ \textsc{FunDecl}$$

$BS\textsc{tart} : \langle \{S_1..S_n\} \mid L\,;\mathcal{N}\rangle \to \langle \{S_1,..,S_n\}^{\mathcal{N}}_{L} \mid L\,;\mathcal{N}\uplus\mathcal{N}_1\rangle$ if $\mathcal{N}_1 = \mathsf{funs}(\{S_1..S_n\}), n>0$

$BE\textsc{mpty} : \langle \{\} \mid L\,;\mathcal{N}\rangle \to \langle \mathsf{regular} \mid L\,;\mathcal{N}\rangle$

$SC\textsc{nt} : \langle \{\mathsf{regular},\vec{S}\}^{\mathcal{N}}_{L} \mid L_1\,;\mathcal{N}_1\rangle \to \langle \{\vec{S}\}^{\mathcal{N}}_{L} \mid L_1\,;\mathcal{N}_1\rangle$ if $\vec{S} \neq \epsilon$

$SR\textsc{eg} : \langle \{\mathsf{regular}\}^{\mathcal{N}}_{L} \mid L_1\,;\mathcal{N}_1\rangle \to \langle \mathsf{regular} \mid L_1{\restriction}L\,;\mathcal{N}\rangle$

$SI\textsc{rreg} : \langle \{\mathbb{M},\vec{S}\}^{\mathcal{N}}_{L} \mid L_1\,;\mathcal{N}_1\rangle \to \langle \mathbb{M} \mid L_1{\restriction}L\,;\mathcal{N}\rangle$ if $\mathbb{M}\in\{\mathsf{break},\mathsf{continue},\mathsf{leave}\}$

$FD\textsc{ef} : \langle \mathsf{function}\,x\,(\vec{y})\,\vec{z}\,S \mid L\,;\mathcal{N}\rangle \to \langle \mathsf{regular} \mid L\,;\mathcal{N}\rangle$

Fig. 3. Big-step (top) and small-step (bottom) block semantics. Note that G is omitted in small-step rules that leave it unchanged (cf. Def. 6).

3.2 Variable Manipulation Statements

Single- and multi-value variable declaration are handled by the rules in Fig. 4. Assignment rules are similar, with the only differences being that the statement is missing the let-keyword, and the side condition in all rules requires $x \in \mathsf{dom}(L)$ (or $\vec{x} \in \mathsf{dom}(L)$) instead of $x \notin \mathsf{dom}(L)$ (resp., $\vec{x} \notin \mathsf{dom}(L)$). These rules behave as expected: they add or update L bindings. Unlike functions, variables must be declared before use. Only function and opcode calls may return multiple values. Although unrestricted in the rules, $\Downarrow_{\mathsf{exp}}$ does not change L.

3.3 Branching Statements

Control branching in Yul is done by conditional and switch statements. Conditionals make use of tt and ff, which are not necessarily boolean values (as Yul is parametric in the dialect types) but, instead, placeholders for values that are to be interpreted correspondingly. We assume a dialect comes equipped with predicates that let us identify which values are tt and ff.

The rules for these statements are given in Fig. 5. On the big-step side, we have standard IfFalse and IfTrue rules for conditionals. Switch statements evaluate expression M to a value v; if v matches c_i from the listed cases, the statement of that case is evaluated (SwC), otherwise the default case statement is

$$\frac{\langle M \mid G\,;L\,;\mathcal{N}\rangle \Downarrow_{\mathsf{exp}} \langle v \mid G_1\,;L_1\,;\mathcal{N}\rangle \qquad x \notin \mathsf{dom}(L)}{\langle \mathsf{let}\ x\,\mathtt{:=}M \mid G\,;L\,;\mathcal{N}\rangle \Downarrow \langle \mathsf{regular} \mid G_1\,;L_1[x \mapsto v]\,;\mathcal{N}\rangle}\ \textsc{VarDecl}$$

$$\frac{\langle M \mid G\,;L\,;\mathcal{N}\rangle \Downarrow_{\mathsf{exp}} \langle <\vec{v}> \mid G_1\,;L_1\,;\mathcal{N}\rangle \qquad \vec{x} \notin \mathsf{dom}(L)}{\langle \mathsf{let}\ \vec{x}\,\mathtt{:=}M \mid G\,;L\,;\mathcal{N}\rangle \Downarrow \langle \mathsf{regular} \mid G_1\,;L_1[\vec{x} \mapsto \vec{v}]\,;\mathcal{N}\rangle}\ \textsc{TupleDecl}$$

$$\textit{TDecl} : \langle \mathsf{let}\ \vec{x}\,\mathtt{:=}<\vec{v}> \mid L\,;\mathcal{N}\rangle \to \langle \mathsf{regular} \mid L[\vec{x} \mapsto \vec{v}]\,;\mathcal{N}\rangle \text{ if } \vec{x} \notin \mathsf{dom}(L)$$

$$\textit{VDecl} : \langle \mathsf{let}\ x\,\mathtt{:=}v \mid L\,;\mathcal{N}\rangle \to \langle \mathsf{regular} \mid L[x \mapsto v]\,;\mathcal{N}\rangle \qquad \text{if } x \notin \mathsf{dom}(L)$$

Fig. 4. Big-step (top) and small-step (bottom) declaration semantics. Similar rules for assignments (e.g. $\vec{x}\,\mathtt{:=}<\vec{v}>$, with $\vec{x} \in \mathsf{dom}(L)$) omitted for economy.

$$\frac{\langle M \mid G\,;L\,;\mathcal{N}\rangle \Downarrow_{\mathsf{exp}} \langle \mathsf{ff} \mid G_0\,;L_0\,;\mathcal{N}\rangle}{\langle \mathsf{if}\ MS \mid G\,;L\,;\mathcal{N}\rangle \Downarrow \langle \mathsf{regular} \mid G_0\,;L_0\,;\mathcal{N}\rangle}\ \textsc{IfFalse}$$

$$\frac{\langle M \mid G\,;L\,;\mathcal{N}\rangle \Downarrow_{\mathsf{exp}} \langle \mathsf{tt} \mid G_0\,;L_0\,;\mathcal{N}\rangle \qquad \langle S \mid G_0\,;L_0\,;\mathcal{N}\rangle \Downarrow \langle \mathbb{M} \mid G_1\,;L_1\,;\mathcal{N}\rangle}{\langle \mathsf{if}\ MS \mid G\,;L\,;\mathcal{N}\rangle \Downarrow \langle \mathbb{M} \mid G_1\,;L_1\,;\mathcal{N}\rangle}\ \textsc{IfTrue}$$

$$\frac{\langle M \mid G\,;L\,;\mathcal{N}\rangle \Downarrow_{\mathsf{exp}} \langle v \mid G_0\,;L_0\,;\mathcal{N}\rangle \qquad \langle S_d \mid G_0\,;L_0\,;\mathcal{N}\rangle \Downarrow \langle \mathbb{M} \mid G_1\,;L_1\,;\mathcal{N}\rangle}{\langle \mathsf{switch}\ MC_1..C_n\mathsf{default}S_d \mid G\,;L\,;\mathcal{N}\rangle \Downarrow \langle \mathbb{M} \mid G_1\,;L_1\,;\mathcal{N}\rangle}\ \textsc{SwD}$$

$$\frac{\langle M \mid G\,;L\,;\mathcal{N}\rangle \Downarrow_{\mathsf{exp}} \langle c_{n+1} \mid G_0\,;L_0\,;\mathcal{N}\rangle \quad \langle S_{n+1} \mid G_0\,;L_0\,;\mathcal{N}\rangle \Downarrow \langle \mathbb{M} \mid G_1\,;L_1\,;\mathcal{N}\rangle}{\langle \mathsf{switch}\ MC_1..C_{n+1}..\mathsf{default}S_d \mid G\,;L\,;\mathcal{N}\rangle \Downarrow \langle \mathbb{M} \mid G_1\,;L_1\,;\mathcal{N}\rangle}\ \textsc{SwC}$$

$$\textit{IfFalse} : \langle \mathsf{if}\ \mathsf{ff}\ S \mid L\,;\mathcal{N}\rangle \to \langle \mathsf{regular} \mid L\,;\mathcal{N}\rangle \qquad \textit{IfTrue} : \langle \mathsf{if}\ \mathsf{tt}\ S \mid L\,;\mathcal{N}\rangle \to \langle S \mid L\,;\mathcal{N}\rangle$$

$$\textit{SwDeflt} : \langle \mathsf{switch}\ v\ C_1..C_n\mathsf{default}\ S_d \mid L\,;\mathcal{N}\rangle \to \langle S_d \mid L\,;\mathcal{N}\rangle$$

$$\textit{SwCase} : \langle \mathsf{switch}\ c_{n+1}\ C_1..C_{n+1}..\mathsf{default}\ S_d \mid L\,;\mathcal{N}\rangle \to \langle S_{n+1} \mid L\,;\mathcal{N}\rangle$$

Fig. 5. Big-step (top) and small-step (bottom) branching semantics. For switch rules we assume side conditions: $C_i = \mathsf{case}\,c_i S_i$ and $v, c_{n+1} \notin \{c_1, .., c_n\}$.

(SwD). The same behaviour is captured by standard small-step rules. Source-level syntax requires inner statements in conditional and switch statements be blocks.

3.4 Loop Statements

Loops in Yul consist of four parts: an initialisation block, a condition, a post-iteration block, and a loop-body block. In well-formed Yul programs break and continue may appear only in a loop body. On the other hand, leave may appear in any part of a loop, so long as the loop itself occurs inside a function. The semantics for loops is given in Fig. 6.

Here we once again make use of the semantics of blocks, this time to handle the sequencing of loops. ForInit refactors the initialisation block out of the loop and places the whole loop in a new block that prefixes the initialisation statements — the small-step rule mirrors this exactly. Halt1 handles break and

$$\frac{\langle \{S_1..S_n\ \mathsf{for}\{\}MS_pS_b\} \mid G_1\,;L_1\,;\mathcal{N}\rangle \Downarrow \langle \mathbb{M} \mid G_2\,;L_2\,;\mathcal{N}\rangle \qquad n>0}{\langle \mathsf{for}\{S_1..S_n\}MS_pS_b \mid G_1\,;L_1\,;\mathcal{N}\rangle \Downarrow \langle \mathbb{M} \mid G_2\,;L_2\,;\mathcal{N}\rangle}\ \textsc{ForInit}$$

$$\frac{\langle M \mid G\,;L\,;\mathcal{N}\rangle \Downarrow_{\mathsf{exp}} \langle \mathsf{ff} \mid G_1\,;L_1\,;\mathcal{N}\rangle}{\langle \mathsf{for}\{\}MS_pS_b \mid G\,;L\,;\mathcal{N}\rangle \Downarrow \langle \mathsf{regular} \mid G_1\,;L_1\,;\mathcal{N}\rangle}\ \textsc{ForFalse}$$

$$\frac{\langle M \mid G\,;L\,;\mathcal{N}\rangle \Downarrow_{\mathsf{exp}} \langle \mathsf{tt} \mid G_1\,;L_1\,;\mathcal{N}\rangle \quad \langle S_b \mid G_1\,;L_1\,;\mathcal{N}\rangle \Downarrow \langle \mathbb{M}_{lb} \mid G_2\,;L_2\,;\mathcal{N}\rangle}{\langle \mathsf{for}\{\}MS_pS_b \mid G\,;L\,;\mathcal{N}\rangle \Downarrow \langle \mathbb{M}'_{lb} \mid G_2\,;L_2\,;\mathcal{N}\rangle}\ \textsc{Halt1}$$

$$\frac{\begin{array}{c}\langle M \mid G\,;L\,;\mathcal{N}\rangle \Downarrow_{\mathsf{exp}} \langle \mathsf{tt} \mid G_1\,;L_1\,;\mathcal{N}\rangle \\ \langle S_b \mid G_1\,;L_1\,;\mathcal{N}\rangle \Downarrow \langle \mathbb{M}_{\neq lb} \mid G_2\,;L_2\,;\mathcal{N}\rangle \quad \langle S_p \mid G_2\,;L_2\,;\mathcal{N}\rangle \Downarrow \langle \mathsf{leave} \mid G_3\,;L_3\,;\mathcal{N}\rangle\end{array}}{\langle \mathsf{for}\{\}MS_pS_b \mid G\,;L\,;\mathcal{N}\rangle \Downarrow \langle \mathsf{leave} \mid G_3\,;L_3\,;\mathcal{N}\rangle}\ \textsc{Halt2}$$

$$\frac{\begin{array}{c}\langle M \mid G\,;L\,;\mathcal{N}\rangle \Downarrow_{\mathsf{exp}} \langle \mathsf{tt} \mid G_1\,;L_1\,;\mathcal{N}\rangle \qquad \langle S_b \mid G_1\,;L_1\,;\mathcal{N}\rangle \Downarrow \langle \mathbb{M}_{\neq lb} \mid G_2\,;L_2\,;\mathcal{N}\rangle \\ \langle S_p \mid G_2;L_2;\mathcal{N}\rangle \Downarrow \langle \mathsf{regular} \mid G_3;L_3;\mathcal{N}\rangle \quad \langle \mathsf{for}\{\}MS_pS_b \mid G_3;L_3;\mathcal{N}\rangle \Downarrow \langle \mathbb{M} \mid G_4;L_4;\mathcal{N}\rangle\end{array}}{\langle \mathsf{for}\{\}MS_pS_b \mid G\,;L\,;\mathcal{N}\rangle \Downarrow \langle \mathbb{M} \mid G_4\,;L_4\,;\mathcal{N}\rangle}\ \textsc{Loop}$$

$$FI\textsc{nit} : \langle \mathsf{for}\{S_1..S_n\}MS_pS_b \mid L\,;\mathcal{N}\rangle \to \langle \{S_1..S_n\ \mathsf{for}\{\}MS_pS_b\} \mid L\,;\mathcal{N}\rangle \qquad \text{if } n>0$$

$$FI\textsc{f} : \langle \mathsf{for}\{\}MS_pS_b \mid L\,;\mathcal{N}\rangle \to \langle [\![\mathsf{if}\ M\{[\![S_b]\!]_{\mathsf{cnt}}S_p\mathsf{for}\{\}MS_pS_b\}]\!]_{\mathsf{brk}} \mid L\,;\mathcal{N}\rangle$$

$$FC\textsc{nt1} : \langle [\![\mathbb{M}]\!]_{\mathsf{cnt}} \mid L\,;\mathcal{N}\rangle \to \langle \mathbb{M} \mid L\,;\mathcal{N}\rangle \qquad \text{if } \mathbb{M} \in \{\mathsf{regular}, \mathsf{break}, \mathsf{leave}\}$$

$$FC\textsc{nt2} : \langle [\![\mathsf{continue}]\!]_{\mathsf{cnt}} \mid L\,;\mathcal{N}\rangle \to \langle \mathsf{regular} \mid L\,;\mathcal{N}\rangle$$

$$FB\textsc{rk1} : \langle [\![\mathbb{M}]\!]_{\mathsf{brk}} \mid L\,;\mathcal{N}\rangle \to \langle \mathbb{M} \mid L\,;\mathcal{N}\rangle \qquad \text{if } \mathbb{M} \in \{\mathsf{regular}, \mathsf{leave}\}$$

$$FB\textsc{rk2} : \langle [\![\mathsf{break}]\!]_{\mathsf{brk}} \mid L\,;\mathcal{N}\rangle \to \langle \mathsf{regular} \mid L\,;\mathcal{N}\rangle$$

Fig. 6. Big-step (top) and small-step (bottom) loop semantics. Side conditions: $(\mathbb{M}_{lb}, \mathbb{M}'_{lb}) \in \{(\mathsf{leave}, \mathsf{leave}), (\mathsf{break}, \mathsf{regular})\}$, whereas $\mathbb{M}_{\neq lb} \in \{\mathsf{regular}, \mathsf{continue}\}$.

leave statements within the loop body, while Halt2 handles leave statements within the post-iteration. Note that continue is handled implicitly by the big-step rules; we piggyback off SeqI in Fig. 3 to exit out of the loop body if continue– i.e. an irregular mode — is encountered. The small-step side handles irregular modes more explicitly: special evaluation contexts $[\![\ldots]\!]_{\mathsf{brk}}$ and $[\![\ldots]\!]_{\mathsf{cnt}}$ are used to encode the frames that check for break and continue respectively. Lastly, note that while Loop directly executes the loop body and otherwise breaks with ForFalse, the small-step equivalent further leverages the semantics of conditional statements to encode this behaviour.

Remark 8 (Break optimisation). Multiple small-step unrollings of a for-loop result in statements containing the pattern $[\![\{[\![\{[\![\ldots\{[\![\cdot]\!]_{\mathsf{brk}}\}^{\mathcal{N}}_{L_n}\ldots]\!]_{\mathsf{brk}}\}^{\mathcal{N}}_{L_2}]\!]_{\mathsf{brk}}\}^{\mathcal{N}}_{L_1}]\!]_{\mathsf{brk}}$ with $\mathsf{dom}(L_1) = \ldots = \mathsf{dom}(L_n)$. Using our semantics we prove that such statements can be simplified by dropping the inner break and block contexts. This has brought significant speedup in the implementation of our interpreter. That is, for any E that contains only interleavings of $[\![\cdot]\!]_{\mathsf{brk}}$ and empty $\{\cdot\}^{\mathcal{N}}_{L_j}$ contexts with

$$\frac{}{\langle x \mid G\,;L\,;\mathcal{N}\rangle \Downarrow_{\mathsf{exp}} \langle L(x) \mid G\,;L\,;\mathcal{N}\rangle}\ \textsc{Id} \qquad \frac{\langle M \mid G\,;L\,;\mathcal{N}\rangle \Downarrow_{\mathsf{exp}} \langle \mathsf{regular} \mid G'\,;L'\,;\mathcal{N}'\rangle}{\langle M \mid G\,;L\,;\mathcal{N}\rangle \Downarrow \langle \mathsf{regular} \mid G'\,;L'\,;\mathcal{N}'\rangle}\ \textsc{Exp}$$

$$\frac{\begin{array}{c}\langle M_n \mid G;L;\mathcal{N}\rangle \Downarrow_{\mathsf{exp}} \langle v_n \mid G_1;L_1;\mathcal{N}\rangle \ldots \langle M_1 \mid G_{n-1};L_{n-1};\mathcal{N}\rangle \Downarrow_{\mathsf{exp}} \langle v_1 \mid G_n;L_n;\mathcal{N}\rangle \\ \langle S_b \mid G_n\,;L_f\,;\mathcal{N}\rangle \Downarrow \langle \mathbb{M} \mid G''\,;L'\,;\mathcal{N}\rangle \qquad \mathbb{M} \in \{\mathsf{regular},\mathsf{leave}\}\end{array}}{\langle f(M_1,..,M_n) \mid G\,;L\,;\mathcal{N}\rangle \Downarrow_{\mathsf{exp}} \langle <L'(z_1),..,L'(z_m)> \mid G''\,;L_n\,;\mathcal{N}\rangle}\ \textsc{Fun}$$

$$\frac{\begin{array}{c}\langle M_n \mid G;L;\mathcal{N}\rangle \Downarrow_{\mathsf{exp}} \langle v_n \mid G_1;L_1;\mathcal{N}\rangle \ldots \langle M_1 \mid G_{n-1};L_{n-1};\mathcal{N}\rangle \Downarrow_{\mathsf{exp}} \langle v_1 \mid G_n;L_n;\mathcal{N}\rangle \\ \langle op\,(v_1,..,v_n) \mid G_n\rangle \Downarrow_{\mathsf{opc}} \langle v'_1,..,v'_m \mid G''\rangle\end{array}}{\langle op\,(M_1,..,M_n) \mid G\,;L\,;\mathcal{N}\rangle \Downarrow_{\mathsf{exp}} \langle <v'_1,..,v'_m> \mid G''\,;L_n\,;\mathcal{N}\rangle}\ \textsc{Opc}$$

$$\textsc{Ident} : \langle x \mid L\,;\mathcal{N}\rangle \to \langle L(x) \mid L\,;\mathcal{N}\rangle \qquad \text{if } x \in L$$

$$FCall : \langle f(\vec{v}) \mid L\,;\mathcal{N}\rangle \to \langle (\!|S_b|\!)^{\vec{z}}_{L} \mid L_f\,;\mathcal{N}\rangle$$

$$FRet : \langle (\!|\mathbb{M}|\!)^{z_1,..,z_m}_{L} \mid L'\,;\mathcal{N}\rangle \to \langle <L'(z_1),..,L'(z_m)> \mid L\,;\mathcal{N}\rangle \quad \text{if } \mathbb{M} \in \{\mathsf{leave},\mathsf{regular}\}$$

$$\textsc{OpCall} : \langle op\,(\vec{v}) \mid G;L;\mathcal{N}\rangle \to \langle S' \mid G';L;\mathcal{N}\rangle \qquad \text{if } \langle op\,(\vec{v}) \mid G\rangle \Downarrow_{\mathsf{opc}} \langle S' \mid G'\rangle \text{ and } S' \in \{<\vec{v}>, \mathcal{M}\}$$

Fig. 7. Big-step (top) and small-step (bottom) expression semantics. Side-conditions: $\mathcal{N}(f) = ((y_1,..,y_n),(z_1,..,z_m),S_b)$, $L_f = \{(y_1,..,y_n) \mapsto (v_1,..,v_n)\} \uplus \{\{z_1,..,z_m\} \mapsto 0\}$. For uniformity, we slightly abuse tuple notation to write $<v> = v$ and $<> = \mathsf{regular}$.

the same $\mathcal{N}$ and domain in all L_j's:

$$\langle [\![E[[\![S']\!]_{\mathsf{brk}}]]\!]_{\mathsf{brk}} \mid L';\mathcal{N}'\rangle \to^* \langle \mathbb{M} \mid L'';\mathcal{N}''\rangle \text{ iff } \langle [\![E[S']]\!]_{\mathsf{brk}} \mid L';\mathcal{N}'\rangle \to^* \langle \mathbb{M} \mid L'';\mathcal{N}''\rangle$$
$$\langle \{E[\{S'\}^{\mathcal{N}}_{L_2}]\}^{\mathcal{N}}_{L_1} \mid L';\mathcal{N}'\rangle \to^* \langle \mathbb{M} \mid L'';\mathcal{N}''\rangle \text{ iff } \langle \{E[S']\}^{\mathcal{N}}_{L_1} \mid L';\mathcal{N}'\rangle \to^* \langle \mathbb{M} \mid L'';\mathcal{N}''\rangle$$

3.5 Expressions

Finally, we define the semantics of expressions, handling variable look-up, function calls and opcode calls. Because of this, they either return values, regular, or an external irregular mode $\mathcal{M}$ if handled. The rules are given in Fig. 7.

Id and its small-step counterpart behave as expected; they look up the value of x within L. Function calls (Fun) on the other hand are less standard. Instead of substitution for function application, calls assign in L the correct value for every argument and then leverage the semantics of blocks to set up the namespace. Similarly, for returns, instead of a return statement, functions assign to the return variables and look up their values in the local L, again leveraging block semantics to clean up the namespace. Note that in Fun, functions may return either a sequence of values if there are return variables, or regular if the function is not expected to return anything (analogous to void functions in C-like languages). As in Sec. 3.2, Fun leaves this unrestricted but argument evaluation does not change L. Opcode calls (Opc) mirror function calls. Note that we do not handle external modes in the big-step semantics. For the small-step transitions involving function calls we introduce a special context $(\!|\ldots|\!)^{\vec{z}}_{L}$. Much like with other special contexts introduced in the small-step semantics, its purpose is to encode within

the evaluation context all the frames that form the call stack, including the local state L to reinstate and the return variables $\vec{z}$. Rule Exp allows void function calls to be evaluated as statements. Example 9 illustrates call scoping.

Example 9. Let $\mathcal{N} = f \mapsto ((a,b), r, \{r\texttt{:=}\mathsf{add}(a,b)\,\mathsf{function}\,g()\{\}\})$, $L' = \{a \mapsto 5, b \mapsto 6, r \mapsto 0\}$, $\mathcal{N}' = \mathcal{N} \uplus \{g \mapsto (\varepsilon, \varepsilon, \{\})\}$, and let there be some L.

$$\langle f(5,6) \mid L ; \mathcal{N}\rangle \to \langle (\!|\{r\texttt{:=}\mathsf{add}(a,b)\,\mathsf{function}\,g()\{\}\}|\!)^{r}_{L} \mid L' ; \mathcal{N}\rangle$$
$$\to \langle (\!|\{r\texttt{:=}\mathsf{add}(a,b)\,\mathsf{function}\,g()\{\}\}^{\mathcal{N}}_{L'}|\!)^{r}_{L} \mid L' ; \mathcal{N}'\rangle$$
$$\to^{*} \langle (\!|\{\mathsf{regular}\}^{\mathcal{N}}_{L'}|\!)^{r}_{L} \mid L'[r \mapsto 11]; \mathcal{N}'\rangle \to \langle (\!|\mathsf{regular}|\!)^{r}_{L} \mid L'[r \mapsto 11]; \mathcal{N}\rangle \to \langle 11 \mid L; \mathcal{N}\rangle \quad \blacksquare$$

Remark 10 (Undefined void return and modes). The evaluation function provided in the Yul specification [38] is undefined for void function calls. This introduces ambiguity in the intended details of the Yul semantics. We decided to resolve this in a consistent manner to [38], making modes a subset of statements, produced by evaluation, and for regular to be returned by void functions. Since such functions are called in block statements, it was necessary that they return regular in order for the evaluation of subsequent block statements to occur.

4 Semantic Equivalence

Here we show that the big-step and small-step semantics are equivalent. A necessary result is compatibility of the small-step semantics with respect to evaluation contexts.

Lemma 11 (Small-step Compatibility). *Given a context E and reduction $\langle S \mid G ; L ; \mathcal{N}\rangle \to^{*} \langle S' \mid G' ; L' ; \mathcal{N}'\rangle$ then $\langle E[S] \mid G; L; \mathcal{N}\rangle \to^{*} \langle E[S'] \mid G'; L'; \mathcal{N}'\rangle$.*

Proof. This follows by induction on the length of the reduction. □

Theorem 12 (Semantics Equivalence). *Given statement S in source syntax, statement sequence $\vec{S}$, expression M, global environment G, local state L, and function namespace $\mathcal{N}$, then:*

1. $\langle S \mid G ; L ; \mathcal{N}\rangle \Downarrow \langle \mathbb{M} \mid G' ; L' ; \mathcal{N}'\rangle$ *iff* $\langle S \mid G ; L ; \mathcal{N}\rangle \to^{*} \langle \mathbb{M} \mid G' ; L' ; \mathcal{N}'\rangle$
2. $\langle \vec{S} \mid G ; L ; \mathcal{N}\rangle \Downarrow_{\mathsf{seq}} \langle \mathbb{M} \mid G' ; L' ; \mathcal{N}'\rangle$ *iff* $\langle \vec{S} \mid G ; L ; \mathcal{N}\rangle \to^{*} \langle \mathbb{M} \mid G' ; L' ; \mathcal{N}'\rangle$
3. $\langle M \mid G ; L ; \mathcal{N}\rangle \Downarrow_{\mathsf{exp}} \langle S' \mid G' ; L' ; \mathcal{N}'\rangle$ *iff* $\langle M \mid G ; L ; \mathcal{N}\rangle \to^{*} \langle S' \mid G' ; L' ; \mathcal{N}'\rangle$

where $S' \in \{v, <\vec{v}>, \mathsf{regular}\}$.

Proof. We prove the three statements simultaneously in each direction. For the forward direction we proceed by induction on the size of the big-step derivation trees, and for the reverse direction we proceed by induction on the length of the reduction sequences. Here we show the case of function application where $S = f(M_1, .., M_n)$ and $\mathcal{N}(f) = (\vec{y}, \vec{z}, S_b)$. We refer to Lem. 11 by SC and the inductive hypothesis by IH.

Forward direction. We assume the big-step derivation from rule Fun in Fig. 7:

$$\langle f(M_1, .., M_n) \mid G ; L ; \mathcal{N}\rangle \Downarrow_{\mathsf{exp}} \langle S_{ret} \mid G' ; L_n ; \mathcal{N}\rangle$$

with $L_f = \{\vec{y} \mapsto \vec{v}\} \uplus \{\{\vec{z}\} \mapsto 0\}$ and S_{ret} being: $L'(z_1)$ if $|\vec{z}| = 1$; $<L'(z_1), .., L'(z_m)>$ if $|\vec{z}| > 1$; regular if $|\vec{z}| = 0$. We then obtain the corresponding reduction as follows:

$$
\begin{array}{ll}
\langle f(M_1, .., M_n) \mid G ; L ; \mathcal{N} \rangle & \\
\to^* \langle f(M_1, .. M_{n-1}, v_n) \mid G_1 ; L_1 ; \mathcal{N} \rangle & \text{by SC and IH} \langle M_n \mid G ; L ; \mathcal{N} \rangle \\
\ldots & \\
\to^* \langle f(v_1, .., v_n) \mid G_n ; L_n ; \mathcal{N} \rangle & \text{by SC and IH} \langle M_1 \mid G_{n-1} ; L_{n-1} ; \mathcal{N} \rangle \\
\to \langle (\!| S_b |\!)^{\vec{z}}_{L} \mid G_n ; L_f ; \mathcal{N} \rangle & \text{where } \mathcal{N}(f), L_f \text{ as above} \\
\to^* \langle (\!| \mathbb{M} |\!)^{\vec{z}}_{L_n} \mid G' ; L' ; \mathcal{N} \rangle & \text{by SC and IH} \langle S_b \mid G_n ; L_f ; \mathcal{N} \rangle \\
\to \langle S_{ret} \mid G' ; L_n ; \mathcal{N} \rangle &
\end{array}
$$

Reverse direction. For a term $S = f(M_1, .., M_n)$, a reduction to S_{ret} requires the intermediate reductions:

$$
\begin{array}{l}
\langle f(M_1, .., M_n) \mid G_0 ; L_0 ; \mathcal{N} \rangle \to^* \langle f(v_1, .., v_n) \mid G_n ; L_n ; \mathcal{N} \rangle \\
\to^* \langle (\!| S_b |\!)^{\vec{z}}_{L_n} \mid G_n ; L_f ; \mathcal{N} \rangle \to^* \langle (\!| \mathbb{M} |\!)^{\vec{z}}_{L_n} \mid G' ; L'' ; \mathcal{N} \rangle \to^* \langle S_{ret} \mid G' ; L_n ; \mathcal{N} \rangle
\end{array}
$$

with $\mathcal{N}(f), L_f, S_{ret}$ as in the forward direction above, and intermediate steps:

$$\langle M_i \mid G_{n-i} ; L_{n-i} ; \mathcal{N} \rangle \to^* \langle v_i \mid G_{n-i+1} ; L_{n-i+1} ; \mathcal{N} \rangle \qquad (\mathrm{A}_i)$$

$$\langle S_b \mid G_n ; L_f ; \mathcal{N} \rangle \to^* \langle \mathbb{M} \mid G' ; L' ; \mathcal{N} \rangle \qquad (\mathrm{B})$$

To obtain (1), by IH on (A_i): $\langle M_i \mid G_{n-i} ; L_{n-i} ; \mathcal{N} \rangle \Downarrow \langle v_i \mid G_{n-i+1} ; L_{n-i+1} ; \mathcal{N} \rangle$. Next, by IH on (B): $\langle S_b \mid G_n ; L_f ; \mathcal{N} \rangle \Downarrow \langle \mathbb{M} \mid G' ; L' ; \mathcal{N} \rangle$. We therefore obtain by Fun: $\langle f(M_1, .., M_n) \mid G_0 ; L_0 ; \mathcal{N} \rangle \Downarrow_{\mathsf{exp}} \langle S_{ret} \mid G' ; L_n ; \mathcal{N} \rangle$. □

5 Implementation

We implemented YulTracer [27, 28], a prototype interpreter based on our operational semantics for Yul that is parametric on dialect implementations. Currently, a partial implementation of the Shanghai upgrade (Solidity 0.8.20 to 0.8.23) of the EVM execution specifications [16] is provided as its sole dialect. We provide opcodes for the following categories: Arithmetic, Bitwise, Comparison, Control-flow, Memory, Stack, and Storage. We omit the implementation of opcodes that are not listed in the official documentation for Yul [38] under version 0.8.23, but otherwise adapt the Python specification to our use as faithfully as possible. YulTracer implements our small-step semantics in the style of CEK machines [20] and is written using OCaml. Modularity for dialects is achieved using functors: a module signature is provided for dialects to satisfy, which are then used to instantiate any concrete implementation of the interpreter.

We evaluate our interpreter using custom tests—1263 Lines of Code (LoC)—and a subset of tests found in the Solidity compiler repository [17]—269 LoC. From the 50 files found under test/libyul/yulInterpreterTests in the Solidity repository (commit 2b2c76c), we kept 19 files that do not make use of unimplemented EVM instructions: opcodes for inter-contract communication in the System category;

those in the Block, Environment or Log categories; the opcode for Keccak, which is implemented but unused as it uses a model that requires symbolic execution; or opcodes implemented in the Cancun upgrade (i.e. Solidity 0.8.24) such as those for transient memory or mcopy. From the 19 remaining tests, we mark and omit 3 tests that do not behave according to the official specifications of the EVM; these are: access_large_memory_offsets.yul, which under standard operation would exceed the gas limit when trying to access large offsets but does not; expr_nesting_depth_exceeded.yul, which imposes a nesting limit on expressions that is not specified in the Yul documentation; and hex_literals.yul, which uses underscores _ in hexadecimal strings, not allowed by the official Yul grammar. For our own custom examples, we implemented two categories of programs: sequence generators (426 LoC) and sorting algorithms (837 LoC). For the former, we have generators for Catalan numbers, the Heighway dragon curve, Fibonacci numbers, Pell numbers, Prime numbers, and the Thue–Morse sequence. For the latter we have implementations of bubble, heap, insertion, quick and shell sort on arrays of 6, 300 and 1000 items. These were chosen as non-trivial algorithms with standard implementations that we can compare to for reference that also make use of a range of Yul features and EVM opcodes while testing YulTracer's performance.

All examples executed correctly on a laptop featuring an 8-core Intel Core i7-8665U CPU with 32 GiB of RAM running Ubuntu 23.04. The interpreter was compiled with the OCaml 4.10.0 compiler using the Dune build system. We measure for each batch of examples a three-trial average total execution time of 0.478s for all 8 sequence generator files, 31.960s for all 12 sorting algorithm files, and 2.932s for all 16 of the Solidity tests (including an infinite loop set to time out at 2s). YulTracer produced the expected output for all programs.

6 Related Work

As discussed in the Introduction, the body of work on formalising Yul involves mechanisations in e.g. Dafny [11,12], the K-framework [18], Isabelle/HOL [3], Lean [34], and ACL2 [25]. To our knowledge, there is no peer-reviewed tool-independent presentation of a formal semantics for Yul. Closely related is the area of formalising Solidity, for which a more substantial body of work exists: e.g. [32] mathematically presents a calculus to reason about Solidity smart contracts and then implements and mechanically verifies its soundness in Isabelle/HOL; [10] presents a shallow embedding of Solidity in F*; [14] presents an operational semantics for a subset of Solidity and presents an abstract model for memory; [40] formalises another subset of Solidity by presenting a big-step semantics; [24] presents a semantics for the memory model of Solidity by mapping a fragment of Solidity to SMT-based constructs; [31] implements a denotational semantics for a subset of Solidity in 1500 lines of Isabelle. Also relevant are specifications for the EVM, which are necessary to implement the EVM dialect for Yul, such as: the Ethereum execution specification in Python [16]; other formalisation work such as [23], which presents the first complete small-step semantics of EVM bytecode and formalises it in F*; and [13], which presents a complete and formal specification in Dafny. Lastly, formalisations efforts are also common practice outside smart contracts, such as

with the Glasgow Haskell Compiler (GHC) intermediate language Core which implements System FC [39] and features a maintained formal specification [35].

7 Conclusions

In this paper, we have presented an operational semantics for Yul. We first defined a source-level abstract syntax constructed by inspecting the grammar specified in [38] (Sec. 2), which we subsequently extended with constructs necessary for evaluation (Sec. 3). We then presented big- and small-step semantics, constructed by carefully inspecting the pseudocode evaluation function provided in [38], and supplied a proof of semantic equivalence between them (Thm. 12). Lastly, we presented YulTracer, an interpreter that implements our small-step semantics for a subset of the EVM dialect based on the Shanghai upgrade [16]. We evaluated our implementation against tests from the Solidity repository [17] and our own ones based on standard sequence generators and sorting algorithms.

We believe our work offers multiple advantages. Being in standard mathematical notation, and free of implementation or formalisation considerations, our semantics can serve to unify various extant tool-specific formalisations and verify any current and future implementations, while it can also be directly used in mathematical proofs, especially those involving the development and verification of language properties. Moreover, we provide an exhaustive specification for statements that irons out issues with ambiguity arising from under-specification of the semantics; e.g. function returns with no return variables. Lastly, we anticipate that a semantics such as ours will aid the development of future theories for Yul. We shall discuss this last point in more detail next.

This document focuses only the semantics of Yul statements. As such, we leave a detailed discussion of the EVM dialect and object semantics for future work. In particular, the EVM semantics includes behaviours which may be of technical interest, such as instructions related to transactions and interaction with the blockchain, as well as inter-contract communication. YulTracer and our semantics work towards the development of symbolic execution tools and theories developed directly in Yul, avoiding low-level complexities of EVM bytecode and the feature-rich, evolving syntax of Solidity; in particular those which combine symbolic execution with game semantics (e.g. [29] and [26]) to model higher-order interactions of smart contracts relevant to reentrancy exploits. Lastly, our semantics could lead to soundness proofs for a future Yul type system.

Acknowledgements: This work was supported in part by: Science Foundation Ireland under grant number 13/RC/2094_2; the Cisco University Research Program Fund, a corporate advised fund of Silicon Valley Community Foundation; and Ethereum Foundation grant FY23-1127. This version of the contribution has been accepted for publication, after peer review but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record is available online at: https://doi.org/10.1007/978-3-031-77382-2_19. Use of this Accepted Version is subject to the publisher's Accepted Manuscript terms of use https://www.springernature.com/gp/open-research/policies/accepted-manuscript-terms.

References

1. Smart Contract Vulnerabilities (SCV) list, https://github.com/sirhashalot/SCV-List, accessed 6 June 2024
2. Oyente (Mar 2017), https://github.com/enzymefinance/oyente/releases/tag/0.2.7
3. Yul-Isabelle: Executable formal semantics of Yul (2021), https://github.com/mmalvarez/Yul-Isabelle/tree/master, accessed 24 June 2024
4. Manticore (Feb 2022), https://github.com/trailofbits/manticore/releases/tag/0.3.7
5. Echidna (Mar 2024), https://github.com/crytic/echidna/releases/tag/v2.2.3
6. Gambit: Mutant generation for Solidity (May 2024), https://github.com/Certora/gambit/releases/tag/v1.0.5
7. Mythril (Mar 2024), https://github.com/Consensys/mythril/releases/tag/v0.24.8
8. Slither (Jun 2024), https://github.com/crytic/slither/releases/tag/0.10.3
9. solhint (May 2024), https://github.com/protofire/solhint/releases/tag/v5.0.1
10. Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Kulatova, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., Zanella-Béguelin, S.: Formal verification of smart contracts: Short paper. In: Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security. p. 91–96. PLAS '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2993600.2993611
11. Cassez, F.: Formal and executable semantics of Yul in Dafny (oct 2023), https://hackmd.io/@FranckC/BJz02K4Za, accessed 24 June 2024
12. Cassez, F.: yul-dafny (oct 2023), https://github.com/franck44/yul-dafny, accessed 24 June 2024
13. Cassez, F., Fuller, J., Ghale, M.K., Pearce, D.J., Quiles, H.M.A.: Formal and executable semantics of the Ethereum Virtual Machine in Dafny. In: Chechik, M., Katoen, J.P., Leucker, M. (eds.) Formal Methods. pp. 571–583. Springer (2023)
14. Crosara, M., Centurino, G., Arceri, V.: Towards an operational semantics for Solidity (2019), https://api.semanticscholar.org/CorpusID:249983842
15. Crystal Intelligence: Adolescent anarchy: Thirteen years of crypto crimes unveiled, https://crystalintelligence.com/rohirov/2024/06/Crystal-Intelligence-Thirteen-Years-of-Crypto-Crimes-Unveiled.pdf, accessed 25 June 2024
16. ethereum.org: Ethereum execution client specifications, https://github.com/ethereum/execution-specs.git, accessed 14 June 2024
17. ethereum.org: The Solidity contract-oriented programming language, https://github.com/ethereum/solidity.git, accessed 13 June 2024
18. ethereum.org: Yul-K (2019), https://github.com/ethereum/Yul-K, accessed 24 June 2024
19. Feist, J., Greico, G., Groce, A.: Slither: A static analysis framework for smart contracts (05 2019). https://doi.org/10.1109/WETSEB.2019.00008
20. Felleisen, M., Friedman, D.P.: Control operators, the SECD-machine, and the λ-calculus. In: Wirsing, M. (ed.) Formal Description of Programming Concepts - III: Proceedings of the IFIP TC 2/WG 2.2 Working Conference on Formal Description of Programming Concepts - III, Ebberup, Denmark, 25-28 August 1986. pp. 193–222. North-Holland (1987)
21. Grech, N., Lagouvardos, S., Tsatiris, I., Smaragdakis, Y.: Elipmoc: Advanced decompilation of Ethereum smart contracts. Proc. ACM Program. Lang. **6**(OOPSLA1) (apr 2022). https://doi.org/10.1145/3527321
22. Grieco, G., Song, W., Cygan, A., Feist, J., Groce, A.: Echidna: Effective, usable, and fast fuzzing for smart contracts. pp. 557–560 (07 2020). https://doi.org/10.1145/3395363.3404366

23. Grishchenko, I., Maffei, M., Schneidewind, C.: A semantic framework for the security analysis of Ethereum smart contracts. In: Bauer, L., Küsters, R. (eds.) Principles of Security and Trust. pp. 243–269. Springer (2018)
24. Hajdu, Á., Jovanović, D.: SMT-friendly formalization of the Solidity memory model. In: Müller, P. (ed.) Programming Languages and Systems. pp. 224–250. Springer (2020)
25. Kestrel Institute: Yul directory, Kestrel Books in ACL2 repository (2023), https://github.com/acl2/acl2/tree/master/books/kestrel/yul, accessed 24 June 2024
26. Koutavas, V., Lin, Y., Tzevelekos, N.: From bounded checking to verification of equivalence via symbolic up-to techniques. In: TACAS. LNCS, vol. 13244, pp. 178–195. Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_10
27. Lin, Y.Y.: LaifsV1/YulTracer: Alpha 0.1.1: Parametric Parser and Main (Jun 2024). https://doi.org/10.5281/zenodo.12511493, https://doi.org/10.5281/zenodo.12511493
28. Lin, Y.Y.: Yultracer: Alpha 0.1.1 (artefact) (Jun 2024). https://doi.org/10.5281/zenodo.12588076, https://doi.org/10.5281/zenodo.12588076
29. Lin, Y., Tzevelekos, N.: Symbolic execution game semantics. In: FSCD. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020)
30. Luu, L., Chu, D., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016. pp. 254–269. ACM (2016). https://doi.org/10.1145/2976749.2978309
31. Marmsoler, D., Brucker, A.D.: A denotational semantics of Solidity in Isabelle/HOL. In: Calinescu, R., Păsăreanu, C.S. (eds.) Software Engineering and Formal Methods. pp. 403–422. Springer (2021)
32. Marmsoler, D., Thornton, B.: SSCalc: A calculus for Solidity smart contracts. In: Ferreira, C., Willemse, T.A.C. (eds.) Software Engineering and Formal Methods. pp. 184–204. Springer (2023)
33. Mossberg, M., Manzano, F., Hennenfent, E., Groce, A., Grieco, G., Feist, J., Brunson, T., Dinaburg, A.: Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. pp. 1186–1189 (11 2019). https://doi.org/10.1109/ASE.2019.00133
34. NethermindEth: Yul IR specification in Lean (2021), https://github.com/NethermindEth/Yul-Specification, accessed 24 June 2024
35. Peyton Jones, S.: System FC, as implemented in GHC (May), https://gitlab.haskell.org/ghc/ghc/-/blob/master/docs/core-spec/core-spec.pdf, accessed 27 June 2024
36. Sharma, N., Sharma, S.: A survey of Mythril, a smart contract security analysis tool for EVM bytecode **13**, 51003–51010 (12 2022)
37. Solidity Team: List of known bugs - Solidity 0.8.27 documentation, https://docs.soliditylang.org/en/develop/bugs.html, accessed 6 June 2024
38. Solidity Team: Yul - Solidity 0.8.27 documentation, https://docs.soliditylang.org/en/latest/yul.html, accessed 6 June 2024
39. Sulzmann, M., Chakravarty, M., Peyton Jones, S., Donnelly, K.: System F with type equality coercions. In: ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI'07). pp. 53–66. ACM (January 2007), https://www.microsoft.com/en-us/research/publication/system-f-with-type-equality-coercions/
40. Zakrzewski, J.: Towards verification of Ethereum smart contracts: A formalization of core of Solidity. In: Piskac, R., Rümmer, P. (eds.) Verified Software. Theories, Tools, and Experiments. pp. 229–247. Springer (2018)