Bringing the WebAssembly Standard up to Speed with SpecTec

DONGJUN YOUN, WONHO SHIN, JAEHYUN LEE, and SUKYOUNG RYU, KAIST, South Korea
JOACHIM BREITNER, Independent, Germany
PHILIPPA GARDNER, Imperial College London, United Kingdom
SAM LINDLEY, The University of Edinburgh, United Kingdom
MATIJA PRETNAR, University of Ljubljana, Slovenia
XIAOJIA RAO, Imperial College London, United Kingdom
CONRAD WATT, University of Cambridge, United Kingdom
ANDREAS ROSSBERG, Independent, Germany

WebAssembly (Wasm) is a portable low-level bytecode language and virtual machine that has seen increasing use in a variety of ecosystems. Its specification is unusually rigorous – including a full formal semantics for the language – and every new feature must be specified in this formal semantics, in prose, and in the official reference interpreter before it can be standardized. With the growing size of the language, this manual process with its redundancies has become laborious and error-prone, and in this work, we offer a solution.

We present SpecTec, a domain-specific language (DSL) and toolchain that facilitates both the Wasm specification and the generation of artifacts necessary to standardize new features. SpecTec serves as a single source of truth — from a SpecTec definition of the Wasm semantics, we can generate a typeset specification, including formal definitions and prose pseudocode descriptions, and a meta-level interpreter. Further backends for test generation and interactive theorem proving are planned. We evaluate SpecTec's ability to represent the latest Wasm 2.0 and show that the generated meta-level interpreter passes 100% of the applicable official test suite. We show that SpecTec is highly effective at discovering and preventing errors by detecting historical errors in the specification that have been corrected and ten errors in five proposals ready for inclusion in the next version of Wasm. Our ultimate aim is that SpecTec should be adopted by the Wasm standards community and used to specify future versions of the standard.

CCS Concepts: • **Theory of computation → Program specifications**; • **Software and its engineering → Syntax**; **Semantics**; **Specification languages**; **Domain specific languages**.

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

ACM Reference Format:
Dongjun Youn, Wonho Shin, Jaehyun Lee, Sukyoung Ryu, Joachim Breitner, Philippa Gardner, Sam Lindley, Matija Pretnar, Xiaojia Rao, Conrad Watt, and Andreas Rossberg. 2024. Bringing the WebAssembly Standard up to Speed with SpecTec. *Proc. ACM Program. Lang.* 8, PLDI, Article 210 (June 2024), 26 pages. https://doi.org/10.1145/3656440

Authors' addresses: Dongjun Youn, f52985@kaist.ac.kr; Wonho Shin, new170527@kaist.ac.kr; Jaehyun Lee, 99jaehyunlee@kaist.ac.kr; Sukyoung Ryu, sryu.cs@kaist.ac.kr, KAIST, Daejeon, South Korea; Joachim Breitner, mail@joachim-breitner.de, Independent, Freiburg, Germany; Philippa Gardner, p.gardner@imperial.ac.uk, Imperial College London, London, United Kingdom; Sam Lindley, Sam.Lindley@ed.ac.uk, The University of Edinburgh, Edinburgh, United Kingdom; Matija Pretnar, matija.pretnar@fmf.uni-lj.si, University of Ljubljana, Ljubljana, Slovenia; Xiaojia Rao, xiaojia.rao19@imperial.ac.uk, Imperial College London, London, United Kingdom; Conrad Watt, conrad.watt@cl.cam.ac.uk, University of Cambridge, Cambridge, United Kingdom; Andreas Rossberg, rossberg@mpi-sws.org, Independent, Munich, Germany.

ACM 2475-1421/2024/6-ART210
https://doi.org/10.1145/3656440

select

1. Assert: due to validation, a value of value type i32 is on the top of the stack.
2. Pop the value i32.const c from the stack.
3. Assert: due to validation, two more values (of the same value type) are on the top of the stack.
4. Pop the value val_2 from the stack.
5. Pop the value val_1 from the stack.
6. If c is not 0, then:
   a. Push the value val_1 back to the stack.
7. Else:
   a. Push the value val_2 back to the stack.

(a) Prose notation

$$val_1\ val_2\ (\text{i32.const}\ c)\ \text{select} \hookrightarrow val_1 \quad (\text{if } c \neq 0)$$
$$val_1\ val_2\ (\text{i32.const}\ c)\ \text{select} \hookrightarrow val_2 \quad (\text{if } c = 0)$$

(b) Formal notation

Fig. 1. Semantics of select in the Wasm 1.0 specification

```
rule Step_pure/select-true:
  val_1 val_2 (CONST I32 c) (SELECT) ~> val_1 -- if c =/= 0

rule Step_pure/select-false:
  val_1 val_2 (CONST I32 c) (SELECT) ~> val_2 -- otherwise
```

Fig. 2. Semantics of select expressed in Wasm SpecTec

1 INTRODUCTION

A programming language is defined by its syntax, static semantics (checking), and dynamic semantics (execution). Since they provide the foundation for all subsequent development and analysis, it is important to define them clearly and rigorously. Many programming languages, such as Java [51] or C# [47], therefore have elaborate language definitions. Languages that are prone to implementation inconsistency issues, such as C [30] and JavaScript [19], manage their standardization processes through international bodies. Few languages, like Standard ML [48], go even further and have formal specifications formulated in precise mathematical terms. However, it is an open problem how to continuously *engineer and maintain* such formal standards, especially at industrial scale.

WebAssembly (Wasm) is a low-level language and virtual machine [20]. Initially released as an efficient and portable compilation target on the Web platform, it has since been adopted across a wide range of ecosystems, including cloud and edge computing [28, 78], mobile and embedded systems [80], IoT [27], and blockchains [76]. Browsers implement Wasm within vendor-specific architectures using multi-tier interpretation or just-in-time compilation; in addition, there exist almost a dozen stand-alone implementations of Wasm. The portability of Wasm across all of these implementations is critically important, especially for Web developers, who have no control over what implementation ultimately executes their code. The diversity of environments and platforms means that there is a heightened risk of implementation divergence.

To mitigate these risks, Wasm has been standardized by the W3C [75] with particularly high rigor. It requires four key artifacts for a feature to be standardized: 1) a *formal specification* for the feature given by **declarative-style** typing and reduction rules, written in LaTeX; 2) a *prose pseudocode* presenting an **algorithmic-style** semantics, written in reStructuredText (reST) markup; 3) a *reference interpreter* providing an implementation of the feature, written in OCaml; and 4) a *test suite* for the feature, written in the Wasm text format (.wast). All of these artifacts must define, or for the test suite evaluate, the same behavior.

Notably, the specification provides *both* declarative and algorithmic styles of semantic definitions. For example, Fig. 1 shows the semantics of the select instruction in the official Wasm specification. Fig. 1a presents it in a prose notation broken down into seven steps, while Fig. 1b specifies it in a formal notation using two rewrite rules. The two styles enjoy complementary strengths: on the one hand, declarative rewrite rules specify the language semantics through rigorous and succinct

mathematical rules that enable faster turn-around in the design phase and are particularly well-suited to formal reasoning, including proving properties such as type safety; on the other hand, algorithmic prose pseudocode specifies the language semantics through a step-by-step natural language description that, though much more verbose, is comprehensible to a broader audience.

Despite the demanding standardization process, the Wasm specification has been written and maintained *manually*, which creates challenges for specification authors. The prose is transliterated from the formal rules, writing it is extremely laborious [62], and code reviews of specification changes written in LaTeX and reST are not user-friendly. Manual processes are vulnerable to human error, potentially leading to inconsistency or incorrectness in the specification. As Wasm grows with new language features such as garbage collection [64], exceptions [3], and threads [69], manually crafting all of the above artifacts poses a challenge to scalability.

These challenges call for automating the process via *mechanizing* the language semantics. The literature shows general-purpose language frameworks such as Ott [68], PLTRedex [22], Skeleton [67], Spoofax [74], and K [79]. However, the most successful examples of a mechanized *normative* specification involve mechanizations *tailored* to the specific target languages. This is because by narrowing attention to a specific language, a far more ambitious variety of mechanizations can be supported with ease. For example, ASL [60], which is singularly concerned with Arm's architectural specification, achieves far more impressive automation than is possible in a general-purpose framework. Taking a slightly different route, the ESMeta framework [1] reconstructs a mechanized specification from the JavaScript standard [19], allowing diverse tools [56, 55, 54] to be incorporated into the continuous integration checks of the specification and official test suite [73, 72].

We aim to develop a mechanized specification that will ultimately become the normative specification for Wasm, hence it is preferable to have a framework specialized for Wasm. But Wasm poses unique challenges. In order to replace the existing manual specification, it is required to generate all standardization artifacts in a format acceptable to the Wasm Community Group (CG). In particular, a framework is needed that can directly *support and match* both styles of semantics used in its specification (declarative and algorithmic), without requiring authors to *write* both. To the best of our knowledge, no framework in the literature directly supports both styles of semantics.

To this end, we propose Wasm *SpecTec*, a framework for mechanizing the Wasm semantics with these goals in mind. SpecTec provides a *domain-specific language (DSL)*, in which Wasm syntax, type system, and execution semantics can be defined in a declarative style. Fig. 2 illustrates this with the reduction rules defining the semantics of Wasm's select instruction, expressed in this DSL. Specification authors only need to write these two rules instead of what is seen in Fig. 1, which involves half a page of LaTeX and reST markdown sources not shown here. The definition in SpecTec then becomes a *single source of truth* from which various artifacts can be auto-generated:

(1) declarative representations, including the LaTeX-based specification and mechanized definitions for diverse theorem provers,
(2) algorithmic representations, such as the prose specification in reST and a Wasm interpreter, and
(3) executable tests in Wasm text format, to check conformance of third-party implementations.

This is made possible by leveraging the domain-specific knowledge present in a specialized tool.

Fig. 3 illustrates the overview of the overall SpecTec architecture. In this paper, we focus on the first stage of the project, which aims at generating those artifacts necessary for the published specification document, namely LaTeX and prose. We leave investigating the greyed-out parts of the diagram, which are concerned with generating test cases and facilitating the automated generation of mechanized semantics for Wasm in theorem provers, for future work.

The SpecTec DSL is designed to resemble familiar "textbook-style" notation for formally describing language semantics. We deliberately keep the syntax of the DSL close to the syntax used in the

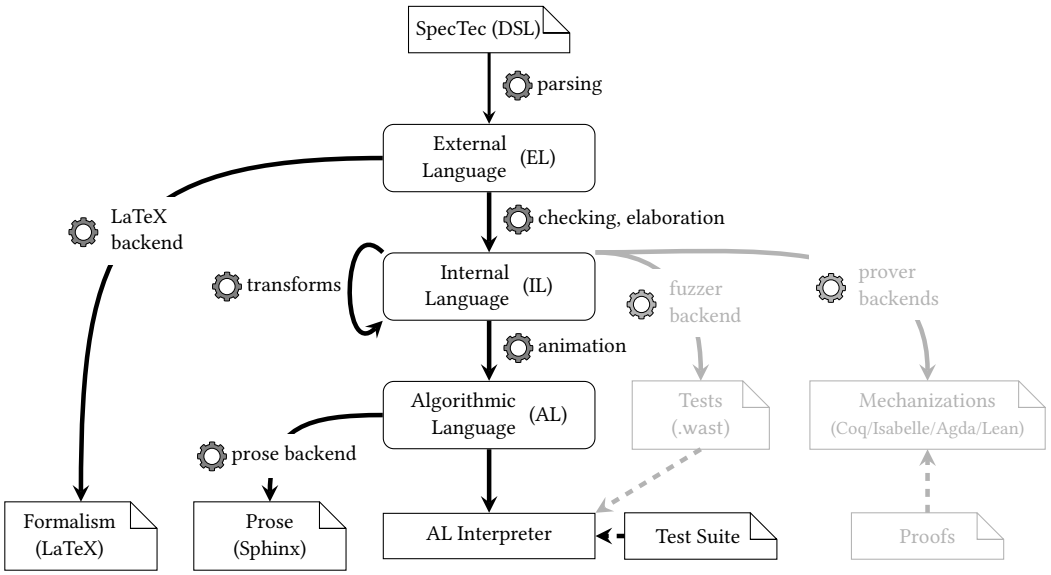

Fig. 3. An overview of the Wasm SpecTec architecture

existing formal specification of Wasm, in order to provide a sort of WYSIWYG experience to the specification authors. All definitions and rules are "type-checked" to detect meta-level errors. From this, SpecTec can directly produce the LaTeX-based formal specifications.

To generate the algorithmic representations, especially prose, SpecTec definitions are first translated — with a few intermediate steps — into an intermediate representation called the *Algorithmic Language (AL)*. It is designed to provide a representation that is close in structure to the prose of the official specification. Thus, the English prose in reST markup can be generated directly from the definitions in the AL. SpecTec also supports the interpretation of Wasm programs through a meta-level interpreter for the AL, following the approach of the ESMeta framework [1].

To bridge the gap between the two styles of semantics, we present a mechanism for automatically deriving the algorithmic AL from the declarative DSL. Several challenges arise in transforming declarative-style reduction rules into algorithmic-style pseudocode. For example, a single Wasm instruction may have multiple reduction rules with different premises (such as select in Fig. 2), but they must all be combined in a single prose pseudocode conditional. Moreover, mathematical equality '=' in rules can be interpreted as (unconditional) assignment or a (conditional) equality check, depending on the context (declaratively, there is no meaningful difference between the two, but algorithmically there is). We show that disambiguating such premises while introducing minimal auxiliary variables is an NP-hard problem and propose a practical solution.

SpecTec is available as an open-source project [2] and covers all of Wasm 2.0, the current release of Wasm. The prototype meta interpreter passes 100% of the official test suite and identified several specification errors in current and upcoming features, as confirmed by standards body members. In fact, we are up to speed with the development of the Wasm standard in the sense that we have extended the SpecTec specification with multiple major proposals soon to be merged into the standard, such as subtyping, garbage collection, and multiple memories. Our aim is for SpecTec to be adopted by the Wasm standards community and used to produce future versions of the standard.

Though the Wasm SpecTec design choices and tooling is specific to Wasm, we believe that the general methodology and architecture we present is applicable to other languages too.

The main technical contributions of this paper are the following:

- We design and implement **Wasm SpecTec**, a declarative specification language and its toolchain, optimized for reading, writing, and maintaining the Wasm specification (§2)
  - It is a framework embracing **both declarative and algorithmic styles** of semantics.
  - It automatically generates **multiple representations** from a single source of truth.
  - It is a **forward-compatible** framework that can adapt to the evolving Wasm semantics.
- We design and implement various backends for SpecTec:
  - a **LaTeX backend**, which outputs LaTeX code describing both static and dynamic semantics declaratively in the format used by the official Wasm specification (§2.4),
  - a **prose backend**, which outputs prose pseudocode in the style used by the official Wasm specification (§2.5 and §3.4), and
  - an **interpreter backend** that indirectly executes Wasm programs by meta-interpreting the specified semantics in its derived algorithmic representation (§3.5).
- We evaluate SpecTec by formalizing all of **Wasm 2.0**, generating a specification document and a meta-level interpreter from it, and validating them (§4).

2 FORMAL SEMANTICS IN SPECTEC

The SpecTec DSL is intended as the single source of truth for all representations of the Wasm semantics. As such, its design needs to strike a balance between the purposes of easy authoring of the official Wasm standard and its artifacts, as well as generating mechanizations for multiple theorem provers. Since the Wasm specification already exists with established idiosyncrasies that the community is familiar with, (1) SpecTec should have minimal impact on the resulting document, and (2) specification authors should not need to learn and understand yet another formalism or second-guess how a tool would translate that to the *intended* representation in the document.

Historically, the formal specification has usually been the first form in which the semantics of Wasm or a new feature has been specified (after informal "explainers"). The prose then was the result of a subsequent manual translation of the formal rules into stylized English. These dynamics are due to the conciseness of the declarative mathematics, which enables much faster turnaround, as well as its precision, which cannot always be adequately matched in prose.

SpecTec hence is designed to closely resemble the formal style used in the Wasm specification — which in turn is closely based on pen-and-paper notation used widely in literature and textbooks on programming language semantics. The main components of Wasm's formal specification are: (1) *formal grammars* defining *binary and textual representations*; (2) *deduction rules* defining the *type system*; (3) a *functional* definition of module *instantiation and linking*; (4) *small-step reduction rules* defining Wasm's *runtime execution*. For SpecTec, we deliberately decided against a type-theoretic notation that would be closer to theorem provers like Coq. The goal is a WYSIWYG user experience, where authors write formal rules (and code-reviewers see them) in the same notation as they will later appear on screen, except in a plain and readable (and diff'able) ASCII. To that end, SpecTec builds in much of the meta-level notation used by the Wasm specification, such as basic arithmetics, meta-variables, the notion of records, iterated sequences, and manipulation of these.

On the other hand, SpecTec does not hard-code any Wasm-specific concepts in its syntax or meta-theory (though its backends specialize for it to a varying degree, see §2.4 and §2.5). Instead, there are three generic mechanisms for describing the language and its semantics:[1]

(1) *Syntax* definitions, which can capture abstract syntax of a language or auxiliary constructs;
(2) *Relations* and respective rules, which can specify typing, evaluation, or other predicates;
(3) *Functions*, which enable auxiliary definitions.

[1]Also, there is a fourth mechanism for describing a concrete text syntax or a binary format, which we omit in this paper.

```
syntax valtype = I32 | I64 | F32 | F64
syntax functype = valtype* -> valtype*
syntax globaltype = MUT? valtype

syntax instr =
  | NOP
  | DROP
  | SELECT
  | CONST valtype const hint(show %.CONST %)
  | LOCAL.GET localidx
  | GLOBAL.GET globalidx
  | ...
```

(a) Abstract syntax

```
syntax context =
  { FUNCS functype*, GLOBALS globaltype*, ... }
```

(b) Syntax of contexts

```
relation Instr_ok: context |- instr : functype

rule Instr_ok/nop:
  C |- NOP : eps -> eps

rule Instr_ok/drop:
  C |- DROP : t -> eps

rule Instr_ok/select:
  C |- SELECT : t t I32 -> t

rule Instr_ok/const:
  C |- CONST t c : eps -> t

rule Instr_ok/local.get:
  C |- LOCAL.GET x : eps -> t
  -- if C.LOCALS[x] = t

rule Instr_ok/global.get:
  C |- GLOBAL.GET x : eps -> t
  -- if C.GLOBALS[x] = MUT? t
```

(c) Typing relation

```
syntax addr = nat
syntax moduleinst = { FUNCS addr*, GLOBALS addr*, ... }

syntax val = CONST valtype const | ...
syntax store = { FUNCS funcinst*, GLOBALS val*, ... }
syntax frame = { LOCALS val*, MODULE moduleinst }
syntax state = store; frame
syntax config = state; instr*
```

(d) Syntax of configurations

```
def $local(state, localidx) : val
def $local((s; f), x) = f.LOCALS[x]

def $global(state, globalidx) : val
def $global((s; f), x) = s.GLOBALS[f.MODULE.GLOBALS[x]]
```

(e) Auxiliary meta-functions

```
relation Step: config ~> config
relation Step_pure: instr* ~> instr*

rule Step/pure:
  z; instr* ~> z; instr'*
  -- Step_pure: instr* ~> instr'*

rule Step_pure/nop:
  NOP ~> eps

rule Step_pure/drop:
  val DROP ~> eps

rule Step_pure/select-true:
  val_1 val_2 (CONST I32 c) SELECT ~> val_1 -- if c =/= 0

rule Step_pure/select-false:
  val_1 val_2 (CONST I32 c) SELECT ~> val_2 -- otherwise

rule Step/local.get:
  z; (LOCAL.GET x) ~> z; val -- if $local(z, x) = val

rule Step/global.get:
  z; (GLOBAL.GET x) ~> z; val -- if $global(z, x) = val
```

(f) Reduction relation

Fig. 4. Wasm semantics in SpecTec (excerpt)

2.1 Wasm in SpecTec

Fig. 4 shows a small, simplified excerpt from the Wasm specification in SpecTec. Fig. 4a defines the abstract syntax of types and instructions: a value type is one of Wasm's numeric types; a function type consists of a sequence of value types for its parameters and another for its results, where the arrow is just custom meta-level notation; a global type can be marked with an optional mutability attribute; an instruction can be one of many forms, some of which have additional parameters representing the instruction's "immediates." Mnemonics (essentially, keywords) are written in all caps, distinguishing them from meta-variables.

To define a typing relation for instructions, we first need to define typing contexts (Fig. 4b). This again is a syntax definition, but this time describing a *record* of different namespaces. Each namespace simply contains a list of respective types — that is because, in Wasm, definitions are referenced by index (via a form of de Bruijn notation), not by names.

The typing relation itself (Fig. 4c) is defined over a triple of context, instruction, and a function type describing its effect on Wasm's operand stack. The stylized turnstile and colon used as separators are again just user-defined notation within SpecTec, chosen to make the relation more readable and reminiscent of standard paper notation. Each rule defining this relation then describes how to type one individual instruction: nop pops nothing from and pushes nothing to the stack (eps denotes the empty sequence ϵ); drop consumes a single value of arbitrary type (t is unrestricted in the rule); in contrast, select consumes three values, two of which are of arbitrary but same type t, and produces one t; the rules for local.get and global.get look up the result types in the context C, with the Boolean side conditions representing a form of *premises* to the rules.

Execution is defined as a small-step reduction relation over configurations (Fig. 4d); a configuration consists of the state and the instruction sequence to reduce. The state, in turn, consists of the global store and the local function frame; besides locals, the latter holds a reference to the current module instance, needed to resolve module-internal indices to global addresses in the store. In order to ease notation, we define a couple of short-hands for accessing locals and globals as auxiliary meta-functions (Fig. 4e). Meta-definitions are distinguished from meta-variables by a preceding $.

Finally, the actual reduction rules (Fig. 4f) are given by two stepping relations:[2] the first is the main relation that defines a step on configurations, while the second is a simpler variant used to define pure reductions that do not access the store. The rule Step/pure defines how they interact: we can take a step if we can take a pure step, which results in the state z being unchanged. The premise in this rule is not a side condition but inductively invokes another relation, which has to be named explicitly. The nop, drop, and select instructions are pure so that they can be formulated with this simpler relation: nop does nothing, hence is reduced to the empty instruction sequence; drop likewise, but it also eliminates a *value* instruction before it, which amounts to popping a value from the stack; select is defined by two cases with different side conditions inspecting the selector value c, and the side condition "**otherwise**" is a short-hand negating all previous premises for a rule with the same left-hand side; the rules for local.get and global.get look up the value in the respective part of the state, invoking the auxiliary meta-functions previously defined.

Following this style, we have translated most of the technical content of the recent Wasm 2.0 specification [20] to SpecTec,[3] as well as substantial newer features, such as garbage collection [64], multiple memories [65], and tail calls [66] (see §4.3). This translation covers (1) abstract syntax, (2) validation, (3) runtime system, (4) execution, (5) module instantiation, and (6) binary format. The main pieces still left for future work are numeric primitives and the text format. By design, the SpecTec tooling allows us to convert the existing specification *progressively*, that is, different parts of it can be replaced with SpecTec-generated content incrementally.

2.2 The SpecTec Language

Fig. 5 presents the syntax of SpecTec definitions. *Expressions* are either Boolean or arithmetic formulas, applications of meta-functions, *sequences* of expressions (where **eps** denotes the empty sequence), indexing into a (homogeneous) sequence, taking the length of a sequence, records, record projection, record and sequence updates, tuples, or user-defined *notation*. The latter consists of sequences of expressions interspersed with *atoms*, which are either upper-case identifiers or one of a set of predefined symbolic tokens. *Types* describe the shape of expressions and can describe either Booleans or natural numbers, sequences, records, variants, tuples, or user-defined notation.

In both cases, homogeneous sequences of variable length can be denoted by *iterations*, similar to regular expressions. For example, nat$*$ would be the type of possibly empty sequences of natural

[2]We are omitting the rule for finding the redex in a longer instruction sequence.

[3]Of course, SpecTec is only concerned with normative parts of the specification. Editorial contents like section introductions and explanatory notes are still authored manually and largely left unchanged. They can, however, refer to SpecTec definitions.

Identifier	*id*	::=	…lower-case-identifier…
Atom	*atom*	::=	…upper-case-identifier… | **–>** | **~>** | **:** | **|–** | …
Operator	*unop*	::=	**~** | **+** | **–**
	binop	::=	**/** | **\/** | **<=>** | **+** | **–** | ***** | **/** | **^** | **=** | **=/=** | **<** | **<=** | **>=** | **>** | …
Iteration	*iter*	::=	**?** | ***** | **+** | **^***exp*
Type	*typ*	::=	*id* | **bool** | **nat** | *typ typ* | *typ iter* | *atom* | **{** (*atom typ*,)* **}** | (| *atom typ*)$^+$ | **(** *typ*,* **)** | …
Expression	*exp*	::=	*id* | *bool* | *num* | *unop exp* | *exp binop exp* | **$***id exp*$^?$ | **eps** | *exp exp* | *exp iter* | *exp*[*exp*]
		|	|*exp*| | *atom* | **{** (*atom exp*,)* **}** | *exp*.*atom* | *exp*[*path* **=** *exp*] | **(** *exp*,* **)** | …
Path	*path*	::=	*path*$^?$[*exp*] | *path*$^?$.*atom* | …
Definition	*def*	::=	**syntax** *id* **=** *typ* | **relation** *id* **:** *typ* | **rule** *id* (**/** *id*)* **:** *exp* (**--** *prem*)*
		|	**def $***id exp*$^?$ **:** *typ* | **def $***id exp*$^?$ **=** *exp* (**--** *prem*)* | **var** *id* **:** *typ*
Premise	*prem*	::=	*id* **:** *exp* | **if** *exp* | **otherwise** | *prem iter*

Fig. 5. SpecTec syntax (excerpt)

numbers, while `nat^3` specifies a sequence of exactly three numbers. In an expression, the use of a variable under an iteration implies that this variable has a respective *dimension*; SpecTec infers the dimension of each variable and checks that it is used consistently when the same variable occurs multiple times in a rule or definition. A variable can occur as part of a larger expression under an iteration, which is a mapping over that variable. For example, `list = {A x, B y}* /\ (x < 100)*` states that `list` is a sequence of records with fields A and B, and every value `x` of respective field A must be smaller than 100. This corresponds to the use of overbars in formal notation [70], and like overbars, iterations can nest freely, leading to variables of higher dimension.

Syntax definitions are essentially type definitions: they name a given type.[4] Besides creating a shorthand, this also enables (mutual) recursion, by which syntax definitions can be interpreted as inductive types. Relations are declared with a type that specifies its notation, which will typically consist of a sequence of syntax types separated by atoms. Each corresponding rule then consists of an expression of the respective type, possibly accompanied by a sequence of *premises*. In its basic form, a premise can either invoke another relation, in which case the name of that relation has to be given along with an expression of a suitable type, or it is a Boolean side condition. A special premise is **otherwise**, which represents the negation of all premises previously used for the same left-hand side. Premises can also be iterated. All rules must be uniquely named, and these names can be hierarchical; the names have no semantic relevance, but allow referring to individual rules in tooling (§2.4). Function definitions are given by a declaration of their type and then individual equational clauses. The clauses express a (sequential) pattern match, with their left-hand side argument expression being the pattern and possible premises acting as pattern guards. Like syntax, relations and functions are all interpreted as inductive definitions. Finally, variable declarations allow us to globally declare the type implicitly associated with each use of a meta-variable. SpecTec recognizes suffixes as in `x_1` or `x'` as variations with the same type. In addition to explicit variable declarations, syntax definitions implicitly declare a variable of the same name as the type. All variables used in expressions must be declared in one of these two ways.

2.3 Elaboration and Lowering

SpecTec definitions are first parsed into a representation called the *External Language (EL)*. This representation is an AST that corresponds directly to the surface syntax of SpecTec, as presented in Fig. 5. On this representation, the frontend then performs binding and recursion analysis, type checking, dimension inference, and type-driven resolution of notational overloading in expressions. During this procedure, the EL is *elaborated* into a more explicit and unambiguous *Internal Language*

[4] In the actual SpecTec syntax, certain forms of types, like records, variants, or custom notation, cannot be written inline but must be named. We have simplified the syntax here for presentational purposes.

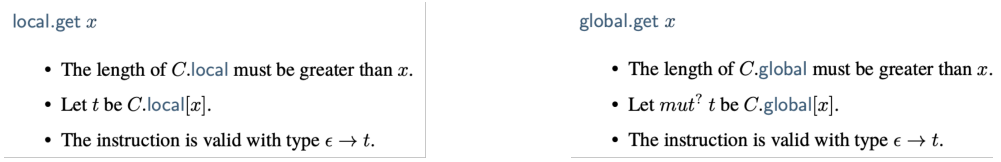

Fig. 6. Validation of local.get and global.get in a generated prose specification

(IL). This representation groups definitions and makes all binders, types, dimensions, recursions, and uses of subsumption explicit. It also annotates every iteration with the variables it iterates over.

From here, the middlend takes over and runs a number of simplifying transformations on the IL:

- Infer implicit *side conditions*, such as the fact that the use of s[i] in a premise requires i < |s|, or that joint iterations like (x < y + 1)* require |x*| = |y*|.
- Lift the result type of *partial* functions (whose clauses are not exhaustive) into options, i.e., iterations t?, and adjust their use sites accordingly.
- Introduce auxiliary variables for option types without variable content, like ATOM?, in order to turn them into Booleans later.
- Turn subsumption (e.g., on variant types) into explicit *injections*.
- Perform *dataflow analysis* on premises, annotate suitable equations as variable bindings, and reorder premises accordingly (§3.3).

Not all these transformations are run for every backend; each backend can select them individually.

2.4 LaTeX Backend

The simplest of all backends is the LaTeX backend. It produces LaTeX directly from the EL, bypassing the IL, giving users of SpecTec fairly direct control over the output. Most of it is straightforward, except for the rendering of identifiers, which distinguishes different identifier classes and can handle subscripts of mixed classes. The main complication is handling rendering hints like "**hint**(show ...)" in the DSL (omitted from Fig. 5), which allows customizing the rendering of selected identifiers, atoms, and whole function invocations. Moreover, it distinguishes forms of relations based on their syntactic shape and renders them as either typing or reduction relations accordingly.

The LaTeX backend operates on secondary input in the form of text files with *splice commands*. For example, it replaces the splice @@{**rule**: Instr_ok/local.get Instr_ok/global.get} with a single LaTeX array containing the corresponding inference rules:

$$\frac{C.\mathsf{local}[x] = t}{C \vdash \mathsf{local.get}\ x : \epsilon \to t} \qquad \frac{C.\mathsf{global}[x] = \mathsf{mut}^?\ t}{C \vdash \mathsf{global.get}\ x : \epsilon \to t}$$

The splice @@{**rule**: Step_pure/select-*}, on the other hand, recognizes Step_pure as a reduction relation and renders the rules matched by the wildcard appropriately:

$$\begin{array}{lcll} val_1\ val_2\ (\mathsf{i32.const}\ c)\ \mathsf{select} & \hookrightarrow & val_1 & \text{if } c \neq 0 \\ val_1\ val_2\ (\mathsf{i32.const}\ c)\ \mathsf{select} & \hookrightarrow & val_2 & \text{otherwise} \end{array}$$

An overall specification document hence is a skeleton (in LaTeX, reST, or another format) with respective splices, that is transformed into the final document by the SpecTec tool.

2.5 Prose Backend

The prose backend covers validation, runtime system, execution, and module instantiation; it does not apply to abstract syntax and binary format, which do not have corresponding prose. Like the LaTeX backend, it can process splice commands and distinguishes forms of relations based on their syntactic shape and handles them accordingly.

Semantics	δ^*		Definition	δ	$::= \rho \mid \lambda$
Reduction rule	ρ	$::= \gamma \rightsquigarrow \gamma - \pi^*$	Configuration	γ	$::= (\eta^?,\ \eta)$
Premise	π	$::= \mathit{prem} \mid \eta \leftarrow \eta$	Function	λ	$::= x(\eta^*) = \eta - \pi^*$
Expression	η	$::= \mathit{expr} \mid \kappa\ \eta^* \mid \cdots$	Constructor	κ	$::= \mathit{atom}$

Fig. 7. Syntax of DL

Because the Wasm specification's prose description of the validation rules is still declarative, generating it from the EL is relatively straightforward: Fig. 6 shows the result for the local.get and global.get instructions.

In contrast, generating the English prose specification for execution (and some auxiliary meta-functions) is not trivial because it requires conversion to an algorithmic formulation. Moreover, it assumes that reduction rules can be interpreted as a suitable form of stack machine. We explain the challenges and our solution in the next section.

3 ALGORITHMIC BACKENDS

SpecTec automates the generation of algorithmic representations from formal Wasm semantics. The process involves translating IL (§2.3) to an *Algorithmic Language (AL)* (§3.2 and §3.3), generating prose specifications from AL (§3.4), or interpreting AL to indirectly execute Wasm programs (§3.5).

3.1 DL: Declarative Language

For presentational purposes, we abstract the SpecTec syntax in Fig. 5 into a *Declarative Language (DL)* that only shows the features relevant to the translation of Wasm's runtime semantics into an algorithmic representation. Fig. 7 presents the syntax of DL.

The Wasm runtime semantics is defined by a sequence of definitions δ^*. A definition is either a reduction rule ρ or an auxiliary helper function λ. A reduction rule $\gamma_1 \rightsquigarrow \gamma_2 - \pi^*$ is an abstraction for the specific subset of **rule**s from Fig. 5, whose role is to express Wasm reduction rules as illustrated in Fig. 4f. Note that we only abstract the reduction rules but not typing rules in Fig. 4c, since the prose notation of typing rules are not algorithmic, and can not be executed. A configuration $(\eta^?,\ \eta)$ denotes a pair of Wasm program state $\eta^?$ which can be omitted (written ϵ), and a list of Wasm instructions η that represents the current stack. If $\eta^?$ is omitted, it indicates that the corresponding reduction rule is a simpler Step_pure rule. A premise π is a premise *prem* extended with the special case $\eta \leftarrow \eta$, denoting an explicit variable binding. A helper function $x(\eta^*) = \eta - \pi^*$ is an abstraction for the function definitions **def**s from Fig. 5. Because the details of the expression are not relevant to this section, we show only some cases used for concrete examples. A constructor κ is an atom that denotes the name of Wasm types, such as I32, or Wasm instructions, such as CONST and SELECT.

For example, the semantics of select in Fig. 2 corresponds to the following in DL:

$$[(\epsilon, [v_1, v_2, (\text{CONST I32 } c), (\text{SELECT})]) \rightsquigarrow (\epsilon, [v_1]) - [c \neq 0],$$
$$(\epsilon, [v_1, v_2, (\text{CONST I32 } c), (\text{SELECT})]) \rightsquigarrow (\epsilon, [v_2]) - [\text{otherwise}]]$$

To avoid clutter, we will sometimes omit brackets around singular lists.

3.2 AL: Algorithmic Language

The Algorithmic Language (AL) represents Wasm's runtime semantics in an imperative style. DL definitions are translated into AL definitions, such that pseudocode representations can be generated.

Fig. 8 presents the core syntax of AL. The metavariable x ranges over variables, n ranges over numbers, and k ranges over constructors. An AL program p is a sequence of algorithms a^*, each of which denotes a group of reduction rules for a single Wasm instruction, or an auxiliary helper

Program	$p ::= a^*$		
Algorithm	$a ::= \text{algorithm } x(e^*)\ s^*$		
Statement	$s ::= \text{if } c\ s^*\ s^*$	If c, then: s_1^* Else: s_2^*	
	$\mid \text{enter } e\ e\ s^*$	Enter e_1 with $e_2 : s^*$.	
	$\mid \text{exit}$	Exit current context.	
	$\mid \text{assert } c$	Assert: c.	
	$\mid \text{push } e$	Push e to the stack.	
	$\mid \text{pop } e$	Pop e from the stack.	
	$\mid \text{let } e\ e$	Let e_1 be e_2.	
	$\mid \text{trap}$	Trap.	
	$\mid \text{nop}$	Do nothing.	
	$\mid \text{return } e^?$	Return $e^?$.	
	$\mid \text{execute } e$	Execute e.	
	$\mid \text{perform } x\ e^*$	Perform $x(e^*)$.	
	$\mid \text{replace } e\ q^*\ e$	Replace $e_1[q^*]$ with e_2.	

Condition	$c ::= \text{not } c \mid c \oplus c \mid \text{iscase } e\ x \mid w$		
Expression	$e ::= x$	x	
	$\mid n$	n	
	$\mid \oplus e$	$\oplus\ e$	
	$\mid e \oplus e$	$e_1 \oplus e_2$	
	$\mid e[q]$	$e[q]$	
	$\mid e[q^* := e]$	e_1 with $q^* = e_2$	
	$\mid \{(x \mapsto e)^*\}$	$\{\ (x :\ e)^*\ \}$	
	$\mid [e^*]$	e^*	
	$\mid e\ ++\ e$	$e_1\ e_2$	
	$\mid \lvert e \rvert$	length of e	
	$\mid (k\ e^*)$	$(k\ e^*)$	
	$\mid x(e^*)$	$x(e^*)$	
	$\mid w$		
Path	$q ::= e \mid .x$		

Fig. 8. Syntax of AL and its prose notation (excerpt)

function, in imperative form. An algorithm consists of a name x, parameters e^*, and a body of meta-level statements s^*. A statement s denotes a prose statement in the Wasm specification. An expression e denotes an expression in the prose that is evaluated to a value, and a condition c denotes a condition in the prose that is evaluated to a Boolean value. The figure also indicates the intended prose rendering of statements and expressions. The Wasm specification often uses specific phrases like "the current frame" and "a label is now on the top of the stack." We abstract such Wasm-specific expressions and conditions as w for brevity.

For example, the semantics of select in Fig. 2 corresponds to the following in AL:

```
algorithm SELECT() [ assert "an I32 value is on the top of the stack", pop (CONST I32 c),
                     assert "a value is on the top of the stack", pop v2,
                     assert "a value is on the top of the stack", pop v1,
                     if (c ≠ 0) (push v1) (push v2) ]
```

The semantics of AL is imperative. The interpretation of a program p starts by calling one of its algorithms, algorithm $x(e^*)\ s^*$, which sequentially executes its body statements s^*. Executing a statement s may alter the implicit program state, and the resulting state after executing the last statement of the algorithm is the result of the program execution.

3.3 DL to AL Translation

Now, we describe how to translate a Wasm semantics δ^* in DL into an AL program p. First, the definitions in δ^* are grouped to represent algorithms; among δ^*, the reduction rules γ^* are grouped according to the Wasm instruction in their redex, and the helper functions λ^* are grouped according to their names. Each group is translated into a single algorithm in two phases: (1) preprocess the group's definitions into a more restricted form, and (2) generate an AL algorithm from the preprocessed DL definitions. Since the translation of helper functions is similar to the translation of reduction rules, this section focuses on the translation of reduction rules.

3.3.1 DL to DL Preprocessing. Preprocessing consists of two steps: for each group of reduction rules, (1) preprocess the left-hand sides of the reduction rules in the group to make them the same, and (2) preprocess the premises of the definitions in the group so that every variable is bound exactly once before its uses.

The first preprocessing step is to generalize the left-hand sides of the reduction rules so that the left-hand sides in each group become identical. Most Wasm definitions satisfy this condition, but some do not, such as the following inductive reduction rules for the br instruction:

```
rule Step_pure/br-zero: (LABEL n `{instr'*} val'* val^n (BR 0) instr*)  ~>  val^n instr'*
rule Step_pure/br-succ: (LABEL n `{instr'*} val* (BR $(l+1)) instr*) ~>  val* (BR l)
```

which corresponds to the following in DL:

$$\rho_1 = (\epsilon, (\text{LABEL}\ n\ i'^*\ (v'^* {+}{+} v^n {+}{+} (\text{BR}\ 0) {+}{+} i^*))) \rightsquigarrow (\epsilon,\ v^n {+}{+} i'^*) \quad -\ \epsilon$$
$$\rho_2 = (\epsilon, (\text{LABEL}\ n\ i'^*\ (v^* {+}{+} (\text{BR}\ (l+1)) {+}{+} i^*))) \rightsquigarrow (\epsilon,\ v^* {+}{+} (\text{BR}\ l)) - \epsilon$$

This process is known as *anti-unification* [15]. For each group of reduction rules, the anti-unification algorithm $\mathcal{AU}$ takes a list of left-hand side expressions as input. For the br instruction, for example, $\mathcal{AU}$ takes two expressions:

$$\eta_1 = (\text{LABEL}\ n\ i'^*\ (v'^* {+}{+} v^n {+}{+} (\text{BR}\ 0) {+}{+} i^*))$$
$$\eta_2 = (\text{LABEL}\ n\ i'^*\ (v^* {+}{+} (\text{BR}\ (l+1)) {+}{+} i^*))$$

For a list of expressions to anti-unify $\eta_1\ \cdots\ \eta_n$, the algorithm $\mathcal{AU}$ (1) generates a general expression η possibly containing some fresh variables and then (2) generates premises π_k^* $(1 \leq k \leq n)$ using the fresh variables to make each η_k be the same as η with π_k^*. More specifically, the general expression η is the most specific expression that generalizes the expressions $\eta_1\ \cdots\ \eta_n$, replacing only the different components with fresh variables. For example, the general expression for the reduction rules for br is $(\text{LABEL}\ n\ i'^*\ (y_1 {+}{+} (\text{BR}\ y_2) {+}{+} i^*))$ with two fresh variables y_1 and y_2. After generating the general expression, $\mathcal{AU}$ generates premises π_k^* for each η_k so that η_k is an instance of the general expression satisfying π_k^*. For example, to make the general expression for br same as the left-hand sides of ρ_1 and ρ_2, $\mathcal{AU}$ infers the conditions $[y_1 = v'^* {+}{+} v^n,\ y_2 = 0]$ and $[y_1 = v^*,\ y_2 = l{+}1]$, respectively. Finally, two rules for the br instruction become:

$$\rho_1 = (\epsilon, (\text{LABEL}\ n\ i'^*\ (y_1 {+}{+} (\text{BR}\ y_2) {+}{+} i^*))) \rightsquigarrow (\epsilon, v^n {+}{+} i^*) \quad -\ [y_1 = v'^* {+}{+} v^n,\ y_2 = 0]$$
$$\rho_2 = (\epsilon, (\text{LABEL}\ n\ i'^*\ (y_1 {+}{+} (\text{BR}\ y_2) {+}{+} i^*))) \rightsquigarrow (\epsilon, v^* {+}{+} (\text{BR}\ l)) - [y_1 = v^*,\ y_2 = l+1]$$

Thanks to the fresh variables y_1 and y_2, both ρ_1 and ρ_2 have the same left-hand sides. The definitions of y_1 and y_2 are in premises: $[y_1 = v'^* {+}{+} v^n,\ y_2 = 0]$ for ρ_1 and $[y_1 = v^*,\ y_2 = l+1]$ for ρ_2.

The second preprocessing step is to change the premises of each group's definitions so that each variable is bound exactly once before it is used. This step is necessary because the order of premises in declarative reduction rules can be arbitrary. In addition, equality expressions in premises can be ambiguous because they can represent equality check conditions or variable bindings. Thus, the goal of this second preprocessing step is to replace every equality premise denoting a variable binding with $\eta \leftarrow \eta'$. In order to do that, we need to identify each variable's binding occurrence, keep track of the variables that each premise binds, and reorder premises so that preceding premises bind all free variables in each premise.

For example, consider the first rule for the br instruction again:

$$\rho_1 = (\epsilon, (\text{LABEL}\ n\ i'^*\ (y_1 {+}{+} (\text{BR}\ y_2) {+}{+} i^*))) \rightsquigarrow (\epsilon, v^n {+}{+} i^*) - [y_1 = v'^* {+}{+} v^n,\ y_2 = 0]$$

The second premise $y_2 = 0$ is an equality check condition, while the first premise is a binding of fresh variables v and v'. Therefore, this step changes the rule as follows:

$$\rho_1 = (\epsilon, (\text{LABEL}\ n\ i'^*\ (y_1 {+}{+} (\text{BR}\ y_2) {+}{+} i^*))) \rightsquigarrow (\epsilon, v^n {+}{+} i^*) - [v'^* {+}{+} v^n \leftarrow y_1,\ y_2 = 0]$$

which clearly indicates that fresh variables v and v' are newly introduced in the first premise. Once the variable binding occurrences in premises are identified, reordering the premises is a simple def-use dataflow analysis.

The primary challenge now lies in identifying each variable's binding occurrence in premises. An intriguing difficulty emerges because of *partial binding*, in which certain free variables on one side of an equality expression are binding occurrences, but others are not. In a premise $(x_1, x_2) = y$, for example, only x_1 might be a binding occurrence but not x_2. Unfortunately, such partial bindings introduce fresh variables while generating prose. The prose rendering of the premise $(x_1, x_2) = y$ would be "1. Let (x_1, t) be y. 2. If $t = x_2$, then: $\cdots$", introducing the free variable t. Overuse of such fresh variables may result in a prose specification that is very different from the current prose specification, leading to a less readable document.

Therefore, it is preferable to minimize the number of partial bindings when identifying the binding occurrences. It turns out that this is actually an *NP-hard problem*. We prove it by reduction from a known NP-hard problem, the *exact cover* problem [46]:

> The exact cover problem aims at deciding whether it is possible to select some subsets within a given collection of subsets in such a way that each element of a given set belongs to exactly one selected subset. This problem is NP-complete [36].

Here, we provide the formal definition of the exact cover problem:

Definition 3.1 (Exact Cover Problem). An instance of the *Exact Cover Problem* (EC) is defined by a tuple (X, S) such that X is a set of elements and $S \subseteq \mathcal{P}(X)$ is a collection of subsets of X. EC aims at deciding if there exists a subcollection $P \subseteq S$ which is a partition of X, that is, $\forall x \in X.\ |\{s \in P \mid x \in s\}| = 1$.

For example, consider $X = \{a, b, c, d, e\}$ and $S = \{\{a, b\}, \{b, c\}, \{c, d, e\}\}$. Because $\{\{a, b\}, \{c, d, e\}\}$, one of the subcollections of S, is a partition of X, the answer is yes. On the other hand, for $S' = \{\{a, b\}, \{b, c\}, \{c, d\}, \{d, e\}\}$, since no subcollection of S' is a partition of X, the answer is no.

Because EC is one of Karp's 21 NP-complete problems [36], we can prove that the problem of identifying each variable's binding occurrence in premises is NP-hard, if we can reduce EC into the problem in polynomial time.

Theorem 3.2. *The problem of identifying each variable's binding occurrence in premises with the minimal partial binding is NP-hard.*

Proof. Assume that we are given EC with a set X and a collection of its subsets $S \subseteq \mathcal{P}(X)$. Let n be the size of S. Let $S_i = \{x_{i1}, ..., x_{ij}\}$ be the i-th subset of S. Now, consider a reduction rule $\gamma \rightsquigarrow \gamma' - \pi^*$ where $\gamma = (\epsilon,\ [v_n, \cdots, v_1])$, $\gamma' = (\epsilon,\ [])$, and the i-th premise of π^* be $v_i = (x_{i1}, ..., x_{ij})$. Note that this reduction rule can be constructed in linear time. The claim is that when we identify each variable's binding occurrence in π^* with the minimal number of partial binding, this gives a solution to EC. Specifically, the answer to EC is YES if and only if the minimum number is zero. Note that every variable must be bound exactly once. Thus, given that there is no partial binding, when we collect all premises that bind new variables, then the subsets S_i corresponding to such premises would form a subcollection of S, which is a partition of X. Conversely, if an exact cover exists for the given collection, then the identification of binding occurrences can be done without any partial bindings, by regarding every variable in the premises that corresponds to the sets in the exact cover as a binding occurrence, and regarding anything else as not. □

Example. Let's consider the example of $X = \{a, b, c, d, e\}$ and $S = \{\{a, b\}, \{b, c\}, \{c, d, e\}\}$ again. Following the proof above, EC for X and S is reduced to the problem of identifying each variable's binding occurrence in the premises of the following reduction rule:

$$\rho = (\epsilon, [v_3, v_2, v_1]) \rightsquigarrow (\epsilon, []) - [v_1 = (a, b), v_2 = (b, c), v_3 = (c, d, e)]$$

If we successfully solve the problem, then the result should look like the following:

$$\rho' = (\epsilon, [v_3, v_2, v_1]) \rightsquigarrow (\epsilon, []) - [(a, b) \leftarrow v_1, (c, d, e) \leftarrow v_3, v_2 = (b, c)]$$

Algorithm 1: Preprocess Premises

Input: A list of premises π^* and a set of bound variables V
Output: A list of preprocessed premises π'^*

$S \leftarrow \emptyset$
for $i \leftarrow 1$ ***to*** $|\pi^*|$ **do**
  if $\pi_i = (\eta = \eta')$ **then** $S_i \leftarrow \{\{i\},\ \{i\} \cup \mathit{free}(\eta),\ \{i\} \cup \mathit{free}(\eta')\}$
  else $S_i \leftarrow \{\{i\}\}$
  $S \leftarrow S \cup S_i$
$P \leftarrow \mathit{Knuth}(S \cup \{X\})$
for $p \in P$ **do**
  $\pi'^* \leftarrow \mathit{replaceVariableBinding}(\pi^*, p)$
  if *reordering* π'^* *succeeds* **then return** *reordered* π'^*
where $\mathit{free}(\eta)$ returns free variables in η and $\mathit{Knuth}(S)$ returns partitions of $\bigcup S$

which does not have any partial bindings. From the result, we can reconstruct the partition of the set X by collecting the binding premises, $\{\{a, b\}, \{c, d, e\}\}$, giving the answer to EC.

Thus, no polynomial-time algorithm can solve the problem. If the numbers of premises are small, a simple brute-force algorithm might be a solution. However, the reduction rule for module instantiation, for example, has more than 10 premises and variables, so a more efficient method is preferrable. Another solution is to use an SMT solver like Z3 [17], since this is a constraint solving problem. However, encoding the problem as an SMT query and invoking the Z3 solver from our implementation can be an unnecessary overhead.

As a practical solution to this NP-hard problem, we adopt an all-or-nothing heuristic approach. We first try to solve the problem under a condition that partial bindings are not allowed, and if that fails, we resort to a greedy algorithm, inevitably introducing some fresh variables. In order to solve the problem with the constraint of no-partial bindings, we reduce the problem into EC and then adopt the Knuth algorithm [39], u well-known and effective algorithm for solving EC. The high-level idea is to encode the premises as a collection of sets, where a solution to EC of the encoded collection corresponds to the solution to our problem.

Algorithm 1 describes the process. It takes two inputs, a list of premises π^* in reduction rules and a set of already bound variables V, and returns a list of new premises π'^*. First, it encodes premises as a collection of subsets of X, where $X = \{1, \cdots, |\pi^*|\} \cup \mathit{free}(\pi^*)$ is a set of the numbers from 1 to the size of premises and all free variables in them, on which the Knuth algorithm performs. For each π_i, if it is an equality expression $\eta = \eta'$, it is encoded as three subsets: $\{\{i\},\ \{i\} \cup \mathit{free}(\eta),\ \{i\} \cup \mathit{free}(\eta')\}$, which denotes its possible interpretations. The first set $\{i\}$ denotes when the i-th premise does not bind any variables, meaning π_i is an equality check condition. The second set $\{i\} \cup \mathit{free}(\eta)$ denotes when π_i binds ALL the variables in η and the third set $\{i\} \cup \mathit{free}(\eta')$ denotes when π_i binds ALL the variables in η'. For example, if the first premise is $x = y$, it is encoded as three subsets: $\{1\}$, $\{1, x\}$, and $\{1, y\}$. If π_i is not an equality expression, it is encoded as only one subset, $\{i\}$, meaning π_i does not bind any variables but checks some non-equality condition. Then, the Knuth algorithm takes the collection S containing all the encoded subsets and the set of bound variables V and returns the partitions P of the set X. Note that the Knuth algorithm may not return a unique partition. In addition, due to the definition of partition, for any partition p, there should be exactly one subset that contains a number i for $1 \leq i \leq n$. Thanks to the design of encoding, a subset is either a singleton set $\{i\}$ or a set with an index and some variables $\{i, x_1, x_2, \cdots\}$.

Algorithm 2: Replace Variable Binding

Input: A list of premises π^* and a partition p
Output: A list of premises π'^* with explicit variable binding

```
Function replaceVariableBinding(π*, p)
    for i ← 1 to |π*| do
        if π_i = (η = η') then
            if {i} ∈ p then π'_i ← (η = η')
            else if {i} ∪ free(η) ∈ p then π'_i ← (η ← η')
            else if {i} ∪ free(η') ∈ p then π'_i ← (η' ← η)
        else π'_i ← π_i
    return π'*
```

Using a partition p, the algorithm replaces every equality expression denoting a variable binding with an explicit variable binding expression via *replaceVariableBinding* in Algorithm 2. For example, if a singleton set $\{2\}$ is in the input partition p, then the second premise π_2 is a condition and therefore remains the same. If a set $\{3, y\}$ is in p and the third premise is $x = y$, then the third premise is replaced with $y \leftarrow x$. For preprocessed premises π'^*, Algorithm 1 tries to reorder them so that all variables are bound before their uses. Reordering premises may fail, if they contain cyclic bindings like $x \leftarrow f(y)$ and $y \leftarrow g(x)$. If reordering π'^* succeeds, the algorithm returns the reordered π'^*; otherwise, it tries with the next partition. If the reordering fails for every possible partition, the translation from DL to AL fails. There were two such cases in the Wasm spec for helper functions regarding module instantiation, instantiate and allocmodule. We re-wrote them to have equivalent non-circular definitions.

3.3.2 AL Generation. After preprocessing, DL definitions satisfy two conditions: (1) the left-hand sides of the reduction rules in each group are identical and (2) every variable in premises is bound exactly once before it is used. For each group of reduction rules $\rho^* = (\eta_1, \eta_1') \rightsquigarrow \gamma_1 - \pi_1^* \cdots (\eta_n, \eta_n') \rightsquigarrow \gamma_n - \pi_n^*$, the translation algorithm $\mathcal{T}$ generates an AL algorithm by translating the left-hand-side (η_1, η_1') first, and then translating the premise π_i and right-hand-side γ_i for each rule $\rho_i (1 \le i \le n)$. For the rules for select in DL, for example:

$$[(\epsilon, [v_1, v_2, (\text{CONST I32 } c), (\text{SELECT})]) \rightsquigarrow (\epsilon, [v_1]) - [c \neq 0],$$
$$(\epsilon, [v_1, v_2, (\text{CONST I32 } c), (\text{SELECT})]) \rightsquigarrow (\epsilon, [v_2]) - [\text{otherwise}]]$$

the identical left-hand-side $(\epsilon, [v_1, v_2, (\text{CONST I32}; c)(\text{SELECT})])$ is translated to the list of statements, [assert w_1, pop (CONST I32 c), ...]. For the first rule ρ_1, the premise $c \neq 0$ and the right-hand-side $(\epsilon, [v_1])$ are translated to if $(c \neq 0)$ and push v_1, respectively.

Roughly speaking, translating a left-hand-side of a reduction rule corresponds to generating the beginning of an algorithm, which pops the values from the stack to use as the inputs for the target instruction and binds new variables that contain the information about the inputs. Translation of premises corresponds to generation of the middle of an algorithm, which generates either variable bindings let e e or if statements with conditions c. Note that a binding premise $\eta \leftarrow \eta'$ may introduce side conditions. For example, a binding $[x, y] \leftarrow arr$ introduces a side condition that the size of the array arr is two. $\mathcal{T}$ generates such side conditions based on binding patterns. Finally, translation of a right-hand-side of a reduction rule corresponds to generation of the end of an algorithm, which pushes the result value onto the stack top or execute other Wasm instructions.

select

1. Assert: Due to validation, a value of value type i32 is on the top of the stack.
2. Pop i32.const c from the stack.
3. Assert: Due to validation, a value is on the top of the stack.
4. Pop val_2 from the stack.
5. Assert: Due to validation, a value is on the top of the stack.
6. Pop val_1 from the stack.
7. If c is not 0, then:
    a. Push val_1 to the stack.
8. Else:
    a. Push val_2 to the stack.

Fig. 9. Execution semantics of select in a generated prose specification

3.4 Prose Backend

As AL is designed to resemble pseudocode, generating an English prose specification from AL algorithms is a straightforward task. Fig. 9 shows the prose pseudocode semantics of the select instruction, generated from the specification in Fig. 2, which is very close to the original handwritten prose in Fig. 1a. AL algorithms are rendered into a prose specification document through three steps. First, the semantics described in AL is printed into English prose in reST markup. For example, the second step of select is printed as follows:

```
2. Pop :math:`\xref{syntax/types}{syntax-numtype}{\mathsf{i32}}.\\
   \xref{syntax/instructions}{syntax-instr-numeric}{\mathsf{const}}~c` from the stack.
```

Next, as in the LaTeX backend (§2.4), prose in reST is spliced into a skeleton specification document. Finally, the spliced document is processed by Sphinx [57], producing formats like PDF and HTML.

Note that prose in reST is not simple plaintext, but has inline math blocks as denoted by the `:math:` markup. Expressions in math blocks are typeset with LaTeX. Furthermore, the prose backend embeds cross-references into the math blocks with `\xref{doc}{section}{text}`. This serves as a reference to a `section` in some reST file `doc`, to be rendered as `text`. As in the original specification document, const in the second step references its syntax production rule in a separate file. This systematic insertion of references rules out possibilities of missing, broken, or misplaced links when inserted manually.

3.5 Interpreter Backend

SpecTec enables the interpretation of Wasm programs through a *meta-level interpretation* approach, similar to the ESMeta framework [1, 56]. By interpreting an AL program that denotes the Wasm semantics, with a Wasm program as its input value, we can indirectly interpret the Wasm program.

Initially, we parse a Wasm program into an AST, which an AL algorithm takes as input. Unlike ESMeta which also generates the parser automatically, SpecTec does not generate a parser for Wasm's text format, but sources out this task to the parser of the existing reference interpreter [85]. Then, the executable semantics represented in AL is automatically extracted as described in §3.3. We have developed an AL interpreter in OCaml based on the AL meta-level semantics, enabling the execution of an AL program. By executing the extracted Wasm semantics using the parsed AST of a Wasm module as input, we can indirectly execute Wasm programs.

As described in §3.2, we can execute an AL program by calling one of its algorithms. According to the Wasm specification, code is executed either when instantiating a module [20, §4.5.4] or when invoking a function exported by a module instance [20, §4.5.5]. Thus, the AL interpreter calls the *instantiate* algorithm to instantiate a module or the *invoke* algorithm to call a function. The algorithms either return a list of values or produce a *trap*, which is the result of execution.

4 EVALUATION

We developed SpecTec as an open-source project [2], and evaluated it based on the following:

- **RQ1. Correctness**: Does SpecTec correctly generate formal and prose specifications, as well as the interpreter backend? (§4.1)
- **RQ2. Bug Prevention**: Can SpecTec prevent bugs during Wasm standard development? (§4.2)
- **RQ3. Forward Compatibility**: Can SpecTec support future language features? (§4.3)

We evaluate SpecTec with the latest Wasm specification, Wasm 2.0 [20]; we manually specified its syntax, validation, and execution in the DSL, which we name Wasm2_{ST}. We have not yet included auxiliary functions for numeric instructions since they are numerous but do not pose new challenges for SpecTec. Instead, we use the reference interpreter's implementations for numeric functions. Also, for some parts of the Wasm 2.0 standard that we expect to be difficult to reason about in theorem provers, we use a different, but equivalent formulation that is closer to the one used by the official reference interpreter. For example, while Wasm 2.0 uses evaluation contexts to escape multiple block contexts in a single step, Wasm2_{ST} uses a bubbling-up technique to exit one block at a time. We also rewrite the module instantiation semantics to remove cyclic bindings within premises. Wasm2_{ST} amounts to 2,957 Lines of Code (LoC), while the corresponding official specification document written in reST is 5,526 LoC.

4.1 Correctness

We evaluate the correctness of the artifacts generated by SpecTec using Wasm2_{ST} as an input.

Formal and Prose Specification. SpecTec can generate two of the four key artifacts required by the W3C Wasm CG in order to standardize a feature. It can generate a formal specification in declarative style, written in LaTeX, for the abstract syntax, validation semantics, execution semantics, and binary format. It can also generate a prose pseudocode presenting validation in declarative style and execution in algorithmic style, written in reST. A generated PDF or HTML document with formal and prose notations [8] is very close to the respective parts of the hand-written specification [20].

Interpreter Backend. We evaluate the correctness by executing the official Wasm test suite [84]. This evaluation demonstrates two things: 1) that the meta-level interpreter can interpret Wasm programs; and 2) that Wasm2_{ST}, the process of translating it to AL, and the resultant Wasm semantics in AL, as represented by the prose, are correct with high confidence. A Wasm test consists of one or more Wasm modules and assertions to verify whether the implementation behaves as expected. Of the seven kinds of assertions [86], we exclude three related to parsing and validation (for which SpecTec does not have an interpreter backend) and one related to testing infinite loops. Thus, we use the following three kinds of assertions:

```
(assert_return <invoke> <result>*) ;; assert invocation has expected results
(assert_trap <invoke> <failure>)   ;; assert invocation traps with given failure string
(assert_trap <module> <failure>)   ;; assert module traps on instantiation
```

We performed our experiments with a MacBook Pro (16-inch, 2019) with a 2.4 GHz 8 core Intel Core i9 and 32GB of RAM. On this machine, the meta-level interpreter executed all 49,833 tests (47,391 `assert_return`, 2,408 `assert_trap` for actions, and 34 `assert_trap` for modules) in 58 seconds. While the meta-level interpreter currently exhibits slower performance compared to the reference interpreter, its design is focused primarily on ensuring fidelity of the algorithmic semantics generated from the DSL. Every applicable test in the official test suite has successfully passed, giving high confidence in the correctness of both the SpecTec implementation and the Wasm semantics written in it, and by extension, the correctness of the generated prose specification.

4.2 Bug Prevention

We evaluate SpecTec's ability to detect or prevent bugs during the authoring process by evaluating whether it can prevent the actual bugs that occurred during the Wasm standard's development [87]. We collected the bugs by investigating specification fixes within the Wasm standard's main GitHub branch over the last two years. A few bugs were not in scope for our framework, such as edits to non-normative explainers. Other than that, our analysis reveals that all remaining bugs would have been prevented using our tool. We classified these bugs into four categories based on the phase at which they would have been prevented.

Type Errors. We found three bugs that SpecTec's type checking would have prevented: 1) a missing field in the execution semantics of the elem.drop instruction [11], 2) a missing argument in the semantics of module instantiation [32] and 3) a wrong use of tabletype in the validation rule of table.set instruction [33]. We injected each bug into Wasm2_{ST}, used it as an input to SpecTec, and confirmed that SpecTec detected both bugs as type errors with informative error messages that included the error locations in the specification as well as the reasons for the errors.

Prose Errors. We found seven bugs that SpecTec's automatic prose generation would have prevented: 1) free identifiers in the execution semantics of control instructions [18], 2, 3) a free identifier in the execution semantics of the allocelem function [88] and memory.init instruction [4], 4) a missing parameter in the execution semantics of the table.set instruction [37], 5, 6) missing steps in the execution semantics of function invocation [58] and the module instantiation [5], and 7) an obsolete step in the execution semantics of function invocation [7]. Such prose errors do not exist in prose specifications generated by SpecTec, and we confirmed that the corresponding semantics were correctly specified in the generated PDF document [8].

Semantics Errors. We found three bugs that SpecTec's meta-level interpretation may have prevented: 1) a missing value in the value stack of the reduction rule of the table.grow instruction [77], 2) a wrong memory index for the memory.fill and memory.init instructions [31], and 3) popping a wrong number of values when exiting from a label [6]. We injected each bug into Wasm2_{ST}, used it as an input to SpecTec, executed the Wasm test suite with the generated meta-level interpreter, and confirmed that the meta-level interpreter detected all three bugs as semantics errors.

Editorial Fixes. Finally, we found numerous editorial and presentational issues, such as typographical errors in LaTeX, inconsistencies in writing style across the specification, or incorrect cross-references or hyperlinks to definitions. These errors do not arise in formal and prose specifications generated by SpecTec since they follow a predefined structure and style, removing the possibility of human errors or inconsistencies.

These results demonstrate that SpecTec is effective in preventing a wide range of human errors throughout the Wasm standardization process. It can detect or prevent various errors, including type errors, prose errors, semantics errors, and editorial fixes. We believe that SpecTec can significantly enhance the robustness and reliability of the Wasm standardization process while also reducing the burden on specification authors.

4.3 Forward Compatibility

We evaluate SpecTec's forward compatibility by applying it to five proposals ready for inclusion in the next version of Wasm ("Wasm 3.0"): typed function references [63], garbage collection [64], tail calls [66], multiple memories [65], and extended constant expressions [16]. The proposals add or modify 44 Wasm instructions and dozens of auxiliary helper functions; in particular, the garbage collection proposal adds substantial complications to the Wasm type system, introducing structural types, subtyping, and type recursion.

For each proposal, we extended Wasm2_{ST} with the proposal specified in SpecTec. Remarkably, describing all five proposals required only trivial adjustments to the DSL, mostly adding new custom operators for new user-definable notation. Qualitatively, we found that SpecTec's type-checking of DSL definitions was a significant boon during this process. In Wasm's current LaTeX specification which is manually written, updating the structure of an abstract syntax definition does not ensure that all of the definition's uses are likewise updated. SpecTec, in contrast, ensures that all uses respect the definition's new structure, avoiding a very common class of drafting error.

After extending Wasm2_{ST}, we utilized SpecTec to generate formal and prose specifications automatically. It readily generated formal and prose specifications for 42 of the 44 Wasm instructions and for all helper functions. The remaining two instructions' reduction rules used new patterns, such as complex use of inverse functions, which the DL-to-AL translator was not able to handle. After we revised the rules to reflect the translator's intended style, it correctly generated their prose specifications. Extending the translator's capability to handle reduction rules with more diverse styles is future work. Finally, we executed the proposals' test suites using the generated meta-level interpreter. For the subtyping validation of Wasm types introduced by the garbage collection proposal, SpecTec uses the reference interpreter's implementation. During this evaluation, we found ten bugs in the proposals: two type errors, two prose errors, four semantics errors, and two editorial fixes. We reported them to the specification authors and received conformation [**? ? ? ? ? ? ?**]. After fixing the errors in the proposals, we applied SpecTec to the revised Wasm2_{ST}. For each proposal, the meta-level interpreter passed all the 1,331 tests.

The results demonstrate that, with few adjustments, SpecTec can handle future language features and detect and prevent numerous human mistakes throughout the standardization process. We believe that SpecTec is a long-term solution for supporting a growing language like Wasm.

5 RELATED WORK

Programming Language Frameworks. Researchers have presented numerous frameworks to mechanize the definitions of programming languages.

Sail [24] is a DSL and toolchain for defining processor instruction set architectures, which can output SMT encodings, theorem prover definitions, and LaTeX documentation fragments. While SpecTec relies extensively on relations, following a declarative approach to language semantics as used in the existing Wasm standard, Sail is first-order and imperative, and hence its definitions can be directly executed. The Sail type system can express complex dimension constraints which must be checked by an external SMT solver. SpecTec uses a simpler form of dimension constraints which are checked fully within the tool (§2.2).

Lem [49] is a toolchain and DSL for defining semantic models, which can output LaTeX, executable code, and theorem prover definitions. Lem does not include strong support for custom syntax, and so it is challenging for Lem to output LaTeX which fits the format of an existing specification such as Wasm. While Lem does support both relational and functional styles of definition, unlike SpecTec it does not support inference of a functional definition from a relational one – as in our algorithmic backends (§3). SpecTec benefits from its specialization to Wasm as its inference algorithm for this purpose can be tightly scoped, and can fail in cases that are irrelevant to the Wasm semantics.

Ott [68] allows language designers to specify the semantics in inference rules, and generates code for Coq, HOL, and Isabelle/HOL. It has been used for case studies like a large fragment of OCaml. Spoofax [74] supports agile development of textual domain-specific languages with the Eclipse IDE support. Skeleton [67] specifies language semantics in big-step semantics and constructs both concrete and abstract interpretation. PLTRedex [22] describes language semantics in reduction rules, and it has specified the semantics of Scheme [61]. The K framework [79] can generate various tools, including interpreters, model checkers, and verifiers, so long as the definitions under consideration

can be encoded in K's term-rewriting system. K has been used to specify semantic fragments of languages such as C [21], Java [14], and JavaScript [52]. While the aforementioned frameworks aim to support general-purpose languages with declarative semantics, ESMeta [1] is designed to support JavaScript with imperative semantics. By devising an algorithmic language, $\mathsf{IR}_{\mathsf{ES}}$, to specify the semantics in the prose ECMAScript standard, ESMeta can generate diverse tools [56, 55, 54, 53], including a test harness which has been officially integrated into the continuous integration (CI) systems of ECMAScript [73] and the Test262 conformance test suite [72] since November 2022.

Wasm Semantics in Existing Language Frameworks. Fragments of the Wasm 1.0 semantics have been mechanized in PLTRedex and K. Two PLTRedex models specify a large core of its reduction rules [26]: wasm-redex [71] and Wasm-Redex [23], which are executed by the rewrite system of PLTRedex. Both models do not specify the Wasm module semantics, thus complete execution of Wasm programs is not available. KWasm [29] specifies the Wasm runtime semantics, including the module semantics, in a single rewrite system in the K framework. Since it does not model the entire Wasm semantics yet, it fails for a number of tests in the official Wasm test suite. While both PLTRedex and K execute the Wasm semantics using their rewriting engines, SpecTec provides indirect interpretation over an algorithmic representation of the semantics.

Theorem Prover Language Mechanizations. Two complete mechanizations of the Wasm 1.0 semantics exist in the Coq and Isabelle theorem provers [82]. Unlike the works described above, which use frameworks specifically designed for language semantics, these two mechanizations are written manually as collections of inductive definitions and functions within the formal languages of their theorem provers. The models are accompanied by proofs of the soundness of the Wasm type system. However, these models have not yet been updated to support all the upcoming features of Wasm 2.0 due to the onerous process of manually extending all of the models' existing definitions. Moreover, in comparison to a purpose-built framework for language semantics, the process of constructing a proof involves many hand-coded inference steps, and therefore, even minor changes to the structure of the underlying model can invalidate large swathes of existing results, such as each model's type soundness proof. A number of mechanizations of other language semantics exist in general-purpose interactive theorem provers – such as Java [38], JavaScript [13, 25], Standard ML [43, 41], C [44, 40, 50], and Rust [34]. Except for the mechanizations of Standard ML, these almost invariably stick to fixed fragments of the languages in order to make the mechanization process tractable. The process of generating executable code from a mechanized inductive definition is sometimes referred to as animation [12, 45]. In contrast with our approach, prior work is generally not concerned with minimizing the number of new variables introduced by the animation process.

6 FUTURE WORK

Declarative specification to algorithmic pseudocode. The current paper focuses on the use of SpecTec to generate algorithmic pseudocode from a declarative specification of Wasm's dynamic semantics (i.e., the instantiation and execution phases of Wasm). The declarative specification provides a single source of truth and is rendered directly into the LaTeX formal specification. The algorithmic pseudocode generated from the declarative specification is rendered into the reST markup prose pseudocode. The generated algorithmic representation is executable, so it can be directly tested against the existing Wasm test suite and reference interpreter.

While SpecTec supports the definition of Wasm's type system and basic formal grammar and their rendering into specification text, it does not yet support the generation of executable code (i.e., a type checker and parser) from these definitions. SpecTec also does not yet include the formal definition of Wasm's arithmetic operations. Instead, we modularize SpecTec's generated artifacts so that handwritten definitions for these parts of the language can be separately provided. We intend to investigate further support for defining and executing these parts in future work.

The AL interpreter also allows us to easily analyze the coverage of the existing Wasm test suite. In the future, we intend to investigate the automatic generation of test cases and fuzzing harnesses, guided by the semantic definitions in SpecTec — we expect that approach to have the potential for producing highly comprehensive tests tailored to maximize coverage of semantic edge cases.

Declarative specification to theorem prover formal specification. A major future direction not covered by the current paper is the implementation of SpecTec backends for a range of different proof assistants. This will enable the generation of formal specifications for proof assistants, including Agda, Coq, Isabelle, and Lean, from the single source of truth provided by the SpecTec declarative specification. Wasm has previously benefited from the mechanization of its initial draft specification — in attempting to give a mechanized proof of the soundness of the Wasm type system, several errors in the draft specification were discovered and corrected before the proof was ultimately completed [81]. However, this mechanization was transcribed entirely by hand from the pen-and-paper specification and has not been kept up to date with upcoming new Wasm features.

Compounding the issue of mechanization effort, it is beneficial to support several mechanizations of Wasm since each theorem prover has its own strengths and weaknesses. For the two mechanizations of Wasm 1.0 [82], the Coq mechanization uses the Coq-only Iris framework [35] in defining a mechanized program logic [59], while the Isabelle mechanization partially uses the Isabelle-only Sepref [42] in defining a verified monadic interpreter for Wasm [83]. Once SpecTec supports the automatic generation of formal specifications for theorem provers, it will improve confidence that mechanized proofs correspond to the Wasm specification, speeding up the mechanization process as the specification evolves, further improving confidence in Wasm's design and specification.

We have begun work on backends for Agda, Coq, and Lean. In doing so, we implemented a number of IL to IL passes to simplify the output of the elaboration phase and bring it closer to a format suitable for theorem provers (§2.3). An internal IL type-checker re-validates that the IL is still internally consistent after each pass, in order to capture possible bugs in the transformation. We envisage substantial parts of this infrastructure being shared amongst theorem prover backends.

A particular challenge for theorem prover backends is posed by the differing needs of SpecTec and theorem provers. Whereas SpecTec can be tailored to faithfully replicate the precise metatheory of the existing Wasm formalism, theorem provers come with their own existing metatheories, and so we must carefully manage any mismatches to achieve faithful translations. Agda, in particular, heavily encourages the use of an *intrinsically*-typed representation of a language's semantics, where the language's type system is directly encoded into the language's abstract syntax using Agda-level dependent types [10]. The existing Wasm specification and its mechanizations in Coq and Isabelle use an *extrinsic* representation of well-typed terms where the typing judgment is a separate inductive definition. We plan to investigate techniques for generating intrinsically-typed representations of Wasm using SpecTec.

Meta-meta-theory for SpecTec. While the SpecTec DSL is used to describe formal language semantics, we did not give a meta-level semantics for this DSL itself. Other frameworks such as Ott/Lem have taken similar approaches in their academic presentations.

We expect that the core work of doing meta-theory for Wasm will continue to happen in actual theorem provers that are intended for this task and have their own meta-semantics; we have no intention of turning SpecTec into one itself. Rather, we regard it as a versatile frontend supporting such work. Still, formalizing the semantics of the DSL (possibly as an undecidable logic) would be interesting future work; it was not a priority for us, as it is not on the critical path to getting SpecTec adopted by Wasm's standards body, unlike other research questions that we had to address in supporting the variety of backends required. One interesting exercise would be to try specifying the semantics of SpecTec in itself.

Pragmatics of varying backend assumptions. The SpecTec DSL and its frontend are fairly generic. While they build in most of the meta-level *notation* used in the Wasm specification, they know nothing about the actual Wasm syntax and *semantics* — this is entirely specified within the DSL. Consequently, the overall design of SpecTec is relatively future-proof, and our experience with translating a variety of non-trivial Wasm proposals validated that. We expect extensions to the DSL itself to be necessary only on the rare occasion that the specification of a new Wasm feature requires entirely new meta-level concepts.

However, this does not hold for all of the tool's *backends*, some of which make much more specific assumptions about the input, to varying degrees. For example, the prose backend needs to recognize typing relations and reduction relations in order to generate appropriate English (much of this is already embodied in the transformation to the AL). The interpreter backend needs to specialize even further, since it (currently) connects to an externally provided parser and a numerics library from the pre-existing reference interpreter in OCaml.

We hope to be able to eliminate some of this special-casing over time, e.g., by encoding numerics in SpecTec itself. However, the fact will remain that different backends hard code knowledge to a differing degree, and pattern match certain aspects of the input specification. The consequence is that individual backends will reject specifications that they cannot (yet) fully handle.

To function as part of the official authoring workflow, we will need to define an appropriate process for supporting specification authors that hit one of these limitations and need an extension to some of the tool's backends. In general, we do not expect authors to touch the implementation of the tool, though some expertise will have to grow within the Wasm standards community to maintain it over time.

7 CONCLUSION

We have presented Wasm SpecTec, a domain-specific language and toolchain intended to replace the current authoring and specification workflow of the Wasm language standard. As Wasm evolves, we expect to iterate further on some design details of the DSL, for example, revolving around concrete syntax and striking the right balance between WYSIWYG and precision, since classical notation can sometimes be difficult to disambiguate. Likewise, there are various levels of polishing possible for the generated prose, and we expect it to improve further over time. More experience with prover backends in the future may also impact the design.

We ultimately aim for the Wasm standards community to specify all current and future Wasm features using SpecTec and replace most of the manually authored artifacts necessary for Wasm's standardization process with our generated ones, enhancing the efficiency and reliability of the process. Initial feedback from the community has been positive. Our evaluations demonstrate that SpecTec would effectively avoid many historical mistakes in the published Wasm standard and many more in currently in-progress feature specifications.

SpecTec has the potential to empower those working on the standardization of Wasm to engage more directly with the writing and maintenance of the specification. While SpecTec still requires some understanding of the underlying mathematical formalism, rather than unstructured LaTeX, the interface is a checked domain-specific language along with tools that automatically generate the LaTeX and prose. Moreover, SpecTec supports rapid prototyping of extensions, including execution of programs that use those extensions. We believe SpecTec is ready for evaluation by Wasm's standards body and have begun the process of working towards its official adoption.

Finally, we see Wasm and SpecTec as evidence that a rigorous and formal approach to language specification is possible and can be scaled to industrial-strength languages. Although the SpecTec tool itself is specialized to Wasm's needs, we believe that its overall architecture can be used as a blueprint to replicate a similar approach for other language specifications.

ACKNOWLEDGMENTS

This research was supported by National Research Foundation of Korea (NRF) (2022R1A2C200366011 and 2021R1A5A1021944), Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (2022-0-00460 and 2023-2020-0-01819), Samsung Electronics Co., Ltd (G01210570), and UKRI Future Leaders Fellowship MR/T043830/1 (EHOP). We also thank Suhyeon Ryu, Hoseong Lee, and Hyunhee Kang for their assistance with implementing the framework. And we thank Schloss Dagstuhl for hosting seminar 23101 on the Foundations of WebAssembly, which sparked this project.

DATA-AVAILABILITY STATEMENT

The artifact containing the source code of SpecTec and the dataset of our study are publicly available at [**?**]. The latest version of SpecTec is maintained as an open-source project as a GitHub repository at https://github.com/Wasm-DSL/spectec.

REFERENCES

[1] 2022. *ESMeta: An ECMAScript specification metalanguage used for automatically generating language-based tools.* https://github.com/es-meta/esmeta
[2] 2023. Anonymized for double-blind reviewing.
[3] Heejin Ahn and WebAssembly Community Group. 2023. *Exception Handling Proposal for WebAssembly.* https://github.com/WebAssembly/exception-handling/
[4] Andreas Rossberg. 2021. *Fix variable name typos.* https://github.com/WebAssembly/spec/commit/4353b29
[5] Andreas Rossberg. 2022. *Add missing case for declarative elem segments.* https://github.com/WebAssembly/spec/commit/ff149b4
[6] Andreas Rossberg. 2023. *Fix reduction rule for label.* https://github.com/WebAssembly/spec/commit/8f5c489
[7] Andreas Rossberg. 2023. *Remove an obsolete exec step.* https://github.com/WebAssembly/spec/commit/f54b5b8
[8] Anonymized for double-blind reviewing. 2023. Automatically Generated WebAssembly Specification.
[9] Anonymized for double-blind reviewing. 2023. Bugs in WebAssembly Proposals Detected by SpecTec.
[10] Casper Bach Poulsen, Arjen Rouvoet, Andrew Tolmach, Robbert Krebbers, and Eelco Visser. 2017. Intrinsically-Typed Definitional Interpreters for Imperative Languages. *Proc. ACM Program. Lang.* 2, POPL, Article 16 (dec 2017), 34 pages. https://doi.org/10.1145/3158104
[11] Ben Visness. 2023. *Add missing type to elem.drop and store soundness.* https://github.com/WebAssembly/spec/commit/5b18d52
[12] Stefan Berghofer, Lukas Bulwahn, and Florian Haftmann. 2009. Turning Inductive into Equational Specifications. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics* (Munich, Germany) *(TPHOLs '09)*. Springer-Verlag, Berlin, Heidelberg, 131–146. https://doi.org/10.1007/978-3-642-03359-9_11
[13] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. 2014. A Trusted Mechanised JavaScript Specification. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 87–100. https://doi.org/10.1145/2535838.2535876
[14] Denis Bogdănaş and Grigore Roşu. 2015. K-Java: A Complete Semantics of Java. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*.
[15] David M. Cerna and Temur Kutsia. 2023. Anti-unification and Generalization: A Survey. In *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI-23*, Edith Elkind (Ed.). International Joint Conferences on Artificial Intelligence Organization, 6563–6573. https://doi.org/10.24963/ijcai.2023/736 Survey Track.
[16] Sam Clegg and WebAssembly Community Group. 2023. *Extended Constant Expressions Proposal for WebAssembly.* https://github.com/WebAssembly/extended-const/
[17] De Moura, Leonardo and Bjørner, Nikolaj. 2008. Z3: An Efficient SMT Solver *(TACAS'08/ETAPS'08)*. Springer-Verlag, 337–340.
[18] Dongjun Youn. 2023. *Add missing access to current frame in prose.* https://github.com/WebAssembly/spec/commit/be820b2
[19] ECMA International. 2023. *ECMA-262, 14th edition, ECMAScript ®2023 Language Specification.* https://262.ecma-international.org
[20] Andreas Rossberg (editor). 2022. *WebAssembly Specification (Release 2.0).* https://webassembly.github.io/spec/core/

[21] Chucky Ellison and Grigore Roşu. 2012. An Executable Formal Semantics of C with Applications. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*.
[22] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. 2009. *Semantics Engineering with PLT Redex*. The MIT Press.
[23] Adam T. Geller. 2021. *WASM-Redex*. https://github.com/atgeller/WASM-Redex
[24] Kathryn E. Gray, Gabriel Kerneis, Dominic Mulligan, Christopher Pulte, Susmit Sarkar, and Peter Sewell. 2015. An integrated concurrency and core-ISA architectural envelope definition, and test oracle, for IBM POWER multiprocessors. In *Proceedings of the 48th International Symposium on Microarchitecture* (Waikiki, Hawaii) *(MICRO-48)*. Association for Computing Machinery, New York, NY, USA, 635–646. https://doi.org/10.1145/2830772.2830775
[25] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. 2010. The Essence of Javascript. In *Proceedings of the 24th European Conference on Object-Oriented Programming* (Maribor, Slovenia) *(ECOOP'10)*. Springer-Verlag, Berlin, Heidelberg, 126–150.
[26] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the Web up to Speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Association for Computing Machinery, New York, NY, USA, 185–200. https://doi.org/10.1145/3062341.3062363
[27] Adam Hall and Umakishore Ramachandran. 2019. An Execution Model for Serverless Functions at the Edge. In *Proceedings of the International Conference on Internet of Things Design and Implementation*. Association for Computing Machinery, New York, NY, USA, 225–236. https://doi.org/10.1145/3302505.3310084
[28] Pat Hickey. 2019. *Lucet Takes WebAssembly Beyond the Browser*. https://www.fastly.com/blog/announcing-lucet-fastly-native-webassembly-compiler-runtime
[29] Runtime Verification Inc. 2023. *KWasm*. https://github.com/runtimeverification/wasm-semantics
[30] ISO/IEC. 2018. *ISO/IEC 9899:2018*. https://www.iso.org/standard/74528.html
[31] Jim Blandy. 2023. *Remove stray x indices*. https://github.com/WebAssembly/spec/commit/e7f6e1c
[32] Julien Cretin. 2022. *Fix typo in element execution*. https://github.com/WebAssembly/spec/commit/793b3ff
[33] Julien Cretin. 2022. *Fix typos in instruction validation rules*. https://github.com/WebAssembly/spec/commit/79ef7af
[34] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. *Proc. ACM Program. Lang.* 2, POPL, Article 66 (dec 2017), 34 pages. https://doi.org/10.1145/3158154
[35] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151
[36] Richard M. Karp. 1972. *Reducibility among Combinatorial Problems*. 85–103. https://doi.org/10.1007/978-1-4684-2001-2_9
[37] Keith Winstein. 2022. *Fix missing immediate on table.set*. https://github.com/WebAssembly/spec/commit/f6ae547
[38] Gerwin Klein and Tobias Nipkow. 2006. A Machine-Checked Model for a Java-like Language, Virtual Machine, and Compiler. *ACM Trans. Program. Lang. Syst.* 28, 4 (jul 2006), 619–695. https://doi.org/10.1145/1146809.1146811
[39] Donald E. Knuth. 2000. Dancing links. arXiv:cs/0011047
[40] Robbert Krebbers. 2015. *The C standard formalized in Coq*. Ph. D. Dissertation. Radboud University Nijmegen.
[41] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 179–191. https://doi.org/10.1145/2535838.2535841
[42] Peter Lammich. 2019. Refinement to Imperative HOL. 62, 4 (apr 2019), 481–503. https://doi.org/10.1007/s10817-017-9437-1
[43] Daniel K. Lee, Karl Crary, and Robert Harper. 2007. Towards a Mechanized Metatheory of Standard ML. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Nice, France) *(POPL '07)*. Association for Computing Machinery, New York, NY, USA, 173–184. https://doi.org/10.1145/1190216.1190245
[44] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (jul 2009), 9 pages. https://doi.org/10.1145/1538788.1538814
[45] Andreas Lochbihler and Lukas Bulwahn. 2011. Animating the Formalised Semantics of a Java-Like Language. In *Interactive Theorem Proving*, Marko van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 216–232.
[46] Maxime Chabert and Christine Solnon. 2020. A Global Constraint for the Exact Cover Problem: Application to Conceptual Clustering. *Journal of Artificial Intelligence Research* 67 (2020), 509 – 547.
[47] Microsoft. 2013. *C# Language Specification 5.0*. https://www.microsoft.com/en-us/download/details.aspx?id=7029
[48] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. 1997. *The Definition of Standard ML (Revised)*.

[49] Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. 2014. Lem: reusable engineering of real-world semantics. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming* (Gothenburg, Sweden) *(ICFP '14)*. Association for Computing Machinery, New York, NY, USA, 175–188. https://doi.org/10.1145/2628136.2628143

[50] Michael Norrish. 1998. *C formalised in HOL*. Technical Report.

[51] Oracle. 2023. *Java Language and Virtual Machine Specifications*. https://docs.oracle.com/javase/specs/

[52] Daejun Park, Andrei Ştefănescu, and Grigore Roşu. 2015. KJS: A Complete Formal Semantics of JavaScript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*.

[53] Jihyeok Park, Seungmin An, and Sukyoung Ryu. 2022. Automatically Deriving JavaScript Static Analyzers from Specifications Using Meta-Level Static Analysis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Association for Computing Machinery, New York, NY, USA, 1022—-1034.

[54] Jihyeok Park, Seungmin An, Wonho Shin, Yusung Sim, and Sukyoung Ryu. 2021. JSTAR: JavaScript Specification Type Analyzer using Refinement. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Association for Computing Machinery, New York, NY, USA, 606–616.

[55] Jihyeok Park, Seungmin An, Dongjun Youn, Gyeongwon Kim, and Sukyoung Ryu. 2021. JEST: N+1-version Differential Testing of Both JavaScript Engines and Specification. In *Proceedings of IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, Association for Computing Machinery, New York, NY, USA, 13–24.

[56] Jihyeok Park, Jihee Park, Seungmin An, and Sukyoung Ryu. 2020. JISET: JavaScript IR-based Semantics Extraction Toolchain. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 647–658. https://doi.org/10.1145/3324884.3416632

[57] Sphinx Project. 2008. *Sphinx*. https://www.sphinx-doc.org/

[58] R1ru. 2023. *Pop dummy frame after Invocation*. https://github.com/WebAssembly/spec/commit/be1f563

[59] Xiaojia Rao, Aïna Linn Georges, Maxime Legoupil, Conrad Watt, Jean Pichon-Pharabod, Philippa Gardner, and Lars Birkedal. 2023. Iris-Wasm: Robust and Modular Verification of WebAssembly Programs. *Proc. ACM Program. Lang.* 7, PLDI, Article 151 (jun 2023), 25 pages. https://doi.org/10.1145/3591265

[60] Alastair Reid. 2016. Trustworthy Specifications of ARM® V8-A and v8-M System Level Architecture. In *Proceedings of the 16th Conference on Formal Methods in Computer-Aided Design* (Mountain View, California) *(FMCAD '16)*. FMCAD Inc, Austin, Texas, 161–168.

[61] Robert Bruce Findler and Jacob Matthews. 2023. *R6RS Semantics Model and Reference Implementations*. https://www.r6rs.org/refimpl/

[62] Andreas Rossberg. 2023. *As low-level as possible, but no lower*. https://icfp23.sigplan.org/details/icfp-2023-icfp-keynotes/38/As-low-level-as-possible-but-no-lower

[63] Andreas Rossberg and WebAssembly Community Group. 2023. *Function Reference Types Proposal for WebAssembly*. https://github.com/WebAssembly/function-references/

[64] Andreas Rossberg and WebAssembly Community Group. 2023. *GC Proposal for WebAssembly*. https://github.com/WebAssembly/gc/

[65] Andreas Rossberg and WebAssembly Community Group. 2023. *Multiple Memories Proposal for WebAssembly*. https://github.com/WebAssembly/multi-memory/

[66] Andreas Rossberg and WebAssembly Community Group. 2023. *Tail Call Proposal for WebAssembly*. https://github.com/WebAssembly/tail-call/

[67] Alan Schmitt. 2019. *Skeletal Semantics*. https://skeletons.inria.fr/

[68] Peter Sewell and Francesco Zappa Nardelli. 2007. Ott release, version 0.10.9. http://www.cl.cam.ac.uk/~pes20/ott/

[69] Ben Smith, Conrad Watt, and WebAssembly Community Group. 2023. *Threads Proposal for WebAssembly*. https://github.com/WebAssembly/threads/

[70] Guy Steele. 2017. It's Time for a New Old Language. In *Principal and Practices of Parallel Programming (ACM SIGPLAN Notices, Vol. 52)*. 1. Issue 8.

[71] Asumu Takikawa. 2019. *wasm-redex*. https://github.com/takikawa/wasm-redex

[72] TC39. 2022. *CI: Integrate ESMeta*. https://github.com/tc39/test262/pull/3730

[73] TC39. 2022. *Meta: integrate esmeta type checker into CI*. https://github.com/tc39/ecma262/pull/2926

[74] Spoofax Team. 2010. *Spoofax: The Language Designer's Workbench*. https://spoofax.dev

[75] W3C Team. 2015. *WebAssembly Community Group*. https://www.w3.org/community/webassembly/

[76] Ata Tekeli. 2022. *WebAssembly (WASM) in Blockchain*. https://blog.devgenius.io/webassembly-wasm-in-blockchain-f651a8ac767b

[77] Tom Stuart. 2023. *Add missing value to table.grow reduction rule*. https://github.com/WebAssembly/spec/commit/3545ad0

[78] Kenton Varda. 2017. *Introducing Cloudflare Workers: Run JavaScript Service Workers at the Edge.* https://blog.cloudflare.com/introducing-cloudflare-workers/
[79] Runtime Verification. 2013. *K Semantic Framework.* https://kframework.org/
[80] Stefan Wallentowitz, Bastian Kersting, and Dan Mihai Dumitriu. 2022. Potential of WebAssembly for Embedded Systems. In *2022 11th Mediterranean Conference on Embedded Computing (MECO).* 1–4. https://doi.org/10.1109/MECO55406.2022.9797106
[81] Conrad Watt. 2018. Mechanising and verifying the WebAssembly specification. *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs* (2018). https://dl.acm.org/doi/10.1145/3167082
[82] Conrad Watt, Xiaojia Rao, Jean Pichon-Pharabod, Martin Bodin, and Philippa Gardner. 2021. Two Mechanisations of WebAssembly 1.0. In *Proceedings of the 24th international symposium of Formal Methods (FM21), Beijing, China; November 20-25, 2021 (Lecture Notes in Computer Science, Vol. 13047)*, Marieke Huisman, Corina S. Pasareanu, and Naijun Zhan (Eds.). Springer, 61–79. https://doi.org/10.1007/978-3-030-90870-6_4
[83] Conrad Watt, Maja Trela, Peter Lammich, and Florian Märkl. 2023. WasmRef-Isabelle: A Verified Monadic Interpreter and Industrial Fuzzing Oracle for WebAssembly. *Proc. ACM Program. Lang.* 7, PLDI, Article 110 (jun 2023), 24 pages. https://doi.org/10.1145/3591224
[84] WebAssembly Community Group. 2023. *WebAssembly Core Testsuite.* https://github.com/WebAssembly/spec/tree/main/test/core
[85] WebAssembly Community Group. 2023. *WebAssembly Parser.* https://github.com/WebAssembly/spec/tree/main/interpreter/text
[86] WebAssembly Community Group. 2023. *WebAssembly Reference Interpreter: Scripts.* https://github.com/WebAssembly/spec/blob/main/interpreter/README.md#scripts
[87] WebAssembly Community Group. 2023. *WebAssembly specification, reference interpreter, and test suite.* https://github.com/WebAssembly
[88] Whirlicote. 2022. *Fix naming typo.* https://github.com/WebAssembly/spec/commit/04beeb7

Received 2023-11-16; accepted 2024-03-31