# Extracting executable prose specification from formal semantics of Web Assembly

ANONYMOUS AUTHOR(S)

WebAssembly (Wasm) is a versatile binary instruction format enabling high-performance code execution across diverse environments. As Wasm gains momentum with continuous feature enhancements, the specification of its execution semantics becomes pivotal. Wasm-DSL has emrged as an innovative approach to streamline the meticulous documentation process: Wasm-DSL is used as front-end language to define semantics and automatically generate both formal and prose notations as back-ends. While generating formal notations is relatively straightforward, the challenge of generating accurate and consistent prose notations arises due to the fundamental disparity between prose and formal notations.

This paper presents Algorithmic Language (AL), an executable language designed to resemble prose, automating the generation of precise prose descriptions. AL is extracted from Wasm-DSL, and a comprehensive testing framework validates its correctness. This approach addresses the challenges posed by manual specification composition and offers a transformative solution for enhancing the accuracy and efficiency of programming language specifications.

Additional Key Words and Phrases: WebAssembly, language specification, executable prose, DSL,

**ACM Reference Format:**
Anonymous Author(s). 2024. Extracting executable prose specification from formal semantics of Web Assembly . In . ACM, New York, NY, USA, 5 pages. https://doi.org/XXXXXXX.XXXXXXX

## 1 INTRODUCTION

WebAssembly, commonly referred to as Wasm, is a versatile and efficient binary instruction format that enables high-performance execution of code across diverse environments, from web browsers to edge devices. The design philosophy of Wasm revolves around platform independence, security, and efficient performance, resulting in a versatile technology that bridges the gap between high-level programming languages and the web ecosystem. As the adoption of Wasm gains momentum, a continuous stream of new features is incorporated, extending the capabilities of the language.

At the heart of WebAssembly's functionality lies its execution semantics, which define how WebAssembly programs operate. The behavior of a WebAssembly program is defined in terms of an abstract machine that models the program's state. This abstract machine encompasses a stack to record operand values and control constructs, as well as an abstract store housing global state information. The execution process unfolds by iteratively executing each instruction of the WebAssembly program, one step at a time. Each step alters the internal state of the abstract machine, with the precise effects being specified by reduction rules within the operational semantics.

The official WebAssembly specification [1] meticulously documents the execution semantics using the LaTeX framework, presenting them in two complementary yet equivalent forms: prose and formal notation. Prose notation provides a stepwise, algorithmic description of WebAssembly's execution semantics, offering explicit guidance on the operations to be executed on various parts


*PLDI'24, June 03–05, 2018, Woodstock, NY*

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

**ref.is_null**

1. Assert: due to validation, a reference value is on the top of the stack.
2. Pop the value *val* from the stack.
3. If *val* is ref.null *t*, then:
    a. Push the value i32. const 1 to the stack.
4. Else:
    a. Push the value i32. const 0 to the stack.

(a) Prose notation

$$\begin{array}{lcll} val\ \textsf{ref.is\_null} & \hookrightarrow & (\textsf{i32. const}\ 1) & (\text{if}\ val = \textsf{ref.null}\ t) \\ val\ \textsf{ref.is\_null} & \hookrightarrow & (\textsf{i32. const}\ 0) & (\text{otherwise}) \end{array}$$

(b) Formal notation

Fig. 1. Semantics of 'ref.is_null'

of the program state and their sequential order. This prose notation aims to provide a human-readable understanding of how WebAssembly instructions operate. In contrast, formal notation articulates these semantics through mathematical, declarative rules. This formal representation closely resembles the original operational semantics, comprising reduction rules that correspond to each WebAssembly instruction. The dual representation serves to cater to comprehension of both language implementers and designers, ensuring clarity and correctness within the language specification.

In Figure 1, we present an illustration of how the specification defines the execution semantics of the 'ref.is_null' WebAssembly instruction. This instruction evaluates whether a given WebAssembly value is null. In Figure 1a, we specify its behavior using prose notation. The description begins by asserting the existence of a value with a type of 'reference' at the top of the stack in line 1. Subsequently, in line 2, the value is removed from the stack. Depending on whether the removed value is 'ref.null t' or not, either the integer value 1 or 0 is pushed onto the stack in lines 3-a or 4-a, respectively. In Figure 1b, we provide the operational semantics using formal notation. The reduction rule specifies the shape of the stack before and after executing one step, potentially with side conditions indicating when this reduction should occur. For instance, the stack 'val ref.is_null' should reduce to (i32.const 1) only if 'val' equals 'ref.null t', and otherwise, it should reduce to (i32.const 0).

Creating and maintaining this meticulous specification holds paramount significance as it forms the foundation for consistent and reliable behavior among WebAssembly programs across different implementations. This is crucial because developers, implementers, and tool creators need a comprehensive understanding of these semantics. This understanding is essential for building WebAssembly implementations that are not only robust but also optimized for efficiency, with the specification playing a pivotal role in facilitating this understanding. The ongoing evolution and growing adoption of WebAssembly further emphasize the heightened importance of precision and clarity within this documentation. As the WebAssembly ecosystem continues to develop and establish its presence, the accuracy and clarity of the specification become even more critical to ensure smooth integration and dependable execution.

However, the challenge of crafting and maintaining precise specifications is particularly daunting due to the intricate nature of the documentation process. Compounded by the utilization of the complex LaTeX typesetting system, the manual authorship of these specifications becomes a laborious and error-prone endeavor. It demands meticulous attention to detail for accurate representation, and errors can be inadvertently introduced. A case in point is the incorporation of proposals for new features into Wasm 2.0. A substantial number of new instructions, including a staggering 80 SIMD instructions, were added, necessitating the manual composition of both formal and prose specifications for each instruction. Moreover, the specifications of several existing instructions underwent modifications. In the midst of this process, an error found its way into a prose

**call_indirect $x$ $y$**

1. ...
2. Let **ref** $a$ be the function reference $r$.
3. ...
4. Let $f$ be the function instance $S.\mathsf{funcs}[a]$.
5. ...
6. If $ft_{\text{actual}}$ and $ft_{\text{expect}}$ differ, then:
   a. Trap.
7. Invoke the function instance at address $a$.

(a) Prose notation

$$
\begin{array}{l}
S; F; (\mathsf{i32.const}\ i)\ (\mathsf{call\_indirect}\ x\ y) \hookrightarrow S; F; (\mathsf{invoke}\ a) \\
\quad (\mathsf{if}\ \ S.\mathsf{tables}[F.\mathsf{module.tableaddrs}[x]].\mathsf{elem}[i] = \mathsf{ref}\ a \\
\quad\ \wedge\ \ S.\mathsf{funcs}[a] = f \\
\quad\ \wedge\ \ F.\mathsf{module.types}[y] = f.\mathsf{type}) \\
S; F; (\mathsf{i32.const}\ i)\ (\mathsf{call\_indirect}\ x\ y) \hookrightarrow S; F; \mathsf{trap} \\
\quad (\mathsf{otherwise})
\end{array}
$$

(b) Formal notation

Fig. 2. Semantics of 'call_indirect'

specification, and it took two years to rectify this issue. This example, exemplified by the expansion to Wasm 2.0, underscores the challenges posed by the addition of new instructions, which demand a comprehensive understanding of their behavior. Simultaneously, modifying existing instructions introduces the risk of errors within the specification, thereby highlighting the complexities and associated risks of manual specification composition. Given Wasm's planned extensions, addressing this challenge becomes increasingly critical.

In response to the challenges of crafting precise specifications manually, a promising approach has emerged in the form of Wasm-DSL (WebAssembly Domain Specific Language). Wasm-DSL presents a domain-specific language tailored to describing the execution semantics of WebAssembly. This innovative language is designed to bridge the gap between high-level descriptions and concrete documentation. The key advantage of Wasm-DSL lies in its ability to offer a clear, visually aligned representation of the operational semantics of WebAssembly that is easy to read and write. By defining the semantics of WebAssembly operations in this specialized language as a front-end and then compiling to both formal and prose notation as back-ends, documenting the specification becomes a more streamlined process, potentially reducing the burden of manual specification writing and enhancing accuracy.

The process of generating a formal notation specification from Wasm-DSL is relatively straightforward. Since both Wasm-DSL and formal specifications adopt a declarative approach based on mathematical reduction rules, the translation from Wasm-DSL to formal notation can be achieved through a single, linear scan of the DSL. This one-to-one correspondence minimizes the risk of generating a specification that deviates from the intended execution semantics outlined in DSL. Even in cases where discrepancies arise, they can be readily identified and rectified.

Generating prose notation specification, however, presents a distinct and intricate challenge. In contrast to Wasm-DSL, prose notation entails an algorithmic, step-by-step description of execution semantics. The transition between these two styles is not straightforward, as it necessitates the mapping of declarative statements to algorithmic prose while ensuring consistency with the formal specification.

Figure 2 illustrates one of these challenges, depicting the manually-written specifications of the Wasm 'indirect_call' instruction. In the formal specification (Figure 2b), three premises are presented, each comprising a simple equality. These premises are translated into three instructions in the prose specification (Figure 2a), respectively. Notably, despite their identical format of simple equality, some of these premises are intended to establish a new variable, while others are designed

to serve as conditions. Consequently, the first two are translated into the Let statement, as seen in lines 2 and 4, while the last premise is transformed into the If statement in line 6. Adding to the complexity, the three premises in the formal specification could be arranged in any order, but the translated statements must adhere to a specific fixed order due to interdependencies. This challenge of inferring the role and order of the premises, which we refer to as "animation," has been shown to be NP-hard.

Furthermore, the inherent disparity between prose and formal notations poses a challenge in ensuring that the generated prose accurately reflects the intended behavior of the formal semantics. As an illustration, the prose specification in Figure 2a introduces temporary variables, such as ft_actual or ft_expect, which were absent in the formal specifications presented in Figure 2b. This divergence complicates manual verification of consistency since one must meticulously track such deviations. Given that the process of prose generation is notably susceptible to errors due to its intricate translation nature, the pursuit of a methodology to systematically verify its alignment with the formal notation stands as a substantial research endeavor.

In response to these challenges, we propose a novel solution: Algorithmic Language (AL), an executable language that closely resembles the structure and style of prose notation. Our approach seeks to automate and enhance the process of generating prose descriptions from the formal semantics described in Wasm-DSL. We achieve this in two pivotal steps. First, we establish an automated pipeline for the extraction of AL from Wasm-DSL. Subsequently, we generate prose descriptions by stringifying the extracted AL. We also identify the challenges encountered during the process including the animation problem previously mentioned, and suggest the effective, lightweight solutions for them. Furthermore, to underscore the correctness and reliability of the extracted prose, we have developed an interpreter for AL which enables automatic and rigorous testing of the behavior described by the extracted AL against the official WebAssembly test suite. Our results confirm the consistency and accuracy of our methodology, boasting a 100% pass rate across official WebAssembly tests, thereby validating the fidelity of the extracted prose descriptions.

In summary, this paper offers the following key contributions:

- We provide a formal definition of the syntax and semantics of Algorithmic Language (AL), an executable language designed to resemble prose specification.
- We present an automated method for extracting prose descriptions from the execution semantics of Wasm described in Wasm-DSL, substantially mitigating the identified challenges during the process.
- We establish the correctness and reliability of our automatically extracted prose descriptions by subjecting them to comprehensive testing against the official Wasm test suite.

The subsequent sections of this paper will delve into the specifics of our methodology, detailing the creation of AL, the extraction process and the challenges, the development of the AL interpreter, and the comprehensive testing framework employed. Through this exploration, we aim to not only address the challenges associated with prose notation in Wasm, but also explore the potential of AL as a transformative tool for enhancing the accuracy and efficiency of programming language specifications in general.

## 2 BACKGROUND

This section explains about Wasm-DSL. Consider merge this with introduction.

## 3 AL

This section explains about syntax and semantics of AL.

## 4 EXTRACTING AL FROM WASM-DSL

This section explains about translating Wasm-DSL into AL.

## 5 EVALUATION

This section explains about evaluation.

## 6 RELATED WORK

This section explains about related works.

## 7 CONCLUSION

To address the challenge of extracting prose specification of Wasm from DSL, we introduced Algorithmic Language (AL), an executable language closely mirroring the structure and style of prose notation. AL serves as an intermediary step in the automated generation of precise prose descriptions from Wasm-DSL. We established an automated pipeline for extracting AL from Wasm-DSL and validated the correctness of these descriptions by developing an AL interpreter and subjecting it to rigorous testing against the official WebAssembly test suite. Our results showcased a 100% pass rate, affirming the consistency and accuracy of the extracted prose descriptions.

In summary, our research presents a significant advancement in the realm of programming language specification. We offer an innovative solution to streamline the process of generating precise prose notation from formal semantics, mitigating the complexities and risks associated with manual composition. This approach not only enhances the accuracy and clarity of Wasm's documentation but also has the potential to revolutionize the field of programming language specification, bridging the gap between declarative and algorithmic descriptions.

## REFERENCES

[1] Andreas Rossberg. 2023. *WebAssembly Specification.* https://webassembly.github.io/spec/core/index.html