MEng Individual Project

Imperial College London

Department of Computing

Lemmagen: Extending the WebAssembly Specification Toolchain for Theorem Provers

Author:
Taichi Maeda

Supervisor:
Prof. Philippa Gardner

Second Marker:
Prof. Sophia Drossopoulou

July 16, 2025

Abstract

WebAssembly (Wasm) is a virtual instruction set architecture (ISA) introduced in 2017 by Andreas Rossberg as a low-level bytecode language [HRS+17]. One of the defining features of Wasm is its formalism, which has been fundamental to its design process [Ros25].

This formalism facilitates the mechanisation of Wasm, as demonstrated by pioneering efforts such as WasmCert, which established the soundness properties of the language [WRPP+21]. However, as the Wasm specification continues to expand, maintaining these mechanisations manually has become increasingly challenging.

To address this problem, SpecTec was developed to streamline the Wasm specification process [YSL+24]. It introduces a domain-specific language (DSL) from which artefacts such as LaTeX documentation, prose specifications, fuzzer test suites, a reference interpreter in OCaml and even mechanised definitions in Coq (Rocq) can be auto-generated.

The mechanised proof of the preservation property in Wasm has been successfully ported from WasmCert to SpecTec, using definitions auto-translated from the SpecTec DSL [Cup24]. This builds upon prior work involving IL2Coq, a component of the SpecTec toolchain responsible for producing mechanised definitions in Coq (Rocq).

This project extends this work by developing a mechanised proof of the progress property, which, combined with the preservation property, establishes the soundness of Wasm. The development of the progress proof identified several key areas for improvement, which were subsequently addressed by introducing new features to the SpecTec DSL, including first-order logic and template constructs. These contributions highlight potential future directions for the evolution of the SpecTec toolchain and the design process of the WebAssembly language as a whole.

Acknowledgements

I would first like to thank my supervisor, Prof. Philippa Gardner, for her consistent support, which guided me in the right direction throughout this project. The discussions with you were both inspiring and insightful, and it was a pleasure to be part of such a high-impact project.

I would also like to thank the PhD student, Xiaojia Rao, for his valuable insights into the intricate details of the Coq theorem prover and the Wasm specification, which reflected his profound understanding of the subject.

I am also grateful to another PhD student, Diego Cupello, who developed IL2Coq and completed the preservation proof as a master's student last year. Thank you for your prompt and accurate feedback — your advice was invaluable in making the right design choices in this project.

Finally, I would like to thank my family for their continued support throughout my academic journey. Thank you for every opportunity and experience you have made possible — I would not be where I am today without you.

Contents

Chapter 1

Introduction

1.1 Motivation

WebAssembly (Wasm) is a virtual instruction set architecture (ISA) introduced in 2017 by Andreas Rossberg as a low-level bytecode language. Currently, it serves as a compilation target for languages like C, C++ and Rust, enabling these languages to run efficiently and securely across platforms [HRS+17][Wor22a].

Wasm is designed with the following goals [HRS+17]:

- **Safety**: Wasm ensures the safe execution of portable low-level code, providing stronger guarantees than traditional managed language runtimes. The safety on the Web is crucial as browsers often execute code from untrusted sources.
- **Performance**: Wasm serves as a compilation target for highly optimised low-level code, minimising the performance overhead typically associated with managed languages, especially those using garbage collection.
- **Portability**: Wasm is designed as a virtual ISA rather than a physical one. This design ensures portability, enabling Web applications to run consistently across different browsers and platforms.
- **Compactness**: Wasm's compact binary format reduces load times, saves bandwidth and enhances responsiveness when transmitted over the network.

Wasm has been rigorously standardised by the W3C [Wor19]. Unlike other bytecode languages like Java bytecode [Ora23], Wasm's specification includes formal semantics and proven theorems for its soundness, making it one of the first industrial-strength bytecode languages designed with formal semantics [HRS+17].

This formalism enables the mechanisation of Wasm. The initial mechanisation in Isabelle was developed by Conrad in 2018 [Wat18], focusing on the draft version of Wasm. Building on this work, Conrad and Rao completed the full mechanisation of Wasm 1.0 in Isabelle and Coq, known as WasmCert-Isabelle and WasmCert-Coq [WRPP+21]. These mechanisations provide rigorous proofs of the formal semantics of the language, ensuring alignment with its intended behaviour and soundness properties.

In contrast to the advantages of this formalism, manual maintenance of the Wasm specification has proven labour-intensive and prone to errors. Translating formal rules into both LaTeX and prose formats, while simultaneously producing other artefacts, complicates code reviews and increases the risk of inconsistencies. As Wasm rapidly evolves with new features, this manual process becomes unsustainable [YSL+24].

To address these challenges, the Wasm community introduced SpecTec, a domain-specific language (DSL) that serves as a "single source of truth" for the Wasm specification [YSL+24]. SpecTec automates the generation of key artefacts, including formal specifications in LaTeX and prose formats, pseudocode in reStructuredText and a reference interpreter in OCaml.

One of the long-term goals of SpecTec is to automate mechanisation and proof generation in theorem provers like Coq and Isabelle. An initial step towards this goal is IL2Coq, developed by Diego [Cup24], a SpecTec backend that translates the intermediate language (IL) of SpecTec into inductive definitions in Coq. IL2Coq has already demonstrated its capability by completing the mechanised proof of the preservation property using the auto-translated definitions [Cup24].

The next phase of this work focuses on completing the progress proof, thereby establishing the soundness of the WebAssembly language using the auto-translated definitions. The development of the progress proof will help identify limitations of the SpecTec toolchain, which will guide a series of improvements to be made in this project, collectively referred to as "Lemmagen".

1.2 Contributions

This project makes the following contributions:

- The mechanised proof of the progress property using the definitions auto-translated from the SpecTec DSL, which, when combined with Diego's previous work on the preservation proof, establishes the soundness of Wasm. The development of the proof identified key inconsistencies in the DSL that had previously gone undetected by both the SpecTec toolchain and the preservation proof, which were subsequently resolved. Additionally, an experimental attempt was made to mechanise the proof of a weak form of the soundness property.

- The auto-generation of decidable equality proofs, in order to automate the definition of SSReflect's `EqType` instances for data types produced by IL2Coq. The integration with IL2Coq and additional Coq techniques led to a significant simplification of these decidable equality proofs in comparison to WasmCert-Coq.

- The extension of the SpecTec DSL and its downstream translation with first-order logic constructs, enabling specification of theorem statements, auxiliary lemmas and predicates as part of the "single source of truth". The Coq, LaTeX, prose and splice backends were subsequently integrated to support automatic generation of theorem statements in their corresponding formats.

- The introduction of a template mechanism in the SpecTec DSL and its downstream translation, which enables finer specification of the proof structure by abstracting auxiliary lemmas through templates. As part of the template middlend, a simultaneous tree/trie-traversal algorithm was designed to handle the expansion of template definitions containing nested wildcards and Cartesian products of wildcards.

Chapter 2

Background

2.1 Language Standardisation

Standardisation of a programming language refers to the process of formally defining the syntax and semantics of the language. Notable examples of such standardisation efforts include the ANSI standards for C [ISO11], the JVM specification for Java [Ora23] and the formal semantics outlined in the Wasm specification [Wor22a].

This standardisation enables the validation of the syntax and semantics of a language across its various implementations, including compilers, interpreters and other tools that interact with the language. It ensures that the behaviour of the language remains consistent across platforms and that undefined or unspecified behaviour [cpp25] is eliminated.

2.2 Software Mechanisation

In the context of software validation, mechanisation refers to the process of formally representing and automating the verification of the semantics of a language, ensuring that its behaviour aligns with the formal specification. This typically involves translating the high-level behaviours of the language into a formal framework, often using a proof assistant such as Coq [Inr25a] and Isabelle [Isa].

Mechanisation is highly valuable because it significantly reduces the risk of errors in software validation. Manual verification of complex systems is not only time-consuming but also prone to mistakes and inconsistencies. Human-readable proofs can often overlook edge cases or misinterpret details, while automated mechanisms can exhaustively check all possible scenarios. This is particularly important in safety-critical systems, such as aircraft control software, where even small bugs can lead to catastrophic failures [Sou14]. With software mechanisation, we can ensure correctness with a level of precision that would be nearly impossible to achieve through manual testing, thereby offering strong guarantees about the safety and reliability of the system.

2.3 Proof Assistants

A proof assistant is a software tool that helps users construct mathematical proofs by providing a framework for formalising and certifying statements. These tools are typically based on functional programming languages and allow users to express their reasoning in a formal manner.

Proof assistants generally rely on formal systems such as dependent type theory [nLaa] or higher-order logic (HOL) [nLab]. Dependent type-based proof assistants are based on types that may be indexed by values, while HOL-based systems use a logical framework based on higher-order logic to construct proofs.

Several proof assistants are widely used, each with its own strengths. Some of the most well-known proof assistants include:

- **Coq**: A dependent type-based proof assistant that allows for the formalisation of mathematical theories and the development of certified software [Inr25a]. Coq is built on constructive logic but can be extended to classical logic by introducing consistent axioms [PdAC+25].
- **Isabelle**: A HOL-based proof assistant known for its flexibility and extensive library of pre-built theorems [Isa]. Isabelle offers strong for proof automation such as the `sledgehammer` tactic [BDP24].
- **Lean**: A dependent type-based proof assistant designed for both academic research and practical software verification [Lea]. Lean is taught in the Department of Mathematics at Imperial College London [Buz].

Proof assistants have been applied to a variety of well-known projects, demonstrating their ability to ensure correctness in both mathematics and software development. Two notable examples include:

- **Four Colour Theorem**: One of the most famous uses of proof assistants is the verification of the Four Colour Theorem [Gon08]. This theorem, which states that any map can be coloured with no more than four colours such that no two adjacent regions share the same colour, was proven using the Coq proof assistant [Gon08].
- **CompCert**: Another prominent example is the CompCert project, which uses the Coq proof assistant to implement a formally verified C compiler [Ler09]. The compiler is fully specified, programmed and proven within Coq, ensuring that the generated machine code adheres to the semantics of the C programming language.

2.4 Coq (Rocq)

Coq (Rocq) is a proof assistant designed for the formalisation of mathematical proofs and the development of certified software [Inr25a]. It consists of three main languages — Vernacular, Gallina and Ltac.

Vernacular is the command language for interacting with the proof assistant [Inr18d]. It allows users to issue commands for defining variables, stating theorems and constructing proofs, in order to guide the internal proof state.

Gallina is the functional programming language used for defining theorems, functions and data structures [Inr18a]. It operates within the dependent type system and supports inductive types, recursion and higher-order functions, making it suitable for formalising mathematical theories.

Ltac is the tactical language for proof construction [Inr18c]. It includes tactics such as `apply` and `induction`, which are used in proof mode to construct proofs interactively. Tactics may employ backward reasoning to manipulate goals, or forward reasoning to manipulate hypotheses [Inr21b].

A proof environment in Coq consists of **goals** and **local contexts**. A **goal** represents the statement to be proven, which may be reduced into smaller subgoals as the proof progresses. A **local context** contains hypotheses and local definitions available for use in the proof.

As an example, Figure 2.1 demonstrates the proof of the associativity property for list concatenation in Coq.

```
Theorem app_assoc :
  forall (T : Type) (l1 l2 l3 : list T),
  (l1 ++ l2) ++ l3 = l1 ++ (l2 ++ l3).
Proof.
  intros T l1 l2 l3.
  induction l1 as [| n l1' IHl1'].
  - reflexivity.
  - simpl. rewrite -> IHl1'. reflexivity.
Qed.
```

Figure 2.1: Example of list associativity proof in Coq

The proof proceeds by induction on the list `l1`. It begins with the **`intros`** tactic, which introduces the variables `l1`, `l2` and `l3` into the local context. The **`induction`** tactic is then applied to `l1`, dividing the proof into two cases: the base case, where `l1` is the empty list (`nil`), and the inductive step, where `l1` is a non-empty list (`cons n l1'`). In the base case, the goal is resolved by the **`reflexivity`** tactic, which solves the goal by reflexivity of equality after simplification. In the inductive step, the **`rewrite`** tactic is used to incorporate the inductive hypothesis (`IHl1'`) into the goal.

The core of Coq is its dependent type system, based on the Curry-Howard correspondence, which establishes a link between logic and computation. In this system, propositions are treated as types, and proofs as terms inhabiting those types. Therefore, verifying the validity of a proof is reduced to type-checking the corresponding term within the dependent type system [PdAC+25].

2.5 SSReflect

SSReflect (Small Scale Reflection) is a powerful extension of Coq that enhances its syntax and proof development capabilities [Inr18b]. Originally developed for the mechanised proof of the Four Colour Theorem [Gon08], SSReflect introduces concise yet expressive constructs designed to facilitate proof development [GR09].

SSReflect offers explicit control over the movement of hypotheses through its bookkeeping tacticals [Inr18b]. For instance, `tactic: a b c` moves hypotheses from the local context to the goal, while `tactic=> a b c` moves hypotheses from the goal to the local context. These tacticals offer minimal syntax for managing hypotheses without relying on tactics like **`intros`** at each step.

SSReflect further enhances Ltac by introducing a more flexible syntax. For instance, tactics such as **`have`** no longer require brackets (open syntax) [Inr18b], resulting in more concise proof. This syntactic improvement makes proofs easier to read and write, thereby improving the user experience by reducing verbosity.

SSReflect also extends the Gallina syntax. It introduces pattern assignment and pattern conditional syntax, such as `let: (x, (y, z)) := t` and `if t is Some _ then true ⌟ else false`, allowing users to destructure nested patterns directly without relying on **`match`** [Inr18b].

To structure proofs more effectively, SSReflect provides terminating tacticals such as `by` and `done` [Inr18b]. These tacticals work by ensuring an error is triggered immediately if the given tactic fails to solve its current goal. This explicitly marks the end of the proof for each subgoal and improves readability when replaying proofs interactively.

Alongside other features not discussed here, these syntactic and tactical improvements significantly enhance the clarity, efficiency and maintainability of proofs in Coq.

As an example, Figure 2.2 presents the proof of the same associativity property for list concatenation in SSReflect, which we previously proved in Figure 2.1.

```
Theorem app_assoc :
  forall (T : Type) (l1 l2 l3 : seq T),
  (l1 ++ l2) ++ l3 = l1 ++ (l2 ++ l3).
Proof.
  move=> T l1 l2 l3.
  elim: l1 => [|n l1' IHl1'] //=. by rewrite IHl1'.
Qed.
```

Figure 2.2: Example of list associativity proof in SSReflect

This SSReflect proof leverages the bookkeeping tacticals => and : to manage hypotheses. It also uses the simplification item //= to automatically simplify and solve subgoals generated by the tactic. In this case, the base case is shown trivially, leaving only the inductive case to be proven.

2.6 WebAssembly

2.6.1 Concepts

The following outlines key concepts of Wasm that are relevant to this project [Wor22a].

Values (value): Wasm defines only four basic number types (`i32`, `i64`, `f32` and `f64`), all of which are available in common hardware [HRS+17]. Floating point numbers are represented in IEEE 754 format [IEE19]. Each value type has a corresponding default value, primarily used to initialise local variables during a function call.

Types (type): Wasm specifies types for values, functions, memories, tables, globals and external values. These types are checked during validation, instantiation and potentially at runtime for certain instructions like call_indirect. The limits type defines the minimum and maximum sizes for a memory or table.

Instructions (instr): Wasm executes instructions within an abstract stack-based machine. Each instruction performs computations by popping input values from, and pushing output values to, an implicit operand stack [HRS+17].

Modules (module): Wasm organises static definitions for functions, globals, tables and memories within a module. A module must define a vector types of all function types used within it, which are indexed by instructions and other constructs. Modules can also import definitions from other modules [HRS+17].

Functions (func): In Wasm, each function is declared with a specific function type, which specifies the sequence of input and output values. Functions may contain recursive calls but cannot be nested within one another [HRS+17].

Globals (global): In Wasm, each global variable is declared as either mutable or immutable. Globals must have an initialiser, which is a constant expression permitted to access only other global variables.

Tables (table): In Wasm, a table stores references to functions and other constructs. Tables are primarily used for dynamic dispatch via the call_indirect instruction, which enables features such as virtual functions in C++ [cpp].

Memories (mem): In Wasm, a memory is a contiguous, mutable array of raw bytes used for data storage. It can be dynamically resized at runtime in units of 64 KiB, which corresponds to the least common multiple of the minimum page sizes on modern hardware [HRS+17].

Control Flow: Wasm handles control flow differently from traditional low-level bytecode languages. Instead of permitting unrestricted jumps, it provides structured control flow similar to those found in high-level programming languages. This ensures that control flow cannot introduce irreducible loops, which are difficult to optimise, or misaligned stack heights at branch points, which would require additional bookkeeping [HRS+17].

Figure 2.3 presents a (contrived) example illustrating the reduction of the loop instruction. This instruction is reduced to a label instruction, which includes the continuation to be executed upon exiting the loop.

$$\begin{aligned}
&(\mathsf{f32.const}\ (-1.0))\ (\mathsf{loop}\ ([\mathsf{i32}] \to [\mathsf{i32}])\ (\mathsf{f32.abs})\ \mathsf{end})\\
&\hookrightarrow \mathsf{label}_1\{\mathsf{loop}\ ([\mathsf{i32}] \to [\mathsf{i32}])\ (\mathsf{f32.abs})\ \mathsf{end}\}\ (\mathsf{f32.const}\ (-1.0))\ (\mathsf{f32.abs})\ \mathsf{end}\\
&\hookrightarrow \mathsf{label}_1\{\mathsf{loop}\ ([\mathsf{i32}] \to [\mathsf{i32}])\ (\mathsf{f32.abs})\ \mathsf{end}\}\ (\mathsf{f32.const}\ (+1.0))\ \mathsf{end}\\
&\hookrightarrow (\mathsf{f32.const}\ (+1.0))
\end{aligned}$$

Figure 2.3: Example of reduction of control instruction in Wasm

The execution of a Wasm module consists of three phases [Wor22a]:

1. **Decoding**: This phase parses the Wasm module from its binary format into an abstract representation used by the validation and execution phases.
2. **Validation**: This phase ensures the correctness of the Wasm module by verifying that values, types, instructions and other static definitions are well-formed.
3. **Execution**: This phase begins with **instantiation**, which creates a runtime representation of the module. It then proceeds to **invocation**, where the exported functions of the Wasm module are called by the host environment.

The relevant definitions from the structure, validation and execution sections of the Wasm 2.0 specification can be found in Appendix A.

2.6.2 Validation

Validity is determined using a type system applied to the module's abstract syntax [Wor22a]. This is specified in terms inference rules, with the premises above the horizontal bar and the conclusion below.

A **context** (context) serves as a typing environment that contains all necessary information required to define the constraints of each judgement [Wor22a]. Notable components of a context include the following:

- **Labels**: The labels component represents the stack of result types corresponding to the surrounding labels. The result type of a label denotes the types of operands expected on the stack when exiting the label. This stack of result types is accessed using a form of de Bruijn index, where index i refers to the i-th innermost label.

- **Return**: The return component specifies the optional return type of the current function. It may be absent in cases where no return is permitted, such as within initialisers of globals.

The following briefly explains key typing judgements [Wor22a]:

Constant Instruction (T-const): A constant instruction is valid with the type $[] \to [t]$, where $[t]$ denotes the type of the value pushed onto the stack.

Numeric Instructions (T-numeric): A unary or binary operation instruction is valid with types $[t] \to [t]$ and $[t\ t] \to [t]$, respectively.

Parametric Instructions (T-parametric): The drop and select instructions are valid with types $[t] \to []$ and $[t\ t\ \mathsf{i32}] \to [t]$, respectively. These instructions are polymorphic over the type t [Wor22a].

Variable Instructions (T-variable): A variable instruction is valid if the type of the variable matches its corresponding entry in the context C. Furthermore, the `global.set` instruction requires the variable to be declared as mutable.

Control Instructions (T-control): The nop instruction is valid only with type $[] \to []$. The unreachable instruction is valid with type $[t_1^*] \to [t_2^*]$. These instructions are polymorphic over any stack type [Wor22a].

The block and loop instructions are typed according to their underlying instruction sequences. The notation $C, \mathsf{label}\ [t_2^*]$ denotes a record update, where the label component of the context C is updated by pushing $[t_2^*]$ onto the label stack.

The br instruction is valid if the expected result type of the label, specified by the label index l, matches the types of the topmost values on the stack. The validity of the return instruction is defined similarly.

Types (T-type): Wasm also validates types, as some composite types may be ill-formed. For instance, the judgement for the limits type imposes the constraint that the minimum size cannot exceed the maximum size, and that neither the minimum nor maximum size may exceed the specified upper bound k.

2.6.3 Execution

Execution is specified using small-step operational semantics. This is expressed through clausal rules, with the conclusion on the left and the premises as side conditions on the right [Spe25b].

A **configuration** $S; F; \mathit{instr}^*$ represents the state of a program during its execution. It consists of the store S, the frame state F and the sequence of instructions instr^* to be executed. The pair of the frame state F and the instruction sequence instr^* is referred to as a **thread** [Wor22a].

A **store** S (store) holds the global state of the execution. It contains the **instances** (inst) of all functions, tables, memories and globals across all modules. These instances serve as the runtime, stateful representations of their corresponding definitions [Wor22a].

A **frame state** F (framestate) refers to the local state of the function that is currently executing. It consists of the values of the local variables and the module instance to which the function belongs [Wor22a].

A **module instance** (moduleinst) maintains a mapping from static indices, which are used to refer to definitions, to dynamic addresses, which are used to access instances in the store S. This mapping is necessary because instructions can only access instances that are defined or imported within the current module.

Furthermore, instructions are extended to include **administrative instructions** (admininstr), as detailed below. These instructions model intermediate steps of execution and cannot appear in the Wasm program itself [Wor22a].

- **Trap Instruction**: The trap instruction represents the occurrence of a trap, which may be triggered by instructions such as unreachable.
- **Label Instruction**: The label instruction denotes a label on the control stack. The subscript n indicates the arity of the label — the number of values expected on a branch. It also includes a continuation, which is executed upon exiting the label.
- **Frame Instruction**: The frame instruction represents a frame on the call stack. The subscript n specifies the arity of the frame — the number of values expected upon return. It also carries the frame state F of the corresponding function, allowing inner instructions to be reduced with respect to this frame state.
- **Invoke Instruction**: The invoke instruction serves as an intermediate step for both the call and call_indirect instructions, thereby unifying the handling of different forms of function calls.

To model the reduction of branch instructions, a **block context** $B^k[\_]$ (blockcontext) and its associated reduction rules (R-block) are introduced. This context represents the stack of labels surrounding the current instruction sequence [Wor22a].

Similarly, to model the reduction of a subsequence of instructions, an **evaluation context** $E[\_]$ (evalcontext) and its associated reduction rules (R-eval) [Wor22a] are introduced. This context effectively captures the call-by-value (CBV) reduction strategy.

Finally, values on the operand stack are represented by constant instructions in the reduction rules. This is convenient, as these intermediate values need not be distinguished from occurrences of the const instructions in the Wasm program.

The following provides a brief overview of key reduction rules [Wor22a]:

Numeric Instructions (R-numeric): A unary or binary operation instruction may trigger a trap if the operation is undefined for the given operand values. In contrast, test and relation operations never cause a trap.

Parametric Instructions (R-parametric): The drop and select instructions are reduced by dropping or selecting the operand values, respectively.

Variable Instructions (R-variable): Local variables are accessed through the frame state F, while global variables are accessed through the store S. The get instructions do not modify either the store or the frame state, unlike the set instructions.

Control Instructions (R-control): The nop and unreachable instructions reduce to an empty instruction sequence and a trap, respectively.

The block and loop instructions are both reduced to a label instruction, allowing branches to be handled by common reduction rules [Wor22a]. The block instruction has an empty continuation, whereas the loop instruction includes itself as the continuation.

If a branch occurs within a block context B^l, the control flow escapes all labels up to the one specified by the label index l and proceeds to execute the corresponding continuation.

Otherwise, the label exits with the values on the operand stack.

2.7 WasmCert-Coq

As this project focuses on mechanised proofs in Coq, we will only discuss the structure of WasmCert-Coq [WRPP+21] in this report.

2.7.1 Definitions

The structures defined in the Wasm specification [Wor22a] are manually translated into inductive definitions in Coq. Figure 2.4 presents the definitions for both the basic and administrative instructions.

```
Inductive basic_instruction : Type :=
| BI_unop : number_type -> unop -> basic_instruction
| BI_binop : number_type -> binop -> basic_instruction
...

Inductive administrative_instruction : Type :=
| AI_label : nat -> list administrative_instruction
    -> list administrative_instruction -> administrative_instruction
...
```

Figure 2.4: Definitions of instructions in WasmCert-Coq [BGP+25]

2.7.2 Typing Rules

The typing rules are represented as inductively defined relations in Coq, as shown in Figure 2.5. These relations are effectively dependent types indexed by the terms over which the relations are defined.

```
Inductive be_typing : t_context -> seq basic_instruction -> instr_type -> Prop :=
| bet_unop : forall C t op, unop_type_agree t op
    -> be_typing C [::BI_unop t op] (Tf [::T_num t] [::T_num t])
| bet_binop : forall C t op, binop_type_agree t op
    -> be_typing C [::BI_binop t op] (Tf [::T_num t; T_num t] [::T_num t])
...
```

Figure 2.5: Definitions of typing rules in WasmCert-Coq [BGP+25]

2.7.3 Reduction Rules

The reduction rules are similarly defined as inductively defined relations in Coq, as presented in Figure 2.6. The `reduce_simple` relation represents the reduction rules for simple instructions that do not interact with the store or frame state, while the `reduce` relation models the remaining reduction rules.

```
Inductive reduce_simple :
seq administrative_instruction -> seq administrative_instruction -> Prop :=
| rs_unop : forall v op t,
    reduce_simple [::$VN v; AI_basic (BI_unop t op)] [::$VN (@app_unop op v)]
| rs_binop_success : forall v1 v2 v op t, app_binop op v1 v2 = Some v ->
    reduce_simple [::$VN v1; $VN v2; AI_basic (BI_binop t op)] [::$VN v]
...

Inductive reduce :
host_state -> store_record -> frame -> list administrative_instruction ->
host_state -> store_record -> frame -> list administrative_instruction -> Prop :=
...
```

Figure 2.6: Definitions of reduction rules in WasmCert-Coq [BGP+25]

2.7.4 Soundness

The soundness of Wasm follows directly from the preservation and progress properties [Wor22a].

Informally, the preservation property states that the type of an instruction sequence is preserved before and after execution, and the progress property states that any non-terminal instruction sequence is always reducible.

To express these properties more formally, the typing rules are extended to include various forms of judgements that validate the structure of the configuration and its components:

The **configuration validity** is defined such that a configuration $S; F; instr^*$ is valid with type $[t^*]$ of the underlying instruction sequence, if the store S is valid and $instr^*$ is valid with the same type $[t^*]$ in terms of thread validity [Wor22a].

$$\textsc{Configuration Validity}\ \frac{\vdash S\ \mathsf{ok} \qquad S; \epsilon \vdash F; instr^* : [t^*]}{\vdash S; F; instr^*; [t^*]}$$

The **thread validity** is defined as valid with type $[t^*]$ if $instr^*$ is valid with type $[] \to [t^*]$ with respect to the context C determined by frame validity.

$$\textsc{Thread validity}\ \frac{S \vdash F : C \qquad S; C, \mathsf{return}\ resulttype^? \vdash instr^* : [] \to [t^*]}{S; resulttype^? \vdash F; instr^* : [t^*]}$$

The **store validity** is defined in terms of the validity of its components [Wor22a]. The **frame validity** types the frame state F with respect to the context C by extending the validity of a module instance [Wor22a]. The validity of individual instances is omitted here for brevity.

$$\textsc{Store validity}\ \frac{\begin{array}{c}(S \vdash \mathit{funcinst} : \mathit{functype})^* \qquad (S \vdash \mathit{tableinst} : \mathit{tabletype})^* \\ (S \vdash \mathit{meminst} : \mathit{memtype})^* \qquad (S \vdash \mathit{globalinst} : \mathit{globaltype})^* \\ (S \vdash \mathit{eleminst} : \mathit{elemtype})^* \qquad (S \vdash \mathit{datainst} : \mathit{datatype})^* \\ S = \{\mathsf{funcs}\ \mathit{funcinst}^*, \mathsf{tables}\ \mathit{tableinst}^*, \mathsf{mems}\ \mathit{meminst}^*, \\ \mathsf{globals}\ \mathit{globalinst}^*, \mathsf{elems}\ \mathit{eleminst}^*, \mathsf{datas}\ \mathit{datainst}^*\}\end{array}}{\vdash S\ \mathsf{ok}}$$

$$\textsc{Frame validity}\ \frac{S \vdash \mathit{moduleinst} : C \qquad (S \vdash \mathit{val} : t)^*}{S \vdash \{\mathsf{locals}\ \mathit{val}^*, \mathsf{module}\ \mathit{moduleinst}\} : (C, \mathsf{locals}\ t^*)}$$

Furthermore, the typing rules are also extended to handle administrative instructions. This is necessary because the preservation property concerns the validity of a contractum, which may include administrative instructions following reduction.

Finally, the **preservation** and **progress** properties can be formulated as follows [Wor22a]. The notation $S \preceq S'$ represents the **store extension** judgement, which ensures that reduction neither removes nor alters the types of its components. Additionally, a configuration is **terminal** if it consists of either a sequence of constant instructions or a trap.

Theorem 1 (Preservation)
If $\vdash S; F; \mathit{instr}^* : [t^*]$ *and* $S; F; \mathit{instr}^* \hookrightarrow S'; F'; \mathit{instr}^*$*,*
then $\vdash S'; F'; \mathit{instr}'^* : [t^*]$ *and* $S \preceq S'$*.*

Theorem 2 (Progress)
If $\vdash S; F; \mathit{instr}^* : [t^*]$*, then either* $S; F; \mathit{instr}^*$ *is terminal,*
or there exist $S', F', \mathit{instr}'^*$ *such that* $S; F; \mathit{instr}^* \hookrightarrow S'; F'; \mathit{instr}'^*$*.*

The **soundness** of the WebAssembly language then follows intuitively from these two properties. More informally, soundness states that every thread in a valid configuration either runs indefinitely, traps or terminates with a configuration of the expected type [Wor22a].

Corollary 1 (Soundness)
If $\vdash S; F; \mathit{instr}^* : [t^*]$*, then* $S; F; \mathit{instr}^*$ *either diverges*
or takes a finite number of steps to reach a terminal $S'; F'; \mathit{instr}'^*$
where $\vdash S'; F'; \mathit{instr}'^* : [t^*]$ *and* $S \preceq S'$*.*

2.8 SpecTec

2.8.1 Overview

Figure 2.7 provides an overview of the SpecTec architecture, inspired by the original SpecTec paper [YSL+24].

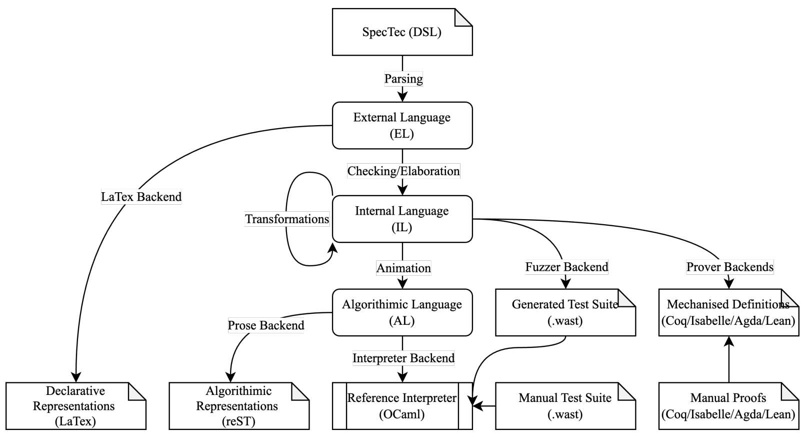
SpecTec (DSL)
Parsing
External Language (EL)
LaTex Backend
Checking/Elaboration
Transformations
Internal Language (IL)
Animation
Fuzzer Backend
Prover Backends
Algorithimic Language (AL)
Generated Test Suite (.wast)
Mechanised Definitions (Coq/Isabelle/Agda/Lean)
Prose Backend
Interpreter Backend
Declarative Representations (LaTex)
Algorithimic Representations (reST)
Reference Interpreter (OCaml)
Manual Test Suite (.wast)
Manual Proofs (Coq/Isabelle/Agda/Lean)

Figure 2.7: Overview of SpecTec architecture inspired by SpecTec paper [YSL+24]

The DSL is initially parsed into the external language (EL), which is essentially an abstract syntax tree of the DSL. The EL is then elaborated into the internal language (IL), which specifies details absent in the EL syntax. The IL may subsequently be reduced to lower-level representations, such as the algorithmic language (AL) [YSL+24].

The following summarises the key artefacts generated by SpecTec [YSL+24]:

- **Declarative Representations**: The LaTeX backend generates declarative representations directly from the EL. A simple template mechanism using splice commands allows integration with external text input files for generating the final LaTeX source for the documentation.
- **Algorithmic Representations**: The prose backend converts the AL into algorithmic representations written in prose notation. These complement the declarative style of the Wasm specification by offering a more readable, implementation-oriented view of the semantics.
- **Reference Interpreter**: The interpreter backend produces a reference interpreter in OCaml from the AL.
- **Executable Tests**: The fuzzer backend is responsible for generating comprehensive test suites via fuzzing in Wasm text format from the IL.
- **Mechanised Definitions**: The theorem prover backend is responsible for generating theorem prover code from the IL. Currently, IL2Coq is capable of translating IL definitions into inductive definitions in Coq.

The complete grammars of the SpecTec DSL and IL are provided in Appendix B for reference.

2.8.2 Domain Specific Language

At its core, the domain-specific language (DSL) serves as a single source of truth from which all key artefacts of the Wasm specification can be auto-generated. The DSL has a notation-heavy syntax that closely mirrors the formal style of the Wasm specification, which is inspired by traditional pen-and-paper notations used in programming language semantics [YSL+24].

The following outlines the key top-level constructs in the DSL that are relevant to this project. Further details and examples can be found in the documentation [Spe25b].

Syntax types are used to define the abstract syntax of the language and its auxiliary constructs. They support various forms of type definitions, including type aliases, constructor types, variant types, range types and record types [Spe25b].

```
syntax valtype =
  | I32 | I64 | F32 | F64

syntax Inn = I32 | I64
syntax Fnn = F32 | F64
```

Figure 2.8: Examples of syntax definitions in DSL [YSL+25]

Relations and their associated **rules** are used to define typing, evaluation and other predicates. These relations are declared with a type that specifies their custom notation, with the corresponding rules often accompanied by premises that specify side conditions [Spe25b]. Symbols like -> and ~> are called atoms and may be used to form parts of custom notations [Spe25b].

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

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

rule Step_pure/nop:
    NOP  ~>  eps
```

Figure 2.9: Examples of relation definitions in DSL [YSL+25]

Functions are used to define auxiliary definitions within the Wasm specification. They consist of a declaration followed by individual clauses. These clauses may perform pattern matching on the parameter position and may also specify premises that serve as guards for the pattern [YSL+24].

```
def $funcaddr(state) : funcaddr*
def $funcaddr((s; f)) = f.MODULE.FUNCS

def $funcinst(state) : funcinst*
def $funcinst((s; f)) = s.FUNCS

def $func(state, funcidx) : funcinst
def $func((s; f), x) = s.FUNCS[f.MODULE.FUNCS[x]]
```

Figure 2.10: Examples of function definitions in DSL [YSL+25]

2.8.3 External Language

The external language (EL) refers to the abstract syntax tree of the DSL[YSL+24], which explicitly represents operator precedence and other syntactic elements in its syntax. As the syntax of the EL is almost identical to that of the DSL, its details are omitted.

2.8.4 Internal Language

The purpose of the internal language (IL) is to explicitly represent details that are absent in the DSL, thereby facilitating its use in downstream translation. A notable example is expressions in the IL, which are annotated with their corresponding types, although these annotations are not reflected in the grammar.

The syntax of the IL largely mirrors that of the DSL. Note that the IL syntax does not necessarily have a conflict-free textual grammar, as the IL is not intended to be parsed. If parsing is required, it can be represented as S-expressions in AST mode [Spe25a].

The following briefly highlights several distinct elements of the IL that are relevant to this project. Further details and examples can be found in the documentation [Spe25a].

Free variables in the IL are annotated by binds of the innermost construct [Spe25a], which consist of pairs of variable identifiers and their associated types, as demonstrated in Figure 2.11. The IL directly embeds this information within its syntax for simplicity, although most programming languages typically use a symbol table for this purpose.

```
rule Instr_ok/unop:
  C |- UNOP t unop_t : t -> t

rule unop{C : context, t : valtype, unop_t : unop_(t)}:
   `%|-%:%`(C, UNOP_instr(t, unop_t), `%->%`_functype([t], [t]))
```

Figure 2.11: Examples of binds in IL [YSL+25]

Custom notations in the DSL are represented by a sequence of expressions interleaved with atoms. In the IL, these sequences of atoms and expressions are explicitly separated [Spe25a], as shown in Figure 2.12. For instance, `@(Instr_ok: C |- instr : ft)|` in the DSL corresponds to `Instr_ok: `%|-%:%`(C, instr, ft)|` in the IL. The notation `` `%;%|-%:%` `` represents a "mixop", where `%` denotes the holes to be filled by the tuple `(C, instr, ft)`.

```
relation Instr_ok: context |- instr : functype
relation Instr_ok: `%|-%:%`(context, instr, functype)

relation Step_pure: admininstr* ~> admininstr*
relation Step_pure: `%~>%`(admininstr*, admininstr*)
```

Figure 2.12: Examples of mixops in IL [YSL+25]

Additionally, the IL explicitly annotates subsumptions from one type to another occurring in expressions, as illustrated in Figure 2.13. These subsumptions are inserted according to the subtyping rules of the DSL [Spe25b].

```
rule Step_pure/drop:
  val DROP  ~>  eps

rule drop{val : val}:
    `%~>%`([(val : val <: admininstr) DROP_admininstr], [])
```

Figure 2.13: Examples of subsumptions in IL [YSL+25]

2.8.5 IL2Coq

IL2Coq is a proof of concept solution to the theorem prover backend of the SpecTec toolchain, which is capable of producing the inductive definitions from the DSL source [Cup24].

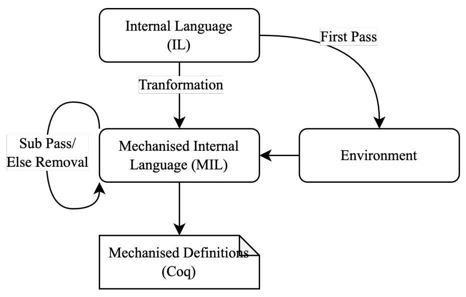

Figure 2.14: Overview of IL2Coq design inspired by IL2Coq report [?]

Figure 2.14 provides an overview of the IL2Coq design, inspired by Diego's report [Cup24].

The IL is translated into the mechanised internal language (MIL) through the main transformation pass. Although the MIL is currently specific to Coq, it is intended to be generalised to other theorem provers such as Isabelle/HOL [Isa].

IL2Coq has three auxiliary passes — the environment pass, the sub pass and else removal [Cup24]. In the environment pass, the IL is scanned to generate an environment that collects information necessary for the main transformation and other auxiliary passes. The sub pass generates explicit subtyping conversion functions. The else removal pass replaces occurrences of `otherwise` premises with predicates that explicitly negate the corresponding conditions.

```
Inductive instr  : Type :=
  | instr__UNOP (v_valtype : valtype) (v_unop_ : unop_) : instr
  | instr__BINOP (v_valtype : valtype) (v_binop_ : binop_) : instr
  ...

Inductive Step_pure: (list admininstr) -> (list admininstr) -> Prop :=
  | Step_pure__unreachable : Step_pure [(admininstr__UNREACHABLE )]
  ↪  [(admininstr__TRAP )]
  ...

Fixpoint fun_min (v_reserved__nat_0 : nat) (v_reserved__nat_1 : nat) : nat :=
  match (v_reserved__nat_0, v_reserved__nat_1) with
    | (0, v_j) => 0
    | (v_i, 0) => 0
    | ((S v_i), (S v_j)) => (fun_min v_i v_j)
  end.
```

Figure 2.15: Examples of auto-translated definitions in IL2Coq [Cup25]

Figure 2.15 illustrates the results of the Coq translation produced by IL2Coq. Syntax and relation definitions in the DSL are translated into inductive definitions in Coq, while function definitions are translated into (recursive) definitions.

Chapter 3

Project

3.1 Progress Proof

The first milestone of this project is the mechanised proof of the progress property. As stated in Theorem 2, the progress property asserts that any valid configuration is either in terminal form or can step to another configuration. Together with Diego's previous work on the preservation proof [Cup24], the proofs of these two properties establish the soundness of Wasm.

WebAssembly is distinguished by its rigorous formalisation, as evidenced by the soundness theorems included within its specification [Wor19] — an uncommon feature in the documentation of most programming languages. In Wasm, formal proofs plays a crucial role in the iterative feedback loop between language design and specification [Ros25].

The development of formal proofs in Wasm is therefore not only an academic exercise but also a direct contribution to the design process of the language. Consequently, these proofs must be produced to a high standard of quality in order to facilitate future maintenance and development.

The progress proof is also beneficial for validating the DSL translation of the hand-written Wasm specification [Wor22a]. This is significant because individual backends may not always use every aspect of the DSL. For instance, the interpreter backend may disregard some reduction rules and instead hardcode functionality within the OCaml codebase. Mechanised proofs play a crucial role in detecting errors that might be missed by other artefacts.

This section therefore aims to develop the progress proof in Coq with respect to Wasm 1.0 specification [Wor19]. We will introduce the necessary modifications to the DSL while acknowledging the existing differences between the Wasm 1.0 specification and the DSL translation. Subsequently, we will proceed to develop the main proof, maintaining an SSReflect-native proof style.

3.1.1 Modifications to SpecTec DSL

Several modifications had to be made to the DSL translation of the Wasm 1.0 specification [Wor19] before the main proof could be developed. This was necessary because some parts of the translation, originally authored by Andreas, were incorrect or incompatible with the progress property.

A number of additions and modifications have already been made by Diego to complete the preservation proof [Cup24], which are not discussed in this report. Consequently, those presented in this section represent the remaining errors in the DSL that were not detected through the preservation proof.

The first notable modification is the addition of the `Step/ctxt-seq` rule, as shown in Figure 3.1, which was missing in the original version. The DSL translation lacked this base case of the "bubbling up" semantics (explained later) that permits the reduction of a sequence of instructions following the constants `val*` at the beginning. This omission was not detected by the interpreter backend, as it hardcodes this evaluation context within the OCaml codebase, nor by the preservation proof.

```
+rule Step/ctxt-seq:
+  z; val* admininstr* admininstr''* ~>  z'; val* admininstr'* admininstr''*
+  -- Step: z; admininstr* ~> z'; admininstr'*
```

Figure 3.1: Modifications to `Step/ctxt-seq` rule

Another notable modification, spanning many reduction rules, is the replacement of certain occurrences of `instr` with `admininstr`. This is necessary because the reduction rules that depend on other relations, such as those shown in Figure 3.2, do not consider cases in which an administrative instruction is reduced. These reduction rules would otherwise apply only in cases where all the instructions are basic, thereby excluding instructions such as label, frame and trap.

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

 rule Step/read:
-  z; instr*  ~>  z; instr'*
-  -- Step_read: z; instr* ~> instr'*
+  z; admininstr*  ~>  z; admininstr'*
+  -- Step_read: z; admininstr* ~> admininstr'*
```

Figure 3.2: Modifications to rules using `instr` instead of `admininstr`

Replacing `instr` with `admininstr` also helps to simplify the progress proof. To illustrate, consider the example shown in Figure 3.1. Applying the `Step_pure/trap-vals` reduction rule would otherwise require proving that the redex contains only `instr*` after the `TRAP`. This distinction between `admininstr*` and `instr*` in the trailing position is, however, not essential. WasmCert-Coq, for instance, simplifies such cases by directly using `admininstr*` — we follow the same approach here to streamline the proof development.

```
 rule Step_pure/trap-vals:
-  val* TRAP instr*  ~>  TRAP
-  -- if val* =/= eps \/ instr* =/= eps
+  val* TRAP admininstr*  ~>  TRAP
+  -- if val* =/= eps \/ admininstr* =/= eps
```

Figure 3.3: Modifications to `Step/trap-vals` rule

Additionally, we noticed that the `Step/ctxt-frame` rule assumed that the frame state `f'` remains unchanged after the reduction. This poses an issue, as it effectively prevents any change in both the local and global state of the current execution. We have therefore resolved this by allowing the frame state `f'` to change to `f''`, as shown in Figure 3.4.

```
  rule Step/ctxt-frame:
-   s; f; (FRAME_ n `{f'} admininstr*)  ~>  s'; f; (FRAME_ n `{f'} admininstr'*)
-   -- Step: s; f'; admininstr* ~> s'; f'; admininstr'*
+   s; f; (FRAME_ n `{f'} admininstr*)  ~>  s'; f; (FRAME_ n `{f''} admininstr'*)
+   -- Step: s; f'; admininstr* ~> s'; f''; admininstr'*
```

Figure 3.4: Modifications to `ctxt-frame` rule

Furthermore, we identified another inconsistency concerning the iteration dimensions of numeric instructions. The issue arises because these reduction rules, such as those demonstrated in Figure 3.5, assume that `$binop` returns either zero or one value. However, this contradicts the actual return type of `$binop`, as specified in Figure 3.6, which may yield zero or more values `val_(valtype)*`, including more than one.

Operator functions such as `$binop` may return zero or more values because certain floating-point operations can exhibit non-deterministic behaviour in the NaN payload, depending on the runtime implementation [Fog18]. For the purposes of the progress proof, we follow WasmCert-Coq's approach and disregard this non-determinism for simplicity [BGP^{+}25].

```
rule Step_pure/binop-val:
  (CONST t c_1) (CONST t c_2) (BINOP t binop)  ~>  (CONST t c)
  -- if $binop(t, binop, c_1, c_2) = c

rule Step_pure/binop-trap:
  (CONST t c_1) (CONST t c_2) (BINOP t binop)  ~>  TRAP
  -- if $binop(t, binop, c_1, c_2) = eps
```

Figure 3.5: Modifications to `Step_pure` rules handling `binop`

```
-def $binop(valtype, binop_(valtype), val_(valtype), val_(valtype)) : val_(valtype)*
+def $binop(valtype, binop_(valtype), val_(valtype), val_(valtype)) : val_(valtype)?
```

Figure 3.6: Modifications to declaration of `$binop` function

A further inconsistency concerning the iteration dimensions was identified between the typing rule and the reduction rule for `Admin_instr_ok/call_addr`. This was subsequently resolved by correcting the typing rule, as illustrated in Figure 3.7.

```
  rule Admin_instr_ok/call_addr:
-   S; C |- CALL_ADDR funcaddr : t_1* -> t_2*
-   -- Externvals_ok: S |- FUNC funcaddr: FUNC (t_1* -> t_2*)
+   S; C |- CALL_ADDR funcaddr : t_1* -> t_2?
+   -- Externvals_ok: S |- FUNC funcaddr: FUNC (t_1* -> t_2?)
```

Figure 3.7: Modifications to `Admin_instr_ok/call_addr` rule

Moreover, we observed that the reduction rule `Step_read/loop` incorrectly assumed that the resulting label instruction has an arity of zero, rather than matching the arity specified by the block type of the corresponding block instruction, as shown in Figure 3.8. This poses an issue, as the reduction rule cannot be applied when the block type is non-empty.

To address this, we introduced a premise explicitly associating the block type with its corresponding arity.

```
  rule Step_read/loop:
-   z; (LOOP t? instr*)  ~>  (LABEL_ 0 `{LOOP t? instr*} instr*)
+   z; (LOOP t? instr*)  ~>  (LABEL_ n `{LOOP t? instr*} instr*)
+   -- if t? = eps /\ n = 0 \/ t? =/= eps /\ n = 1
```

Figure 3.8: Modifications to `Step_read/loop` rule

Finally, another missing premise was identified in the typing rule of the memory instance, as shown in Figure 3.10. This constraint, `|b*| = n * 64 * $Ki`, is indeed specified in the Wasm specification but was absent from the DSL. This is a condition that enforces the invariant that the memory instance is always initialised and grown in multiples of 64 KiB — an assumption upon which instructions such as memory.size depend, as illustrated in Figure 3.10.

```
rule Step_read/memory.size:
  z; (MEMORY.SIZE)  ~>  (CONST (INN I32) n)
  -- if $(n * 64 * $Ki) = |$mem(z, 0).BYTES|
```

Figure 3.9: Definition of `Step_read/memory.size` rule [YSL+25]

```
  rule Memory_instance_ok:
    S |- {TYPE mt, BYTES b*} : mt
    -- if mt = `[n .. m]
+   -- if $(|b*| = n * 64 * $Ki)
    -- Memtype_ok : |- mt : OK
```

Figure 3.10: Modifications to `Memory_instance_ok` rule

3.1.2 Divergence from WasmCert-Coq

In addition to the modifications discussed above, several changes in the DSL translation that diverge from the Wasm specification must be acknowledged before presenting the main proof.

The most notable change is the omission of the block context (blockcontext), which is used to speicfy the reduction of br and return instructions, as well as the evaluation context (evalcontext), which controls the reduction of inner instructions within a context. In the DSL translation, both the block and evaluation context are replaced by an equivalent set of reduction rules implementing the "bubbling up" semantics, as illustrated in Figures 3.11 and 3.12.

Figure 3.11 illustrates how branch and return instructions are "bubbled up" by recursively decrementing the label index in the br instruction until the base case is reached. The return instruction similarly propagates up through enclosing labels until the innermost frame is encountered. This behaviour is effectively equivalent to a single reduction rule defined using the block context $B[\_]$.

Figure 3.12, in contrast, shows how inner instructions are reduced with respect to their surrounding labels and frames. Once again, reduction may be "bubbled up" through these enclosing contexts. The trap instruction is also propagated upward in a similar manner. This approach is likewise equivalent to a single reduction rule defined in terms of the evaluation context $E[\_]$.

The DSL translation uses this "bubbling up" semantics as it significantly simplifies the development of both the interpreter backend and, more critically, the mechanised proof. By eliminating the need to represent the block and evaluation contexts as separate data structures, the overall proof process becomes considerably more tractable. However, this also requires us to depart from the proof strategy used in WasmCert-Coq and develop an alternative approach tailored to this version.

```
rule Step_pure/br-zero:
  (LABEL_ n `{instr*} val'* val^n (BR 0) admininstr*)  ~>  val^n instr*
  -- if |val^n| = n

rule Step_pure/br-succ:
  (LABEL_ n `{instr*} val* (BR $(l+1)) admininstr*)  ~>  val* (BR l)

rule Step_pure/return-frame:
  (FRAME_ n `{f} val'* val^n RETURN admininstr*)  ~>  val^n
  -- if |val^n| = n

rule Step_pure/return-label:
  (LABEL_ n `{instr*} val* RETURN admininstr*)  ~>  val* RETURN
```

Figure 3.11: Bubbling up reduction rules that replace block context [YSL+25]

```
rule Step/ctxt-label:
  z; (LABEL_ n `{instr*} admininstr*)  ~>  z'; (LABEL_ n `{instr*} admininstr'*)
  -- Step: z; admininstr* ~> z'; admininstr'*

rule Step/ctxt-frame:
  s; f; (FRAME_ n `{f'} admininstr*)  ~>  s'; f; (FRAME_ n `{f''} admininstr'*)
  -- Step: s; f'; admininstr* ~> s'; f''; admininstr'*

rule Step_pure/trap-label:
  (LABEL_ n `{instr*} TRAP)  ~>  TRAP

rule Step_pure/trap-frame:
  (FRAME_ n `{f} TRAP)  ~>  TRAP
```

Figure 3.12: Bubbling up reduction rules that replace evaluation context [YSL+25]

Additionally, we must note that numeric functions are assumed to always succeed in the preservation proof, since their implementations are currently unspecified in the SpecTec DSL. This is demonstrated in Figure 3.13, where the generated code is defined as an `Axiom` that holds for arbitrary inputs. This assumption will also be necessary for the progress proof.

```
Axiom fun_iadd : forall (v_reserved__N_0 : reserved__N) (v_iN_1 : iN) (v_iN_2
↪  : iN), iN.
```

Figure 3.13: Auto-translated definition of `$iadd` function in Coq [Cup25]

Furthermore, we assume the axiom `val_wf`, as given in Figure 3.14, which is necessary to ensure that constant values are well-formed with respect to their type. This requirement arises from a limitation in the current implementation of IL2Coq — the dependent argument of `val_` in the definition of `val` is ignored in the corresponding Coq definition of `val`, as illustrated in Figures 3.15 and 3.16 respectively. As a consequence, the resulting inductive definition does not guarantee that the `v_val_` and `v_valtype` have matching types.

The lack of this well-formedness property prevents us from applying some reduction rules. To address this, we introduce the `val_wf` axiom as a temporary solution. This is an acceptable compromise as the issue is expected to be resolved by the monomorphisation feature in the near future.

```
Definition is_val_wf (c : val) : Prop :=
  match c with
    | val__CONST (valtype__INN inn) (val___inn__entry n) => True
    | val__CONST (valtype__FNN fnn) (val___fnn__entry n) => True
    | _ => False
  end.

Axiom val_wf : forall (v : val), is_val_wf v.
```

Figure 3.14: Definition of `val_wf` axiom in Coq

```
syntax val =
  | CONST valtype val_(valtype)
```

Figure 3.15: Definition of `val` in SpecTec DSL [YSL+25]

```
Inductive val  : Type :=
  | val__CONST (v_valtype : valtype) (v_val_ : val_) : val .
```

Figure 3.16: Auto-translated definition of `val` in Coq [Cup25]

Finally, some minor updates have been made to IL2Coq, such as replacing conjunctions of premises with nested implications in inductive constructors and resolving the incorrect handling of nested iteration types. These details are omitted in this section, as they are not essential for explaining the proof structure.

3.1.3 Proof Style

In order to make the proofs clean and maintainable, we put significant effort into ensuring that the progress proof is written in an SSReflect-native style. This is in contrast to the proofs in WasmCert-Coq, which use a mixture of Coq and SSReflect syntaxes. In addition to the consistent use of bookkeeping tacticals and `by` terminators [Inr18b], the following stylistic conventions have been maintained throughout the progress proof:

- Use `move` tactic over `intros` and `generalize` tactics
- Use `case` tactic over `destruct`, `injection` and `inversion` tactics
- Use `elim` tactic over `induction` tactic
- Use `rewrite` tactic over `unfold` and `subst` tactics
- Use `have` tactic over `assert` tactic
- Use `set` tactic over `remember` tactic
- Use `do` tactical over `repeat` tactical
- Use `done` tactic or `by` tactical over `reflexivity` and `discriminate` tactics
- Use simplification items over `simpl` tactic
- Use view mechanisms of `move` tactic over `specialize` tactic

However, there are some exceptions to these conventions.

The first notable exception concerns the use of the `inversion` tactic [Inr25a], for which there is no direct counterpart in SSReflect. We would otherwise need to manually perform the necessary bookkeeping prior to applying the `case` tactic, which can result in considerably more laborious proofs.

In addition, we permit the occasional use of tactics such as `destruct` and `subst` [Inr25a] within custom Ltac definitions. This is because their SSReflect counterparts require us to explicitly name each newly introduced variable using the `let x := fresh "x" in` syntax, which can quickly become cumbersome.

The last exception concerns the use of the `apply term with (ident := term)` syntax [Inr25a]. SSReflect does not support this syntax because the identifier `ident` is, in theory, subject to alpha-renaming at any point during proof mode. Nevertheless, we allow the use of this syntax because the progress proof involves theorems and lemmas with a large number of dependent arguments, which makes it impractical to specify them all explicitly.

3.1.4 Proof Structure

A large part of the high-level proof structure, specifically the formulation of the auxiliary lemmas, is based on the proofs in WasmCert-Coq [BGP+25]. The main exception is the handling of the block and evaluation contexts, which must be reformulated to accommodate the equivalent reduction rules adopted in the DSL translation.

The statements of these lemmas closely mirror those in WasmCert-Coq, but we have made a deliberate effort to rewrite their individual proofs from scratch. This ensures that the proofs are not simply copied and pasted without understanding their logical flow, as such an approach would likely result in missed opportunities for refactoring and lead to cumbersome code that fails to account for the subtle differences between the specification and the DSL translation. Moreover, this approach helps us to reinforce the proof styles discussed earlier in a more reliable manner.

Figure 3.17 presents the dependency graph of the auxiliary lemmas, with the progress property shown at the very bottom. While we omit the actual proofs and some of the definitions in this report, interested readers may refer to the code available in the repository.

Note that the statements of these theorems and lemmas may contain some inconsistent naming, as they are derived from the auto-generated definitions of IL2Coq, which have yet to be improved.

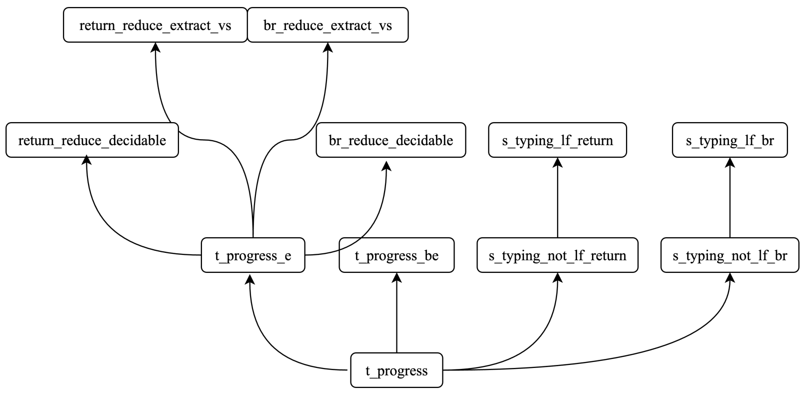

Figure 3.17: Dependency graph of auxiliary lemmas in progress proof

The theorem `t_progress`, as shown in Figure 3.18, is the Coq formulation of the progress property. The predicate `terminal_form` specifies that the instructions are either constants or a single `trap`, as defined in Figure 3.19. This theorem depends on four major auxiliary lemmas: `t_progress_e`, `t_progress_be`, `s_typing_not_lf_return` and `s_typing_not_lf_br`. The first two establish that the progress property holds for administrative and basic instructions, respectively, under the assumption that no `br` or `return` instructions appear at the top level. The latter two show that this assumption indeed holds true, provided that the configuration is valid.

```
Theorem t_progress :
  forall s f es ts,
  Config_ok (config__ (state__ s f) es) ts ->
  terminal_form es \/
  exists s' f' es',
  Step (config__ (state__ s f) es) (config__ (state__ s' f') es').
```

Figure 3.18: Definition of progress property adapted from WasmCert-Coq [BGP+25]

```
Definition terminal_form es :=
const_list es \/ es = [:: admininstr__TRAP].

Definition not_lf_br es :=
forall vcs l es',
es <> list__val__admininstr vcs ++ [:: admininstr__BR l] ++ es'.

Definition not_lf_return es :=
forall vcs es',
es <> list__val__admininstr vcs ++ [:: admininstr__RETURN] ++ es'.

Definition br_reduce es :=
exists vcs l es',
es = list__val__admininstr vcs ++ [:: admininstr__BR l] ++ es'.

Definition return_reduce es :=
exists vcs es',
es = list__val__admininstr vcs ++ [:: admininstr__RETURN] ++ es'.
```

Figure 3.19: Definitions of auxiliary predicates adapted from WasmCert-Coq [BGP+25]

The lemma `t_progress_e`, as given in Figure 3.20, is referred to as the (administrative) fragment progress property in the paper "Two Mechanisations of WebAssembly 1.0" by Conrad et al. This lemma essentially reformulates the progress property by assuming several conditions derived through inversion of the configuration validity. It also strengthens the statement by allowing appropriate `vcs` to be inserted in the redex, as well as permitting arbitrary `lab` and `ret` in the context `C`. This is necessary for some inductive steps, since the `es` may refer to any sub-instructions within a valid configuration.

The proof of `t_progress_e` proceeds by a triple mutual induction over three relations — `Admin_instr_ok`, `Admin_instrs_ok` and `Thread_ok`. The inductive predicates for `Admin_instr_ok` and `Admin_instrs_ok` are defined trivially. In contrast, the inductive predicate for `Thread_ok` requires fewer assumptions, as it follows directly from the inductive hypothesis on `Admin_instrs_ok`.

The predicates `not_lf_br` and `not_lf_return`, as given in Figure 3.19, enforce that no `br` or `return` instructions appear at the top level. This is crucial because, for any sub-instructions within a valid configuration, these instructions cannot be reduced independently but must be reduced as part of the enclosing label and frame instructions.

The reduction of a br or return instruction occurs only when the first non-constant instruction in a sequence is a br or return. This condition is captured by the predicates `br_reduce` and `return_reduce`, as shown in Figure 3.19. These predicates are equivalent to the negations of `not_lf_br` and `not_lf_return`, respectively.

Therefore, in the inductive steps for label and frame instructions, we must perform a case analysis on whether the reduction of a br or return instruction may occur or not. In manually written proofs, such case analysis would proceed immediately. In Coq, however, we must first establish that this case analysis is valid by proving that the logical truth of `br_reduce` and `return_reduce` is decidable. These decidability properties are stated in the lemmas `br_reduce_decidable` and `return_reduce_decidable`, as given in Figure 3.23.

In the case where the reduction of a br or return instruction occurs, we must further ensure that there are as many constants as the arity of the outermost label or frame instructions, respectively. This is established in the lemmas `br_reduce_extract_vs` and `return_reduce_extract_vs`, as noted in the diagram of Figure 3.17. The proofs of these lemmas proceed by inverting the administrative instruction validity and establishing a correspondence between the number of constants and the top element of the labels component of the context.

Note that these lemmas are completely reformulated from those in WasmCert-Coq to accommodate the omission of the block and evaluation contexts in the DSL translation. These parts of the proofs therefore represent the most original aspects of the progress proof in terms of its overall structure, although some similarities remain.

```
Lemma t_progress_e :
  forall s C C' f vcs es tf ts1 ts2 lab ret,
  Admin_instrs_ok s C es tf ->
  tf = functype__ ts1 ts2 ->
  C = (upd_local_label_return C'
      (map typeof f.(frame__LOCALS)) lab ret) ->
  Module_instance_ok s f.(frame__MODULE) C' ->
  map typeof vcs = ts1 ->
  Store_ok s ->
  not_lf_br es ->
  not_lf_return es ->
  terminal_form (list__val__admininstr vcs ++ es) \/
  exists s' f' es',
  Step (config__ (state__ s f) (list__val__admininstr vcs ++ es))
       (config__ (state__ s' f') es').
```

Figure 3.20: Definition of administrative fragment progress property adapted from WasmCert-Coq [BGP+25]

Similarly, the lemma `t_progress_e`, as presented in Figure 3.21, is referred to as the (basic) fragment progress property by Conrad et al [WRPP+21]. This lemma also reformulates the progress property by assuming several fine-grained conditions and strengthening the statement for certain inductive steps.

The proof of `t_progress_be` proceeds by mutual induction over the relations `Instr_ok` and `Instrs_ok`, sharing a structure similar to that of `t_progress_e`. While the inductive steps of `t_progress_be` are generally simpler than those of `t_progress_e`, some instructions, such as call__addr and memory__grow, require extensive case analysis.

```
Lemma t_progress_be :
  forall s C C' f vcs bes tf ts1 ts2 lab ret,
  Instrs_ok C bes tf ->
  tf = functype__ ts1 ts2 ->
  C = (upd_local_label_return C'
      (map typeof f.(frame__LOCALS)) lab ret) ->
  Module_instance_ok s f.(frame__MODULE) C' ->
  map typeof vcs = ts1 ->
  Store_ok s ->
  not_lf_br (list__instr__admininstr bes) ->
  not_lf_return (list__instr__admininstr bes) ->
  const_list (list__instr__admininstr bes) \/
  exists s' f' es',
  Step (config__ (state__ s f) (list__val__admininstr vcs ++
  ↪  list__instr__admininstr bes)) (config__ (state__ s' f') es').
```

Figure 3.21: Definition of basic fragment progress property adapted from WasmCert-Coq [BGP+25]

Finally, the lemmas `s_typing_not_lf_br` and `s_typing_not_lf_return`, as given in Figure 3.22, state that no br or return instructions appear at the top level following the constants, given that the thread constituting the original configuration is valid. These lemmas depend on `s_typing_lf_br` and `s_typing_lf_return`, as noted in the diagram of Figure 3.17, which assert that every instruction in es is neither a br nor a return instruction, regardless of the occurrences of the constants preceding them.

```
Lemma s_typing_not_lf_br :
  forall s rs f es ts,
  Thread_ok s rs f es ts -> not_lf_br es.

Lemma s_typing_not_lf_return :
  forall s f es ts,
  Thread_ok s None f es ts -> not_lf_return es.
```

Figure 3.22: Definitions of `br` and `return` progress adapted from WasmCert-Coq [BGP+25]

```
Lemma return_reduce_decidable :
  forall es, decidable (return_reduce es).

Lemma br_reduce_decidable :
  forall es, decidable (br_reduce es).
```

Figure 3.23: Definitions of decidability lemmas adapted from WasmCert-Coq [BGP+25]

3.1.5 Soundness

Now that the progress proof is complete, the soundness (Corollary 1) of the WebAssembly language can be established [Wor22a], in combination with the preservation proof by Diego [Cup24].

Soundness is a direct consequence of the preservation and progress properties, which is not intended to be verified by a mechanised proof. For this reason, the Wasm specification only describes this result in a semi-formal style [Wor22a], and WasmCert-Coq does not include its mechanisation at all [BGP+25]. This section therefore serves only as an experimental study, rather than a formal one.

Due to its semi-formal specification, the statement of soundness can be interpreted in various ways. In this section, we propose two possible, slightly more formal interpretation of Corollary 1, referred to as weak soundness (Corollary 2) and strong soundness (Corollary 3).

Corollary 2 (Weak Soundness)
If $\vdash S; F; instr^* : [t^*]$, then ***there exists a trace*** *starting from $S; F; instr^*$ such that it either diverges or takes a finite number of steps to reach a terminal $S'; F'; instr'^*$ where $\vdash S'; F'; instr'^* : [t^*]$ and $S \preceq S'$.*

Corollary 3 (Strong Soundness)
If $\vdash S; F; instr^* : [t^*]$, then ***for every trace*** *starting from $S; F; instr^*$, it either diverges or takes a finite number of steps to reach a terminal $S'; F'; instr'^*$ where $\vdash S'; F'; instr'^* : [t^*]$ and $S \preceq S'$.*

Weak soundness asserts that soundness holds for a particular trace starting from $S; F; instr^*$, while strong soundness requires soundness to hold for every trace starting from $S; F; instr^*$. This distinction is necessary because the execution of Wasm is not always deterministic [Wor22a]. In this section, we focus on establishing weak soundness, as strong soundness can be difficult to formulate intuitively.

The proof of weak soundness remains cumbersome, however, if we translate its statement directly into first-order (constructive) logic. Instead, we formulate this property as a co-inductive relation called `Config_sound`, as illustrated in Figure 3.24. This is an elegant solution originally proposed by Rao for WasmCert-Coq [BGP+25], which we have adapted for SpecTec.

The `Config_sound` relation represents a subset of the validity relation `Config_ok`. This soundness relation is co-inductively defined, meaning it may hold through infinite applications of the inductive case `Config_sound__`, even without establishing the base case `Config_sound__terminal`. The inhabitance of this relation corresponds to the existence of a trace that may either diverge or reach a terminal configuration of the same type after a finite number of steps.

```
CoInductive Config_sound s f es ts
  (Hconfig : Config_ok (config__ (state__ s f) es) ts) : Prop :=
  | Config_sound__terminal :
    terminal_form es ->
    Config_sound s f es ts Hconfig
  | Config_sound__step s' f' es'
    (Hconfig' : Config_ok (config__ (state__ s' f') es') ts) :
    Step (config__ (state__ s f) es) (config__ (state__ s' f') es') ->
    Config_sound s' f' es' ts Hconfig' ->
    Config_sound s f es ts Hconfig.
```

Figure 3.24: Co-inductive relation for weak soundness inspired by Rao's solution

The proof of weak soundness using this relation is presented in Figure 3.25. The proof proceeds by co-recursively establishing the inhabitance of the `Config_sound` relation. As we can observe, the proof follows trivially from the preservation and progress properties established previously, confirming that soundness is indeed a direct consequence of these two properties. Note that the store extension $S \preceq S'$ is omitted in this proof for brevity.

```
Theorem t_soundness :
  forall s f es ts (Hconfig : Config_ok (config__ (state__ s f) es) ts),
  Config_sound s f es ts Hconfig.
Proof.
  cofix Hsound. move=> s f es ts Hconfig.
  case Hprogress: (t_progress s f es ts Hconfig) => [Hterm | [s' [f' [es' Hstep]]]].
  - by apply: Config_sound__terminal.
  - have Hconfig' := t_preservation s f es s' f' es' ts Hstep Hconfig.
    by apply: (Config_sound__step s f es ts Hconfig s' f' es').
Qed.
```

Figure 3.25: Co-recursive proof of weak soundness inspired by Rao's solution

3.2 Decidable Equality Proofs

The next step of this project is to make each auto-translated data type `T` an instance of SSReflect's `EqType` [Inr25c], which would enable the use of the boolean equality operator `==`, the membership operator `\in` and the associated SSReflect lemmas for boolean equality.

These operators are essential for performing case analysis on equality of certain data types, which is required for some inductive steps in the progress proof. IL2Coq already generates instances for several utility type classes, namely `Inhabited` for providing default values and `Append` for concatenating record values [Cup25]. However, it lacked support for `Eq⌋Type` instances, and therefore we had to manually define `EqType` instances for some data types during the progress proof.

This section thus aims to fully automate this process as part of IL2Coq's auto-translation mechanism. We generate `EqType` instances automatically for data types defined using Coq's `Inductive`, `Record` and `Definition` commands, including automated proofs of decidable equality as required.

3.2.1 Non-Recursive Data Types

Figures 3.26 and 3.27 present an example of the auto-generated `EqType` instances for a non-recursive data type. In this section, we follow WasmCert-Coq's approach [BGP+25], which uses decidable equality proofs to automatically derive the corresponding `op` and `axiom op` definitions. These Coq definitions are generated by extending the rendering logic of IL2Coq, which is also responsible for generating instances of other type-classes [Cup25].

Decidable equality for a given data type `T` asserts that for any two values `v1` and `v2` of type `T`, either the proposition `v1 = v2` or `v1 <> v2` is true [PdAC+25]. This property is represented by the type `decidable (v1 = v2)`, which expands to the sum type `{v1 = v2} + {v1 <> v2}`.

A key characteristic of this sum is that it belongs to the sort `Type` rather than `Prop`, which means that it is computationally relevant. We can therefore pattern match against a proof term of the type `{v1 = v2} + {v1 <> v2}` at runtime, allowing us to extract a boolean equality function from the proof term. The resulting boolean equality function `func_eq⌋b` makes the derivation of the reflection principle `reflect (x = y) (op x y)` trivial, as demonstrated in `eq_dec_Equality_axiom`.

```
Create HintDb eq_dec_db.

Ltac decidable_equality_step :=
  do [ by eauto with eq_dec_db | decide equality ].

Lemma eq_dec_Equality_axiom :
  forall (T : Type) (eq_dec : forall (x y : T), decidable (x = y)),
  let eqb v1 v2 := is_left (eq_dec v1 v2) in Equality.axiom eqb.
Proof.
  move=> T eq_dec eqb x y. rewrite /eqb.
  case: (eq_dec x y); by [apply: ReflectT | apply: ReflectF].
Qed.
```

Figure 3.26: Header of `EqType` instances auto-generated by IL2Coq

```
Inductive func : Type := ...

Definition func_eq_dec : forall (v1 v2 : func),
  {v1 = v2} + {v1 <> v2}.
Proof. do ? decidable_equality_step. Defined.

Definition func_eqb (v1 v2 : func) : bool :=
  is_left (func_eq_dec v1 v2).
Definition eqfuncP : Equality.axiom (func_eqb) :=
  eq_dec_Equality_axiom (func) (func_eq_dec).

Canonical Structure func_eqMixin := EqMixin (eqfuncP).
Canonical Structure func_EqType :=
  Eval hnf in EqType (func) (func_eqMixin).

Hint Resolve func_eq_dec : eq_dec_db.
```

Figure 3.27: `EqType` instance auto-generated by IL2Coq for non-recursive data types

A major benefit of this approach is that the decidable equality proofs can be fully automated using the `decide equality` tactic. Given a goal of the appropriate form, the `decide equality` tactic destructs `v1` and `v2` and recursively applies itself until the equality or the inequality of the subterms can be established trivially.

Finally, data types in Coq are made instances of `EqType` through canonical structures [Inr21a]. Canonical structures are a powerful mechanism adopted in the MathComp library [Inr25c] for overload resolution, providing a form of ad-hoc polymorphism. They are closely integrated with Coq's unification engine, resulting in faster and more predictable resolution compared to typeclasses [Inr21a], which rely on proof search over the implicitly generalised arguments (until recently).

3.2.2 Recursive Data Types

In contrast to non-recursive types, the proof of decidable equality for recursive types, such as `instr` and `admininstr`, requires special handling. This is because, a repeated application of the `decide equality` tactic would simply result in an infinite loop, unlike the `func` type in the previous example.

Consider the block instruction `instr__BLOCK`, which consists of `seq instr` and `blocktyp`⌋
`e`. The `decide equality` tactic begins by destructing `instr__BLOCK`, reducing decidable equality on `instr__BLOCK` to that of `seq instr`. The tactic then infers that, to prove

decidable equality on `seq instr`, it suffices to prove that of `instr`. However, this is identical to the original goal, thereby causing an infinite loop.

```
Inductive instr : Type := ...

Fixpoint instr_eq_dec (v1 v2 : instr) {struct v1} :
  {v1 = v2} + {v1 <> v2}.
Proof. decide equality; do ? decidable_equality_step. Defined.

Definition instr_eqb (v1 v2 : instr) : bool :=
  is_left (instr_eq_dec v1 v2).
Definition eqinstrP : Equality.axiom (instr_eqb) :=
  eq_dec_Equality_axiom (instr) (instr_eq_dec).

Canonical Structure instr_eqMixin := EqMixin (eqinstrP).
Canonical Structure instr_EqType :=
  Eval hnf in EqType (instr) (instr_eqMixin).

Hint Resolve instr_eq_dec : eq_dec_db.
```

Figure 3.28: `EqType` instance auto-generated by IL2Coq for recursive types

Figures 3.26 and 3.28 illustrate an example of the auto-generated `EqType` instances for a recursive type. The key observation is that any proof making use of an inductive principle is fundamentally a recursive Gallina term, and can therefore be defined using `Fixpoint`. By defining the proof term with `Fixpoint` instead of `Definition`, the proposition `forall v1 v2, {v1 = v2} + {v1 <> v2}` is introduced as a hypothesis in the local context. This hypothesis can then be leveraged by the `decide equality` tactic to establish the decidable equality of the subterms of `instr`.

However, the use of `Fixpoint` alone is not sufficient because the `eauto with eq_dec_db` tactic attempts to solve the goal by applying any available hypotheses in the local context before using facts from the hint database. The `instr_eq_dec` would therefore apply the hypothesis `forall v1 v2, {v1 = v2} + {v1 <> v2}` before destructing `v1`, resulting in a proof term like `fun v1 v2 : instr => instr_eq_dec v1 v2`. To prevent such invalid recursion, we insert an additional call to `decide equality` at the start of the proof, ensuring that `v1` is structurally decreasing, as annotated by `{struct v1}`.

Finally, these decidable equalities are maintained in the custom hint database `eq_dec_db`. This is necessary because some instructions, such as `admininstr`, may reference recursive data types. Within the decidable equality proof for `admininstr`, `Fixpoint` introduces inductive hypotheses only for `admininstr`, and not for `instr`. This would force us to reconstruct the proof for `instr` from scratch, thereby causing an infinite loop once again.

3.3 First-Order Logic Extension

The major contribution of this project is the extension of the SpecTec toolchain with first-order logic. This section explores the design and implementation of first-order logic constructs, such as universal and existential quantifiers, as well as rule invocations within the SpecTec DSL [YSL+24].

WebAssembly is characterised by its comprehensive formalisation [Ros25], which is reflected in the soundness theorems included in the appendix of its specification [Wor19]. In addition to these theorems, the draft specification of Wasm 3.0 further introduces a number of key theorems, including principal types, the type lattice and the compositionality of instruction sequences [Wor22b].

The primary motivation for this extension is therefore to specify these theorems within the DSL. This approach allows us to express not only the existing theorems in the DSL but also any new theorems that may be introduced in the future. Given their importance, specifying these theorems within the SpecTec DSL is well aligned with its role as a "single source of truth."

This extension also allows the theorems specified in the DSL to be not only used for Coq translation but also to be rendered as LaTeX formulas and prose descriptions within the specification. This approach further strengthens the role of the SpecTec DSL as a "single source of truth," by ensuring that the theorems in the specification precisely match those used in the mechanised proofs.

Furthermore, a major benefit of these first-order logic constructs is that they enable the unified specification of theorems and auxiliary lemmas across different theorem provers. The proof structure of these theorems is otherwise left entirely to verification engineers, which can lead to inconsistent naming conventions, reformulations via corollaries and divergent proof strategies. We can mitigate such risks by maintaining these statements within the DSL.

```
premise ::=
  "var" id ":" typ              local variable declaration
  "if" exp                      side condition
  "othherwise"                  fallback side condition
  relid ":" exp                 relational premise
  "(" premise ")" iter*         iterated relational premise
```

Figure 3.29: Syntax of premises in SpecTec DSL [Spe25b]

It is important to note that the SpecTec DSL already provides a mechanism to express a subset of first-order logic (and thus propositional logic) via premises [Spe25b]. As illustrated in Figure 3.29, it supports expressing boolean expressions using the `"if"` syntax, rule invocations via the `ruleid ":" exp` syntax and iterated premises via the `"(" premise ")" iter*` syntax.

However, premises come with a critical restriction that they can only appear as side conditions in the top-level definitions and declarations, as specified in Appendix B.2. Consequently, they are not a generic mechanism that can be used as expressions elsewhere, which limits their power to express statements consisting of an arbitrary first-order logic formula.

This section thus aims to lift these premises to general expressions, while strengthening their expressibility by introducing universal and existential quantifiers. The syntax and semantics of the DSL, IL and MIL will be extended accordingly. We will then integrate the Coq backend to allow automatic translation of theorems, alongside the LaTeX, prose and splice backends to display these theorems in the specification in a human-friendly format. All of these changes will be implemented by extending SpecTec's OCaml codebase [YSL+25].

Figure 3.30 provides an overview of the parts of the SpecTec toolchain that have been extended in this section of the project. The interpreter and fuzzer backends remain largely untouched, as their integration is not essential for our purposes.

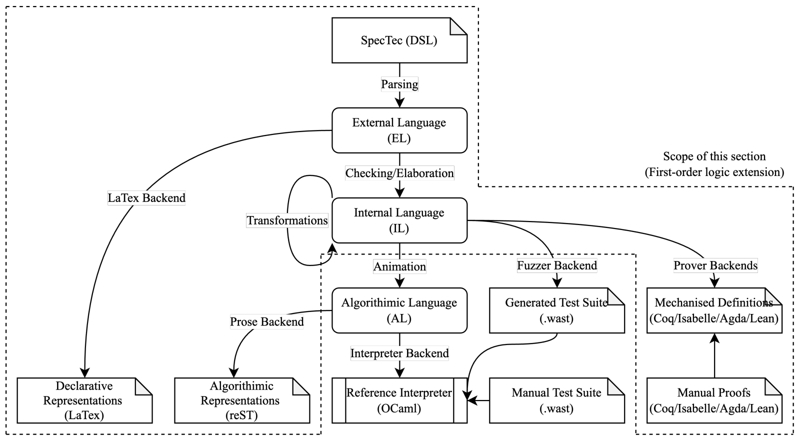
SpecTec (DSL)
Parsing
External Language (EL)
Checking/Elaboration
Scope of this section (First-order logic extension)
LaTex Backend
Transformations
Internal Language (IL)
Animation
Fuzzer Backend
Prover Backends
Algorithimic Language (AL)
Generated Test Suite (.wast)
Mechanised Definitions (Coq/Isabelle/Agda/Lean)
Prose Backend
Interpreter Backend
Declarative Representations (LaTex)
Algorithimic Representations (reST)
Reference Interpreter (OCaml)
Manual Test Suite (.wast)
Manual Proofs (Coq/Isabelle/Agda/Lean)

Figure 3.30: Scope of first-order logic extension in SpecTec toolchain inspired by SpecTec paper [YSL+24]

The design of new language features in the SpecTec DSL is guided by two core principles — maintaining backward compatibility and balancing the trade-off between readability and expressibility.

Backward compatibility is particularly crucial for these new constructs to be considered for adoption in the DSL. This is further emphasised by the fact that SpecTec is currently used not only by the Wasm specification but also by the specification of other languages, notably the P4 language [LR24], a domain-specific language designed for programming packet-processing network devices [Con24].

The trade-off between readability and expressibility is also a recurring concern when designing new features in the DSL. This is because the SpecTec DSL is a highly notation-heavy language that allows specification writers to use a wide range of symbols, such as `:`, `|-` and `->`, as part of custom notations. This design choice was deliberate to make the SpecTec DSL an ASCII-readable, human-friendly language [YSL+24] but poses significant challenges in terms of disambiguation. It is therefore important to acknowledge this trade-off as it imposes many restrictions that are uncommon in other programming languages.

3.3.1 Design Choices

This section discusses possible designs for the first-order logic extension, revisiting assumptions and exploring the designs from scratch.

DSL-Level or MIL-Level Definitions

There is a fundamental question that must be addressed before considering any other designs — whether theorems should be maintained in the DSL or in a lower-level representation, namely the MIL.

Maintaining theorems in the MIL has the advantage of not introducing first-order logic constructs into the DSL syntax. This is ideal for keeping the DSL syntax as minimal as possible, given that it is already notation-heavy. There will also be fewer restrictions in terms of design choices, since the MIL is tailored for use in theorem provers.

However, the MIL is unlikely to be a suitable format, since it lacks a parsable textual grammar. This is a deliberate design choice, as it would otherwise require a separate parser with a fully disambiguated grammar, and force specification writers to learn an entirely new syntax for writing theorems. This is despite the fact that the remainder of the specification can be fully expressed within the SpecTec DSL, which contradicts SpecTec's purpose as a "single source of truth" [YSL+24].

On the other hand, first-order logic constructs can be introduced into the DSL syntax with minimal modifications, thereby flattening the learning curve for specification writers. Additionally, the DSL approach makes it possible to render theorems and auxiliary lemmas in LaTeX and prose descriptions within the specification. This is only feasible in the DSL because the LaTeX and prose backends operate on the EL and IL/AL respectively, which cannot be reconstructed from the MIL.

We therefore conclude that the DSL is the appropriate format for this purpose.

Possible Designs for Basic Formulas

Basic formulas in this context refer to the subset of first-order logic constructs consisting of universal and existential quantifiers and rule invocations. Our goal is to design these basic formulas as first-class citizens, allowing them to be used more flexibly beyond premises.

Figure 3.31 presents the initial design of basic formulas. Basic formulas are modelled as a syntactically distinct entity by assigning the non-terminal `formula`. The `formula` contains `exp`, such that formulas form a strict syntactic superset of all boolean expressions.

The logical operators, namely `notop` and `logop`, are shared between the non-terminals `exp` and `formula`. This causes a reduce-reduce conflict between `exp` and `formula`. We therefore prioritise the production rules `notop formula` and `formula binop formula` over `exp` by placing the former before the latter. These logical operators must be overloaded for both expressions and formulas, in terms of their semantics.

The universal and existential quantifiers are followed by `args` and a sub-`formula`. The `args` is an existing non-terminal used for function calls and definitions B.1, which we reuse here for the quantifiers. We reserve the keywords `forall` and `exists` in addition to the existing keywords, although this may technically break backward compatibility.

Theorems and lemmas are defined using the `theorem` and `lemma` keywords, respectively. These top-level constructs specify theorem statements with `formula`, optionally annotated with hints. The `thmid` is an alias for `id` in the DSL.

Predicates are defined using the `formula` keyword, closely resembling function definitions. While functions map arguments to an `exp`, predicates map arguments to a `formula`. Unlike function definitions, predicate definitions do not require their return type to be explicitly specified in the declaration.

Rule invocations are represented by a syntax nearly identical to that of premises but are preceded by the `@` symbol. This symbol, which is neither an atom nor used elsewhere in the syntax, is required to disambiguate rule invocations from custom notations in the DSL.

To illustrate the need for such disambiguation, consider a version of the syntax without the `@` symbol, like `(Config_ok : |- config : t?)`. Note that `Config_ok` is a valid `varid` and `:` is a valid atom that can be used within expressions to form custom notations. This leads to a shift-reduce conflict between custom notations and rule expressions, because `(Config_ok : |- config : t?)` can be parsed both as a custom notation of the form `varid ":" exp`, or as a rule expression with the syntax `relid ":" exp`.

This is a primary example of the tradeoff between readability and expressibility of the SpecTec DSL. There are several solutions to this problem, but none are ideal and are thus omitted here for brevity. For the purposes of this project, we adopt the `@` symbol as a temporary solution.

```
def ::=                                       exp ::=
  ...                                           notop exp
  "formula" "$" defid params                    exp logop exp
  "formula" "$" defid args "=" formula          ...
  ...
  "theorem" thmid hint* "=" formula           formula ::=
  "lemma" thmid hint* "=" formula               notop formula
  "theorem" thmid hint+                         formula logop formula
  "lemma" thmid hint+                           exp
                                                "@" "(" relid ":" exp ")"
notop ::= "~"                                   "forall" args formula
logop ::= "/\" | "\/" | "=>" | "<=>"            "exists" args formula
```

Figure 3.31: Initial design of basic formulas

This initial design outlined in Figure 3.31 is satisfactory but poses several issues due to the syntactic distinction between `exp` and `formula`. Notably, the similarities between `formula` and `exp`, as well as between predicate and function definitions, imply that there are substantial duplications in handling these constructs. This not only introduces changes that are not necessarily minimal but also harms the maintainability of the SpecTec codebase.

Consequently, Figure 3.32 presents an alternative design that addresses these issues by drastically simplifying the syntax. The key insight is that the syntactic distinction between `formula` and `exp` is in fact unnecessary.

For instance, Isabelle/HOL treats all first-order logic formulas simply as boolean terms, which is illustrated in Figure 3.33. In Isabelle/HOL, quantifiers such as `All` and `Ex` are represented as higher-order boolean-valued functions taking a predicate, which is itself another boolean-valued function [Isa]. This reflects the model theoretic approach of logic, where every formula is assigned a single logical truth. After all, the syntactic distinction is only necessary in systems that interpret formulas differently, such as those based on dependent types.

In this alternative design, rule invocations are represented as `exp` with `bool` type, and similarly for expressions quantified by `forall` and `exists`. There is no longer ambiguity concerning the operators in `notop` and `logop`, nor is there a need to re-implement operator handling for `formula`. Predicates can be defined using the syntax for function definitions, as functions may return arbitrary `exp`. However, this does not imply that basic formulas can be used wherever `exp` is permitted. During validation it remains necessary to enforce that rule expressions and quantifiers do not appear in invalid places, such as in function arguments.

```
def ::=                                       exp ::=
  ...                                           ...
  "theorem" thmid ":" exp hint*                 "@" "(" relid ":" exp ")"
  "lemma" thmid ":" exp hint*                   forall args exp
  "theorem" thmid hint* "=" exp                 exists args exp
  "lemma" thmid hint* "=" exp
```

Figure 3.32: Alternative design of basic formulas

```
definition All :: "('a => bool) => bool"
where "All P == (P = (%x. True))"

definition Ex :: "('a => bool) => bool"
where "Ex P == !Q. (!x. P x --> Q) --> Q"
```

Figure 3.33: Definitions of `All` and `Ex` in Isabelle/HOL

Possible Designs for Iterated Formulas

Iterated formulas in this context correspond to iterated premises in the DSL, allowing us to express that a property holds for each element in a sequence.

Figure 3.34 shows the design of iterated formulas. The syntax of iterated formulas is identical to that of iterated premises, but they can be used within `exp` to form arbitrary boolean expressions.

The `@` symbol is again required for disambiguation. To illustrate the need for such disambiguation, consider an alternative syntax without the `@` symbol, like `(Type_ok: |- type : ft')*`. This can be interpreted in two ways — either as a sequence of boolean expressions of type `bool*`, or as a conjunction over these boolean expressions with type `bool`. In other words, it is not syntactically obvious whether a sequence of boolean expressions should be left as a sequence or folded into a single boolean value.

```
exp ::=
  ...
  "@" "(" exp ")" iter*
```

Figure 3.34: Initial design of iterated formulas

This serves as another example of the trade-off between readability and expressiveness in the SpecTec DSL. While it is possible to eliminate the `@` symbol by introducing appropriate semantic rules, these implicit rules are likely to introduce unexpected behaviour and directly conflict with the bidirectional typing that the SpecTec compiler applies to operator expressions. These solutions are therefore not ideal and omitted in this section.

Instead, a proper solution would be to introduce generic fold operators into the DSL. For instance, we could define a unary "big and" operator `//\\`, such that expressions like `//\\ (@(Type_ok:-` type : ft'))*| represent a fold of a sequence of boolean values by conjunction. However, we will not implement this mechanism in this project, as the handling of these operators requires a generic implementation in individual backends, which adds unnecessary complexity.

Possible Designs for Theorem Definitions

The existing syntax for theorem definitions, as presented in Figure 3.31, can be further simplified by defining theorems as zero-ary predicates.

This approach is illustrated in Figure 3.35. In contrast to theorem provers like Coq, this is possible in the SpecTec DSL because formulas are treated as boolean expressions. The `pr⌋oof` hint annotations such as `hint(proof "theorem")` and `hint(proof "lemma")` enable us to specify that these zero-ary predicates should be rendered as theorems, rather than as boolean constants.

This approach is ideal as it introduces minimal changes to the DSL syntax, but it can often lack readability, especially when multiple hints are involved. In this project, we will implement both syntaxes for theorem definitions, as supporting both does not lead to significant duplication.

```
theorem t_progress =
  forall (s, f, admininstr*, t?) ...

def $t__progress' : bool  hint(proof "theorem")
def $t__progress' =
  forall (s, f, admininstr*, t?) ...
```

Figure 3.35: Definition of theorems as zero-ary predicates

Final Design

The final design of the first-order logic constructs is illustrated in Figure 3.36. Examples of theorem and predicate definitions in the SpecTec DSL are provided in Figure 3.37.

Note that predicates such as `$not__lf__return()` use double underscores `__` because single underscores `_` are reserved for rendering subscripts in LaTeX and prose backends.

```
def ::=                                exp ::=
  ...                                    ...
  "theorem" thmid ":" exp hint*          "@" "(" relid ":" exp ")"
  "lemma" thmid ":" exp hint*            "@" "(" exp ")" iter*
  "theorem" thmid hint* "=" exp          forall args exp
  "lemma" thmid hint* "=" exp            exists args exp
```

Figure 3.36: Final design of first-order logic constructs

```
def $not__lf__return(admininstr*) : bool
def $not__lf__return(admininstr*) =
  forall (val*, l, admininstr'*)
  admininstr* =/= val* (RETURN) admininstr'*

theorem t_progress =
  forall (s, f, admininstr*, t?)
  @(Config_ok: |- s; f; admininstr* : t?) =>
  $terminal__form(admininstr*) \/
  exists (s', f', admininstr'*)
  @(Step: s; f; admininstr* ~> s'; f'; admininstr'*)
```

Figure 3.37: Examples of theorem definitions in DSL

3.3.2 Lexing and Parsing

The SpecTec toolchain makes use of the `ocamllex` [dReIeeA23] and `menhir` libraries [PRG24]. The `ocamllex` is the standard lexer used by the OCaml compiler, while `menhir` is a modern alternative to `ocamlyacc`, offering improved support for error messages and conflict resolution.

Figures 3.38 and 3.39 present the DSL extensions of the lexical tokens and the context-free grammar in BNF. The tokens for `forall` and `exists` include the left parenthesis of `args` because `id` is defined by a regular expression that matches any upper-case or lower-case letters, including the keywords `forall` and `exists`. This is a practical limitation rather than a theoretical one.

Operator precedence in `exp` is enforced by recursively splitting the non-terminal `exp`, ordered from the least binding to the most binding. `exp_rel` represents the least binding level, while `exp_prim` corresponds to the most binding level. Iterated formulas fall under `exp_post`, since its trailing `iter*` can be regarded as a post-fix operator. `forall` and `exists` belong to `exp_un` because quantifiers behave like unary operators.

Note that, since `menhir` uses right-most derivation as an LR(1) parser, `forall` and `exists` technically have the lowest precedence among all operators. For example, `exists (m) n = m * 2` would be correctly parsed as `(exists (m) (n = m * 2))` rather than `((exists (m) (n)) = m * 2)`.

```
atmark ::= "@"
forall_lparen ::= "forall ("
exists_lparen ::= "exists ("
theorem ::= "theorem"
lemma ::= "lemma"
```

Figure 3.38: Extension of DSL tokens for first-order logic constructs

```
def ::=
  ...
  "theorem" thmid ":" exp hint*
  "lemma" thmid ":" exp hint*
  "theorem" thmid hint* "=" exp
  "lemma" thmid hint* "=" exp
```

```
exp_prim ::= ...
  "@" "(" relid ":" exp ")"
exp_post ::= ...
  "@" "(" exp_atom ")" iter*
exp_atom ::= exp_post | ...
exp_seq ::= exp_atom | ...
exp_un ::= exp_seq | ...
  "forall (" arg*"," ")" exp
  "exists (" arg*"," ")" exp
exp_bin ::= exp_un | ...
exp_rel ::= exp_bin | ...
exp ::= exp_rel
```

Figure 3.39: Extension of DSL syntax for first-order logic constructs

Finally, the EL was extended accordingly. Its details are omitted since the EL is only an abstract syntax tree of the DSL,

3.3.3 Elaboration and Validation

The next phase of the translation is the elaboration from EL to IL, as well as the validation of the translated IL. Figure 3.40 illustrates the extension of the IL syntax.

The quantifiers and theorem definitions contain `bind*`. The `forall` and `exists` quantifiers only generate binds for the quantified variables appearing in `args`, while the `theorem` and `lemma` definitions capture any remaining free variables. This semantics is convenient for downstream translation, particularly for the theorem prover backend.

Rule expressions are almost identical to rule invocations in premises, with their custom notation divided into a `mixop` and a tuple of `exp`. Iterated formulas are also almost identical to iterated premises.

```
def ::=
  ...
  "theorem" thmid bind* exp
  "lemma" thmid bind* exp

iterexp ::=
  iter (id, typ)*
```

```
exp ::=
  ...
  id ":" mixop exp
  exp iterexp (; iterated formulas ;)
  forall bind* arg* exp
  exists bind* arg* exp
```

Figure 3.40: Extension of IL syntax for first-order logic constructs

The elaboration of quantifiers begins by introducing a local environment that collects type information for both free and bound variables within the current scope. It then proceeds to recursively elaborate its subterms, namely `args` and `exp`. Unlike other occurrences of `args`, the types of the quantified variables are not declared and are therefore unknown from the top down. Instead, these types must be inferred from the bottom up.

The validation of elaborated quantifiers also begins by introducing a local environment, ensuring that any quantified variables appearing outside the overall quantified expression are classified as unbound. It then proceeds to validate each of its subterms recursively, namely `bind*`, `arg*` and `exp`.

Additionally, it verifies that the inner `exp` has the expected type `bool`, and that the overall expression also conforms to this type. It then ensures that the first-order logic constructs do not appear in unintended contexts. Currently, these constructs are only allowed within the body of function definitions and theorem or lemma definitions, but this restriction may be relaxed in the future.

The validation of theorem definitions proceeds similarly to that of other top-level constructs, except that it additionally ensures theorem names are not referenced elsewhere, as these names refer to boolean expressions that are expected to be true and thus effectively synonymous with the boolean constant `true`. The remaining details of the elaboration and validation are omitted for brevity.

The following modules involved in elaboration, validation, and the middlend passes have been updated to support the first-order logic constructs. The specifics of these changes are likewise omitted.

- Collection of free and "determinate" variables
- Collection of dimensions of iterated variables
- Equality between AST nodes
- Iterator pattern utility
- Variable substitution utility

- Generation of side-conditions
- Totalisation of `def` function definitions
- Substitution of subsumptions by the corresponding function calls
- Substitution of "the" operators by the correponding side conditions
- Substitution of wildcards

3.3.4 Coq Translation

The next phase of the translation involves transforming the IL into the MIL. Figure 3.41 shows the extensions to the MIL. Notably, rule invocations and iterated formulas do not require additional constructs, as they are represented by the application of existing Coq terms.

A key addition to the MIL is the boolean operators such as `~~` and `&&`, which were previously omitted since all logical expressions were represented using their propositional counterparts such as `~` and `/\`. These boolean operators are now necessary because some predicates are better interpreted as decidable, returning `bool` rather than `Prop`.

```
def ::=                                 basic_term ::=
  ...                                     ...
  "Theorem" ident binder* term            "~~"
  "Lemma" ident binder* term              "&&"
                                          "||"
term ::=                                  "==>"
  ...                                     "==" (; equivalence ;)
  "forall" binder* term                   "==" (; equiality ;)
  "exists" binder* term                   "!="
```

Figure 3.41: Extension of MIL syntax for first-order logic constructs

The first-order logic constructs are treated as boolean expressions in the DSL, which significantly simplifies its semantics. In Coq, however, a clear distinction exists between decidable (`bool`) and potentially undecidable (`Prop`) statements, which must be handled appropriately. While it is possible to interpret every boolean expression in the DSL as `Prop` in Coq, this is undesirable because `bool` predicates in Coq are easier to work with in proofs, as they can be reduced by Coq's execution engine automatically.

Consider the examples listed in Figure 3.42. In this case, `$is__const()` and `$const__list()` should be translated as `bool`, whereas `$not__lf__br()` would need to be translated as `Prop`.

```
def $is__const(admininstr) : bool
def $is__const(CONST valtype val_) = true
def $is__const(admininstr) = false

def $const__list(admininstr*) : bool
def $const__list(eps) = true
def $const__list(admininstr admininstr'*) =
  $is__const(admininstr) /\ $const__list(admininstr'*)

def $not__lf__br(admininstr*) : bool
def $not__lf__br(admininstr*) =
  forall (val*, l, admininstr'*)
  admininstr* =/= val* (BR l) admininstr'*
```

Figure 3.42: Examples of predicate definitions in the DSL

To accommodate this semantics, the transformation of `exp` in the IL to `term` in the MIL begins by checking whether the body of function or theorem definitions contains first-order logic constructs. When such constructs are detected, the logical operators are prioritised to be interpreted as `Prop` rather than `bool`, and vice versa.

For instance, the operator `/\` in `$const__list()` would be translated to `/\` rather than `&&`, whereas the operator `=/=` in `$not__lf__br()` would be translated to `<>` rather than `!=`. The boolean equality and inequality operators can be applied to arbitrary data types, due to the `EqType` instances we auto-generated in the previous section.

The transformation of the `forall` and `exists` quantifiers is straightforward. The binders, which represent the quantified variables, are directly generated from `bind*` rather than from `arg*`. Rule expressions are simply transformed into applications of dependent terms to the predicate. Iterated formulas are transformed into `all` and `all2`, respectively.

The transformation of theorem definitions inserts a `forall` quantifier for the variables in the top-level `bind*`. This mechanism allows us to omit explicit quantification of variables, analogous to those introduced by `Variable` or `Hypothesis` in Coq. Finally, the

Coq backend also supports theorem definitions via `proof` hints, as demonstrated in Figure 3.35.

Figure 3.43 shows an example of the results of the Coq translation, generated from the DSL source in Figure 3.37. Since IL2Coq was not originally developed with SSReflect in mind, the generated code may contain a mixture of Coq and SSReflect syntaxes.

```
Definition fun_not_lf_br (v___0 : (list admininstr)) : Prop :=
  match (v___0) with
    | (v_admininstr) => forall (v_val : (list val)) (v_l : labelidx)
    ↪  (v_admininstr' : (list admininstr)), (v_admininstr <> (@app _
    ↪  (list__val__admininstr v_val) (@app _ [(admininstr__BR v_l)]
    ↪  v_admininstr')))
  end.

Theorem t_progress : forall (v_s : store) (v_f : frame) (v_admininstr : (list
↪  admininstr)) (v_t : (option valtype)), ((Config_ok (config__ (state__ v_s
↪  v_f) v_admininstr) v_t) -> ((fun_terminal_form v_admininstr) \/ exists
↪  (v_s' : store) (v_f' : frame) (v_admininstr' : (list admininstr)), (Step
↪  (config__ (state__ v_s v_f) v_admininstr) (config__ (state__ v_s' v_f')
↪  v_admininstr')))).
Proof. Admitted.
```

Figure 3.43: Results of Coq translation of first-order logic constructs

3.3.5 LaTeX Rendering

The next phase is to implement an alternative translation path targeting LaTeX. The LaTeX code is generated directly from the EL and does not undergo the elaboration or validation processes described previously.

Figure 3.44 shows the LaTeX code generated from the DSL in Figure 3.37, together with its rendered output. The splice backend is extended to support rendering theorems in LaTeX, which allows splice commands of the form `$${theorem: name}` to specify where the corresponding LaTeX code should be expanded in reStructuredText documents.

The LaTeX backend supports customising the display name of theorems via `desc` hints, as illustrated in Figure 3.46, which is rendered in a side-box next to the LaTeX formula. The LaTeX backend also supports theorem definitions via `proof` hints, as shown in Figure 3.35.

Since the statements of theorems and auxiliary lemmas tend to be lengthy, a custom line-break logic is implemented for rendering `exp`. In particular, operators and quantifiers may insert two types of markers for line breaks — `mustbreak` and `allowbreak`.

As the name suggests, `mustbreak` indicates a point where a line break must be inserted, whereas `allowbreak` is conditional, which may insert a line break if the character count in the current line exceeds a predefined threshold. Note, however, that the character count is calculated in the LaTeX source code and is therefore only an approximation.

```
\begin{array}{@{}l@{}l@{}}
  \mbox{(Progress)} & \forall s, f, {{\mathit{instr}}^\ast}, {t^?}.{ \vdash
  ↪  }\;s ; f ; {{\mathit{instr}}^\ast} : {t^?} \Rightarrow \\[0.8ex]
  &{\mathrm{terminal\_form}}({{\mathit{instr}}^\ast}) \lor \exists {s'},
  ↪  {f'}, {{{\mathit{instr}}'}^\ast}.s ; f ; {{\mathit{instr}}^\ast}
  ↪  \hookrightarrow {s'} ; {f'} ; {{{\mathit{instr}}'}^\ast}
\end{array}
```

$$\begin{array}{@{}l@{}l@{}} \mbox{(Progress)} & \forall s, f, {{\mathit{instr}}^\ast}, {t^?}.{ \vdash }\;s ; f ; {{\mathit{instr}}^\ast} : {t^?} \Rightarrow \\[0.8ex] &{\mathrm{terminal\_form}}({{\mathit{instr}}^\ast}) \lor \exists {s'}, {f'}, {{{\mathit{instr}}'}^\ast}.s ; f ; {{\mathit{instr}}^\ast} \hookrightarrow {s'} ; {f'} ; {{{\mathit{instr}}'}^\ast} \end{array}$$

Figure 3.44: Results of LaTeX translation of first-order logic constructs

3.3.6 Prose Rendering

The final phase involves implementing an additional translation path targeting prose descriptions.

Unlike LaTeX formulas, prose descriptions are generated from the IL for validation relations, such as `Config_ok`, and from the AL for execution relations, such as `Step_pure`. This distinction arises because validation descriptions require typing information embedded in the IL, whereas execution descriptions align more naturally with the imperative representation of the AL. In the case of theorem definitions, prose descriptions are generated from the IL.

The prose backend introduces a new intermediate representation, called `para`, as shown in Figure 3.45. This representation roughly corresponds to natural language and enables simplification of prose descriptions by manipulating structured data rather than raw text. `para` resembles `exp` in the IL but has the following notable differences:

- Connectives like `"and"` and `"if"` take multiple `para` elements rather than a single `para`. This allows us to simplify descriptions of the form "if X, if Y, then Z" to "if X and Y, then Z" recursively, thereby enhancing readability.
- Special constructs like `"is valid with"` and `"steps to"` are introduced for describing validation and execution relations. This enables these frequently occurring relations to be expressed in a more natural format.

The prose backend supports `prose` hints on predicate definitions, as illustrated in Figure 3.46. This allows predicates to be described in natural language by filling the placeholders `%` with their arguments in left-to-right order. For example, `$terminal__form(ad`↩ `mininstr*)` with `prose` hint `hint(prose "% is in terminal form")` would be rendered as "*admininstr∗* is in terminal form".

Additionally, the prose backend supports customising the display name of theorems via `desc` hints, which is rendered in bold text within parentheses. Similar to the Coq backend, it also supports theorem definitions via `proof` hints, as illustrated in Figure 3.35.

```
cmpop ::=
  "equal to"
  "not equal to"
  "less than"
  "greater than"
  "less than or equal to"
  "greater than or equal to"
```

```
para ::=
  exp
  exp cmpop exp
  "not" para
  "and" para*
  "or" para*
  "if" para* "then" para
  para "iff" para
  "for each" (exp, exp)* para
  "for all" exp* para
  "there exists" exp* para
  exp "is valid with" exp? exp? exp?
  exp "steps to" exp
  "relation" id exp "holds"
  "predicate" id exp "holds"
  string exp*
```

Figure 3.45: Definition of `para` in prose intermediate representations

```
def $terminal__form(admininstr*) : bool
  hint(prose "% is in terminal form")
def $terminal__form(admininstr*) = ...

theorem t_progress hint(desc "Progress") = ...
```

Figure 3.46: Examples of `desc` and `prose` hints in DSL

Figure 3.47 shows the prose description in reStructuredText generated from the DSL in Figure 3.37, along with its rendered result in LaTeX. The splice backend is extended to support rendering theorems in prose, which allows splice commands of the form `$${t⌋ heorem-prose: name}` to specify where the corresponding prose description should be expanded in reStructuredText documents.

```
**Theorem (Progress)**.
For all :math:`s`, :math:`f`, :math:`{{\mathit{instr}}^\ast}`, :math:`{t^?}`,
↪  if :math:`{{\mathit{instr}}^\ast}` is valid with type :math:`{t^?}` under
↪  the context :math:`f` and the store :math:`s`, then
↪  :math:`{{\mathit{instr}}^\ast}` is in terminal form or there exists
↪  :math:`{s'}`, :math:`{f'}`, :math:`{{{\mathit{instr}}'}^\ast}` such that
↪  :math:`((s,\, f),\, {{\mathit{instr}}^\ast})` steps to :math:`(({s'},\,
↪  {f'}),\, {{{\mathit{instr}}'}^\ast})`
```

Theorem (Progress) For all s, f, instr^*, $t^?$, if instr^* is valid with type $t^?$ under the context f and the store s, then instr^* is in terminal form or there exists s', f', instr'^* such that $((s,\, f),\, \mathit{instr}^*)$ steps to $((s',\, f'),\, \mathit{instr}'^*)$

Figure 3.47: Results of prose translation of first-order logic constructs

3.4 Template Mechanism

The final extension to the SpecTec toolchain introduced in this project is the template mechanism. This section details the design and implementation of template constructs, designed to eliminate repetitive definitions in the SpecTec DSL [YSL+24].

The primary motivation for this mechanism is to automate the definition of boilerplate lemmas. Consider, for example, individual cases or inductive steps within the preservation proof [Cup24], some of which are translated in the DSL in Figure 3.48. These auxiliary lemmas often contain repetitive fragments in their statements, as they arise as specialisations of the original statement prior to case analysis or induction.

Specifying these auxiliary lemmas in the DSL offers finer control over proof structure and ensures consistency across theorem provers. However, manually undertaking this process is both laborious and difficult to maintain. Furthermore, the repetitive definitions of auxiliary lemmas arguably conflict with the purpose of the SpecTec DSL as a unified "single source of truth".

```
lemma Step_pure__br_if_true_preserves =
  forall (s, C, c, l, ft)
  @(Admin_instrs_ok: s; C |- (CONST (INN I32) c) (BR_IF l) : ft) =>
  @((CONST (INN I32) c) (BR_IF l) ~> (BR l)) =>
  c =/= 0 =>
  @(Admin_instrs_ok: s; C |- (BR l) : ft)

lemma Step_pure__br_if_false_preserves =
  forall (s, C, c, l, ft)
  @(Admin_instrs_ok: s; C |- (CONST (INN I32) c) (BR_IF l) : ft) =>
  @((CONST (INN I32) c) (BR_IF l) ~> eps) =>
  c = 0 =>
  @(Admin_instrs_ok: s; C |- eps : ft)
```

Figure 3.48: Examples of boilerplate lemmas in preservation proof

The need to avoid repetition is not limited to auxiliary lemmas. Figure 3.49 presents the reduction rules for binary operators on vector values in the Wasm 2.0 specification [Wor22a]. These reduction rules share very similar structures, and the same applies to other operators. As future versions of Wasm add support for wider SIMD widths, such as 256 bits and 512 bits, maintaining these reduction rules may become increasingly difficult. A generic template mechanism would therefore allow us to eliminate such duplication within the DSL.

```
syntax vectype =
  | V128

rule Step_pure/binop-val:
  (CONST nt c_1) (CONST nt c_2) (BINOP nt binop)  ~>  (CONST nt c)
  -- if $binop(nt, binop, c_1, c_2) = c

rule Step_pure/vvbinop:
  (VCONST V128 c_1) (VCONST V128 c_2) (VVBINOP V128 vvbinop)  ~>  (VCONST V128 c)
  -- if c = $vvbinop(V128, vvbinop, c_1, c_2)
```

Figure 3.49: Examples of reduction rules of vector instructions [YSL+25]

This section thus introduces new meta-level template constructs in the DSL, designed to capture repetitive patterns through a generic mechanism handled natively by the SpecTec compiler. The syntax and semantics of both the DSL and IL will be extended accordingly. All of these changes will be realised by extending SpecTec's OCaml codebase [YSL+25].

Figure 3.50 provides an overview of the parts of the SpecTec toolchain that have been extended in this section of the project. The scope of the template mechanism is focused primarily on the frontend and middlend of the SpecTec toolchain.

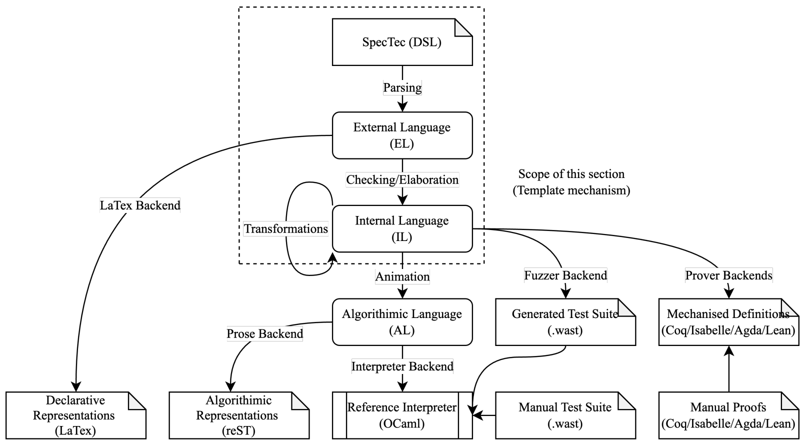
SpecTec (DSL)
Parsing
External Language (EL)
Checking/Elaboration
Scope of this section (Template mechanism)
LaTex Backend
Transformations
Internal Language (IL)
Animation
Fuzzer Backend
Prover Backends
Algorithimic Language (AL)
Prose Backend
Generated Test Suite (.wast)
Mechanised Definitions (Coq/Isabelle/Agda/Lean)
Interpreter Backend
Declarative Representations (LaTex)
Algorithimic Representations (reST)
Reference Interpreter (OCaml)
Manual Test Suite (.wast)
Manual Proofs (Coq/Isabelle/Agda/Lean)

Figure 3.50: Scope of template mechanism in SpecTec toolchain inspired by SpecTec paper [YSL+24]

3.4.1 Design Choices

This section discusses possible designs for the template mechanism, revisiting assumptions and exploring the designs from first principles.

DSL-level or MIL-level Definitions

Similar to the previous section, there is a fundamental question that must be addressed before considering any other designs — whether theorems should be maintained in the DSL or in the MIL.

The drawbacks of the MIL-level approach, as discussed in the previous section on the first-order logic constructs, also apply to the template mechanism in general.

A major benefit of the DSL-level approach, on the other hand, is that the template mechanism can be applied not only to theorem definitions but also to syntax, relation and function definitions. These definitions are interpretable by various backends and enables broader potential applications of the template mechanism beyond boilerplate lemmas.

Another argument in favour of the DSL-level approach is that restricting the mechanism to theorem definitions does not significantly simplify the implementation. There is thus little justification for imposing such limitations on the template mechanism.

We therefore conclude that the DSL is once again the appropriate format for this purpose.

DSL-level or IL-level Expansion

Another key design choice concerns whether template definitions should be expanded at the DSL level or at the IL level. Template expansion refers to the process of filling in the holes within template definitions, thereby instantiating them as non-template definitions.

The DSL-level expansion is primarily restricted to textual expansion, similar to macros in the C programming language. This is because the EL lacks details like typing information, which are essential for performing semantic analysis. This approach would therefore serve as an ad hoc solution, which can be implemented outside SpecTec's OCaml codebase by using an external template engine such as Mustache [HSW+24]. This would simplify the implementation significantly.

This textual expansion is also language-agnostic, allowing us to transform parts of the code beyond the semantics of the DSL. Consider the DSL in Figure 3.51, which illustrates a hypothetical template construct `{{ rule_name }}` embedded within an identifier. By exploiting this flexibility of textual expansion, we can also expand multiple expressions separated by commas, or substitute expressions in a way that overrides operator precedence, for instance.

```
template
lemma Step_pure__{{ rule_name }}__preserves = ...
```

Figure 3.51: Example of textual expansion in DSL

However, there is a practical limitation that the EL cannot be reconstructed from the IL. This is a problem because the IL contains information required for semantic analysis, which is essential for collecting template meta-variables. Substituting these template meta-variables back into the DSL would require reconstructing IL fragments into EL fragments, which is impractical since the elaboration process is not injective. There are several ways to approximate this reconstruction, but they end up complicating the implementation to the extent that the DSL-level approach is no longer ideal.

On the other hand, the IL-level expansion is not subject to such limitations. It would operate as a semantic expansion, enabling semantic checks both before and after expansion. This not only ensures that no ill-formed templates are instantiated but also makes error messages more accurate by detecting non-template issues first. The implementation will be tightly integrated into the semantics of the DSL, offering enhanced type safety, debuggability and maintainability. Furthermore, this approach enables us to model the template mechanism within a meta-theory of the DSL, which will be necessary to establish the correctness of the overall translation process.

We therefore conclude that the advantages of the IL-level approach outweigh those of the DSL-level approach, given the technical challenge associated with the latter.

Logic-ful or Logic-less Expansion

Another important design choice concerns whether to allow meta-level logical handling within the template mechanism or to restrict it to logic-less templates.

Figure 3.52 presents a hypothetical example of logic-less template definitions. In this design, template definitions consist of placeholders in the form `{{ ... }}`, which are substituted with the matching template meta-variables supplied by the SpecTec compiler. This results in a simple notation requiring minimal modifications to the DSL syntax.

A potential downside of this design is the tight coupling between meta-variables and template definitions. `rule_before` and `rule_after`, for example, can only be defined for reduction relations and are primarily restricted to use by auxiliary lemmas. However, this coupling is inevitable in logic-less templates to some extent, as every variation of the existing template meta-variables must be addressed by introducing new ones, rather than being handled within the DSL itself.

```
template
lemma Step_pure__preserves =
  forall (S, C, c, tf)
  @(S; C |- {{ rules_before }} : tf) =>
  @({{ rules_before }} ~> {{ rules_after }}) =>
  {{ rules_premises }} =>
  @(S; C |- {{ rules_after }} : tf)
```

Figure 3.52: Example of logic-less template definitions

While this tight coupling is undesirable, we believe it is a necessary compromise to preserve simplicity. We would otherwise require a meta-level system capable of analysing the AST within the DSL, which is far more expressive and powerful than necessary.

Figure 3.53 demonstrates this point with a hypothetical example of logic-ful template definitions, whose syntax is inspired by Python's Jinja2 [Pal24] and OCaml. This would amount to designing an entirely new meta-language integrated into the SpecTec DSL, which is impractical.

```
{% for rule in relations.Step_pure.rules %}
lemma Step_pure__preserves =
  forall (s, C, ft, {{ rule_freevars rule }})
  @(Admin_instrs_ok: s; C |- {% match rule.exp with (e1, "~>", e2) -> e1 %} : ft) =>
  @(Step_pure: {% match rule.exp with (e1, "~>", e2) -> e1 %} ~>
               {% match rule.exp with (e1, "~>", e2) -> e2 %}) =>
  {{ fold_left (fun ps p -> ps "=>" p) rule.premises }} =>
  @(Admin_instrs_ok: s; C |- {% match rule.exp with (e1, "~>", e2) -> e2 %} : ft)
{% endfor %}
```

Figure 3.53: Example of logic-ful template definitions

Possible Designs

Finally, the design of the templates can be discussed. Figure 3.54 presents the initial design of the template mechanism, accompanied by an example in Figure 3.55.

The syntax is largely inspired by that of Mustache templates [HSW+24], although its semantics differ in that a single template definition expands into multiple non-template definitions. The template meta-variables are organised within hierarchical scopes, starting with top-level scopes such as `relations`.

Currently, these template constructs can only be substituted by expressions in the DSL. This is a practical restriction intended to simplify the implementation, and we may allow substitution by types and other elements in the future.

```
def ::=                                   exp ::=
  ...                                       ...
  "template" def                            "{{" tmplid*"." "}}"
```

Figure 3.54: Initial design of template constructs

```
template
lemma Step_pure__preserves =
  forall (S, C, c, tf)
  @(S; C |- {{ relation.Step_pure.rules.before }} : tf) =>
  @({{ relation.Step_pure.rules.before }} ~> {{ relation.Step_pure.rules.after }}) =>
  {{ relation.Step_pure.rules.premises }} =>
  @(S; C |- {{ relation.Step_pure.rules.after }} : tf)
```

Figure 3.55: Examples of initial design of template constructs

The substitution of template constructs is not straightforward because the substituted expressions may contain free variables. It is not appropriate to implicitly quantify these free variables with a `forall` quantifier at the top level, since these variables may not be intended to be universally quantified at that level, or alternatively, may be intended to be existentially quantified instead.

To address this, we introduce an additional template meta-variable, `freevars`, within ⌋ `relations.Step_pure.rules`. This enables the quantification of these free variables at arbitrary positions, as demonstrated in Figure 3.57. The ellipsis `...` preceding the template meta-variable is a new syntax for variable-length template expressions, indicating that the sequence of free variables should be expanded into multiple `arg`'s separated by commas, rather than a single `arg`.

Furthermore, we introduce a wildcard syntax, `*`, which can be used within template meta-variables, as illustrated in Figure 3.57. This explicitly specifies the scopes under which the template meta-variables are iterated during expansion. For better organisation, this wildcard syntax and the variable-length template syntax are extracted into a new non-terminal called `slot`.

```
def ::=                                slot ::=
  ...                                    tmplid
  "template" def                         slot "." tmplid
                                         slot "." "*"
exp ::=                                  "..." slot
  ...
  "{{" slot "}}"
```

Figure 3.56: Alternative design of template constructs

```
template
lemma Step_pure__preserves =
  forall (s, C, ft, {{ ...relations.Step_pure.rules.*.freevars }})
  @(Admin_instrs_ok: s; C |- {{ relations.Step_pure.rules.*.before }} : ft)
  ↪  =>
  @(Step_pure: {{ relations.Step_pure.rules.*.before }} ~> {{
  ↪  relations.Step_pure.rules.*.after }}) =>
  {{ relations.Step_pure.rules.*.premises }} =>
  @(Admin_instrs_ok: s; C |- {{ relations.Step_pure.rules.*.after }} : ft)
```

Figure 3.57: Examples of alternative design of template constructs

The name of the definitions instantiated from templates is given by appending a suffix inferred from the position of the wildcard `*`. For example, the suffix `br_if_true` is appended for `relations.Step_pure.rules.br_if_true.premises` and `relations.Step_pure.r`⌋ `ules.*.premises`, resulting in `Step_pure__preserves__br_if_true`.

Final Design

The final design of the template mechanism is identical to that illustrated in Figures 3.56 and 3.57, and is therefore omitted in this section.

3.4.2 Lexing and Parsing

Figures 3.58 and 3.59 present the DSL extensions to the lexical tokens and the context-free grammar in BNF. The `def` non-terminal recursively references itself to represent the body of templated definitions. Template constructs are placed in `exp_prim`, as they are intended to be the most tightly binding entity within `exp`.

The `slot` non-terminal represents template meta-variables in hierarchical scopes and includes the ellipsis (`...`) and wildcards (`*`). The `slot` is further divided into `slot_dots` to syntactically enforce that the variable-length marker `...` only appears at the top level.

```
template ::= "template"
```

Figure 3.58: Extension of DSL tokens for template constructs

```
def ::=
  "template" def

exp_prim ::=
  ...
  "{{" slot "}}"
exp ::= ...
```

```
slot_dots ::=
  tmplid
  slot_dots "." tmplid
  slot_dots "." "*"
slot ::=
  slot_dots
  "..." slot_dots
```

Figure 3.59: Extension of DSL syntax for template constructs

Finally, the EL was extended accordingly. Its details are omitted as the EL is simply an abstract syntax tree of the DSL,

3.4.3 Elaboration and Validation

The next phase of the translation involves elaboration from EL to IL, as well as validation of the resulting IL. Figure 3.60 illustrates the extension of the IL syntax, which closely mirrors that of EL. This extension is necessary because template constructs are expanded through manipulation of the AST at the IL level.

A subtle issue in the elaboration process is that each `exp`, including template constructs, must be assigned a corresponding type. This poses an issue, as the type of template constructs such as `{{ relations.Step_pure.rules.*.premises }}` remains unknown until they are substituted with the corresponding `exp`. To address this, we introduce and assign a temporary `bottom` type before expansion. The `bottom` type is defined as a subtype of all other types, thereby preventing type mismatch errors during validation.

```
def ::=                                      slot ::=
  "template" def                               id
                                               slot "." id
typ ::=                                        slot "." "*"
 "bottom"                                      "..." slot

exp ::=
  ...
  "{{" slot "}}"
```

Figure 3.60: Extension of IL syntax for first-order logic constructs

The elaboration of template constructs is trivial, involving an almost one-to-one translation, with the exception of the assignment of the `bottom` type.

The validation logic is modified to operate in two distinct phases — before and after semantic substitution. This approach allows non-template errors to be detected prior to expansion, ensuring that the subsequent template middlend does not operate on an ill-formed IL.

The validation of template constructs is also straightforward. In addition, we enforce that top-level template meta-variables, such as `relations`, cannot appear alone in the form `{{ relations }}`.

Furthermore, we ensure that the name of a template definition, such as `Step_pure__preserves`, is not directly referenced by other definitions. This is necessary as any reference to a template definition prior to expansion would be invalid. We additionally verify that template definitions do not reference themselves recursively — although recursion is prohibited in theorem definitions, it may occur in syntax and function definitions.

3.4.4 Template Middlend

The template middlend is responsible for semantic expansion, processing the input IL to produce a template-free output IL. It is invoked after the other middlends have processed non-template definitions, thereby allowing auxiliary transformations to be performed beforehand.

Environment Pass

The template middlend begins with the environment pass, whose role is to analyse the non-template definitions in the input IL and collect the necessary information for template meta-variables.

Figure 3.61 presents the template meta-variables that are currently available. Asterisks (`*`) indicate that the meta-variables are accessible under any scope. Some meta-variables are restricted to specific scopes, such as `before` and `after`, which are only available within the rules `Step_pure` and `Step_read`.

Meta-variable	Type	Description
`definitions.*.name`	`text`	Name of the function definition
`syntaxes.*.name`	`text`	Name of the syntax type definition
`relations.*.name`	`text`	Name of the relation definition
`relations.*.rules.*.name`	`text`	Name of the rule definition
`relations.*.rules.*.freevars`	`tuple`	Free variables occuring in the rule
`relations.*.rules.*.premises`	`bool`	Premises of the rule
`relations.Step_pure.rules.*.before`	`admininstr*`	Redex of the reduction rule
`relations.Step_pure.rules.*.after`	`admininstr*`	Contractum of the reduction rule
`relation.Step_read.rules.*.before`	`admininstr*`	Redex of the reduction rule
`relations.Step_read.rules.*.after`	`admininstr*`	Contractum of the reduction rule

Figure 3.61: List of available template meta-variables

The collection of most template meta-variables is straightforward. `name` is extracted directly from the `id` of each definition in the IL, while `before` and `after` are found by pattern matching against the body of each rule within `Step_pure` and `Step_read`. `free⌋vars` is collected by identifying the free variables appearing in both the body and premises of each rule.

The collection of `premises`, however, is more involved. This process begins by transforming each premise of the rule into an equivalent boolean expression, according to the following rules:

- If the premise is `"if" exp`, it is converted into `exp`.
- If the premise is `"otherwise"`, it is replaced by the negation of the disjunction of the preceding premises.
- A rule invocation `relid ":" exp` is converted into a rule expression `"@" "(" relid ":" exp ")"`.
- An iterated premise `"(" premise ")" iter*` is converted into an iterated formula `"@" "(" exp ")" iter*`.

The handling of `otherwise` premises involves collecting the preceding premises, which, in the context of the `Step_pure` and `Step_read` relations, refer to the premises of the preceding rules that share the same redex. This is achieved by maintaining a mapping from each redex to its corresponding premises, which is then looked up when the current rule contains an `otherwise` premise.

The semantics of these `otherwise` premises are relevant only within the `Step_pure` and `Step_read` relations, as other relations do not incorporate the notion of redexes. In general, the interpretation of `otherwise` premises is delegated to individual backends, which justifies the special handling in this case.

Finally, after the conversion, these boolean expressions are combined into a single boolean expression by taking a conjunction. This is necessary because a template construct must be substituted by a single `exp` during expansion.

These collected template meta-variables are then organised into a tree-like data structure called `slottree`, whose definition is given in OCaml-based pseudocode in Figure 3.62.

The `slotentry` at each leaf contains the binds `bind*` of the free variables appearing in the collected expression `exp`. These binds are essential for substituting the `exp` into the holes of the template definitions, as explained later.

```
type slotentry = (list bind, exp)
type slottree =
  | LeafT (option slotentry)
  | NodeT (map string slottree)
```

Figure 3.62: Definition of `slottree` in OCaml-based pseudocode

The binds are collected from the corresponding IL definition. However, this process is complicated by the fact that binds may depend on other binds through the dependent arguments of their types. For example, consider that `c` is free in a collected expression, and the corresponding binds are given as `{C : context, t : valtype, c : val_(t)}`. In this case, we must collect both `c` and `t`, since the type of `c` depends on `t`.

The collection of binds is therefore implemented by recursively identifying such dependent binds until a fixpoint is reached, disregarding their order. This algorithm always terminates because the binds in the IL are finite.

Slot Collection Pass

The next step is the slot collection pass, which handles the collection of the slots corresponding to the holes in the template definitions. This is implemented by recursively traversing the input IL.

This preliminary pass is necessary because the scopes under which template expansion should proceed remain unknown until all slots have been collected. For instance, the template definition in Figure 3.57 expands by iterating over all rules under `relations.Step_pure.*`, but this cannot be determined until all six of its slots are collected.

Combination Generation

The next step is the combination generation, which is responsible for enumerating all possible combinations of the template meta-variables that can be substituted to produce each instance of a template definition.

For instance, in Figure 3.57, we generate a combination of `relations.Step_pure.rules.call_addr.before`, `relations.Step_pure.rules.call_addr.after`, `relations.Step_pure.rules.call_addr.premises` and `relations.Step_pure.rules.call_addr.freevars` for the rule `call_addr`, and similarly for `br_if_true`, `br_if_false`, and so forth.

The template mechanism is not limited to an $O(N)$ combination of template meta-variables. With the wildcard `*` syntax, we should be able to produce $O(N^2)$ nested combinations, such as in `relations.*.rules.*.name`, or even form an $O(N^2)$ Cartesian product of combinations, such as between `relations.Step_pure.rules.*.name` and `relations.Step_read.rules.*.name`.

These extended uses of the wildcard `*` are motivated by Figure 3.63, which compares the reduction rules of the `vcvtop` instruction for SIMD values in the Wasm 2.0 specification [Wor22a].The `full`, `half` and `zero` reduction rules share similar structures, which can become difficult to maintain as more SIMD widths are added. Instead, these reduction rules can be written using templates, organising the template meta-variables in a hierarchy like `relations.Step_pure.rules.vcvtop.before.full.v128` and accessing them using nested wildcards like `relations.Step_pure.rules.vcvtop.before.*.*`.

```
rule Step_pure/vcvtop-full:
  (VCONST V128 c_1) (VCVTOP (Lnn_2 X N_2) (Lnn_1 X N_1) vcvtop eps sx) ~>
  ↪  (VCONST V128 c)
  -- if c'* = $lanes_(Lnn_1 X N_1, c_1)
  -- if c = $invlanes_(Lnn_2 X N_2, $vcvtop(Lnn_1 X N_1, Lnn_2 X N_2, vcvtop,
  ↪  sx, c')*)

rule Step_pure/vcvtop-half:
  (VCONST V128 c_1) (VCVTOP (Lnn_2 X N_2) (Lnn_1 X N_1) vcvtop hf sx?) ~>
  ↪  (VCONST V128 c)
  -- if ci* = $lanes_(Lnn_1 X N_1, c_1)[$halfop(hf, 0, N_2) : N_2]
  -- if c = $invlanes_(Lnn_2 X N_2, $vcvtop(Lnn_1 X N_1, Lnn_2 X N_2, vcvtop,
  ↪  sx?, ci)*)

rule Step_pure/vcvtop-zero:
  (VCONST V128 c_1) (VCVTOP (nt_2 X N_2) (nt_1 X N_1) vcvtop eps sx? ZERO) ~>
  ↪  (VCONST V128 c)
  -- if ci* = $lanes_(nt_1 X N_1, c_1)
  -- if c = $invlanes_(nt_2 X N_2, $vcvtop(nt_1 X N_1, nt_2 X N_2, vcvtop,
  ↪  sx?, ci)* $zero(nt_2)^N_1)
```

Figure 3.63: Reduction rules for vector conversion operators in DSL [YSL+25]

Another motivation is illustrated in Figure 3.64. In the future, the `vsplat` and `vextract_lane-num` reduction rules could be defined between any $O(N^2)$ pairs of `CONST`, `VCONST V128`, `VCONST V256` and `VCONST V512`, which can become difficult to manage. Instead, these reduction rules can be written using templates once again, by organising the template meta-variables like `relations.Step_pure.rules.vsplat.before.v128` and `relations.Step_pure.rules.vsplat.after.v128` and taking a Cartesian product between `relations.Step_pure.rules.vsplat.before.*` and `relations.Step_pure.rules.vsplat.after.*`.

```
rule Step_pure/vsplat:
  (CONST $unpack(Lnn) c_1) (VSPLAT (Lnn X N)) ~> (VCONST V128 c)
  -- if c = $invlanes_(Lnn X N, $packnum(Lnn, c_1)^N)

rule Step_pure/vextract_lane-num:
  (VCONST V128 c_1) (VEXTRACT_LANE (nt X N) i) ~> (CONST nt c_2)
  -- if c_2 = $lanes_(nt X N, c_1)[i]
```

Figure 3.64: Reduction rules for `splat` and `extract` instructions in DSL [YSL+25]

We must therefore devise an algorithm capable of handling nested occurrences of the wildcard `*`, as well as Cartesian products of two slots under different parent scopes. Note that a Cartesian product should only be taken when the slots diverge, and not when they share the same parent scope, such as between `relations.Step_pure.rules.*.before` and `relations.Step_pure.rules.*.after`.

To facilitate this, we organise all the slots in a template definition by inserting them into a prefix tree (also known as a trie), whose definition is given in Figure 3.65. For instance, `relations.Step_pure.rules.call_addr.before` is split into `relations`, `Step_pure`, `rules`, `call_addr` and `before`, and then inserted into the leaf node corresponding to this path. The use of a prefix tree allows us to group slots by their parent scopes, ensuring that a Cartesian product is only taken when the slots diverge.

```
type slottrie =
  | LeafP (option slot)
  | NodeP (map string slottrie)
```

Figure 3.65: Definition of `slottrie` in OCaml-based pseudocode

The main procedure of this algorithm is presented in OCaml-based pseudocode in Figure 3.66. This algorithm is based on tree navigation by the path to a leaf node (similar to XPath in XML[Moznd]) except that the path consists of multiple slots organised in a prefix tree and may include wildcards `*`.

```
type subst = (slot, slotentry)
type substs = list subst
type comb = list substs

func sum acc comb = acc ++ comb

func product acc comb =
  if (acc == []) return comb
  return flatten (map (fun x -> map (fun y -> x ++ y) acc) comb)

func make_comb tree trie =
  match (tree, trie) with
  | (LeafT (Some e), LeafP (Some s)) -> [[(s, e)]]
  | (NodeT cs, NodeP ds) -> (fold (fun dk dv acc ->
      if dk == "*" then
        let comb = fold (fun ck cv acc' ->
          let comb' = make_comb cv dv
          return sum acc' comb') cs []
        return product acc comb
      else
        let cv = find dk cs
        let comb = make_comb cv dv
        return product acc comb) ds [])
  | (LeafT _, NodeP _) -> error "too long slot"
  | (NodeT _, LeafP _) -> error "too short slot"
```

Figure 3.66: Template expansion algorithm in OCaml-based pseudocode

This algorithm takes as input a data tree `slottree` of template meta-variables and a prefix tree `slottrie` of all the slots in a template definition. The navigation then proceeds recursively from the root nodes of both trees, as summarised below:

1. If the current node of the prefix tree has a single child, it visits the corresponding child in the data tree, performing a regular tree navigation by path.
2. If the current node of the prefix tree has multiple children, it visits all the children in the data tree in a DFS order and combines the resulting combinations by taking a Cartesian product.
3. If the current node of the prefix tree has a wildcard `*` as a child, it visits all the children in the data tree in DFS order and combines the resulting combinations by taking a sum.
4. Finally, if the current node of the prefix tree is a leaf node and the data tree is also a leaf node, it returns a single combination consisting of the matching `slot`. Otherwise, either the prefix tree or the data tree must be ill-formed.

Note that we take a product of combinations in case 2 because this corresponds to a situation where multiple slots diverge. Conversely, we take a sum of combinations when encountering a wildcard `*` in case 3, since the wildcard `*` signifies the position where enumeration of template meta-variables must occur, generating multiple combinations as a result.

This algorithm can be seen as a hybrid of path-based tree navigation and DFS-like tree traversal. If the prefix tree is linear and does not contain any wildcards `*`, such as `relations.Step_pure.rules.call_addr.before`, it behaves like a regular tree navigation following the path to a leaf node. If the prefix tree is linear but only consists of wildcards `*`, for instance `*.*.*.*.*`, it behaves as a typical DFS tree traversal, visiting all children.

Slot Substitution Pass

Finally, we proceed to substitute the template meta-variables into the holes of template definitions. This is implemented by recursively traversing a template definition and replacing template constructs with the corresponding `slotentry` available in the current combination. This process is performed for each combination, each producing an instance of the template definition.

Recall that during the environment pass, we collected not only the expressions to be substituted but also the binds of the free variables within those expressions. These binds are now propagated in a bottom-up manner and subsequently captured as the binds of the quantifiers and top-level definitions. The quantifiers capture only those binds relevant to their quantified variables, whereas the top-level definitions capture any remaining binds propagated bottom-up.

Repositioning Pass

As part of the slot substitution pass, we also invoke a sub-pass called the repositioning pass, which is responsible for overwriting the positional information embedded in the substituted IL expressions. This helps prevent misleading error messages pointing to the position of the substituted expressions rather than the template constructs in the DSL, if any errors are detected after template expansion.

Simplification Pass

There are several issues to be resolved, although the template expansion is technically complete at this point. This is best illustrated by the result of template expansion in Figure 3.67, where irrelevant parts have been replaced by ellipsis `...`.

The first issue concerns the subsumption `bottom <: bool` inserted at the position of the template constructs. This happens because the template constructs are temporarily assigned the `bottom` type to facilitate validation prior to template expansion. The elaboration process applies subsumption from the actual type `bottom` to the expected type `bool` wherever a cast is applied as part of the bidirectional typing — in this case, under the implication operator `=>`.

Another issue is that the substituted `premises` remains as a conjunction like `$type(z, x) = $funcinst(z)[a].TYPE_funcinst /\ $table(z, 0).REFS_tableinst[i] = ?(a) /\ ...`. Ideally, such conjunctions should be linearised into nested implications, such as `$type(z, x) = $funcinst(z)[a].TYPE_funcinst => $table(z, 0).REFS_tableinst[i] = ?(a) => ...`, to facilitate integration with theorem provers.

To address these issues, we introduce a post-processing pass responsible for performing various simplifications. It currently performs the following tasks, although the list is not exhaustive and may be extended in the future. Some of these simplifications must be performed prior to others, such as the removal of subsumption.

- Removal of subsumption from bottom type
- Linearisation of conjunction of premises in implications
- Removal of empty universal and existential quantifiers
- Removal of empty premises in implications

Figure 3.68 illustrates the result of this simplification pass. This demonstrates that the simplifications are applied correctly, producing cleaner code optimised for downstream translation.

```
lemma Step_read__preserves__call_indirect-call :
  forall (...) ... =>
  (@(Admin_instrs_ok: `%;%|-%:%`(s, C',
  ↪  [CONST_admininstr(INN_valtype(I32_inn), i)
  ↪  CALL_INDIRECT_admininstr(x)], ft)) => ... =>
  (((($type(z, x) = $funcinst(z)[a].TYPE_funcinst) /\ (($table(z,
  ↪  0).REFS_tableinst[i] = ?(a)) /\ ((a < |$funcinst(z)|) /\ (i <
  ↪  |$table(z, 0).REFS_tableinst|)))) : bottom <: bool) => ... =>
  @(Admin_instrs_ok: `%;%|-%:%`(s, C', [CALL_ADDR_admininstr(a)], ft))))
```

Figure 3.67: Results of Coq translation before simplification pass

```
lemma Step_read__preserves__call_indirect-call :
  forall (...) ... =>
  (@(Admin_instrs_ok: `%;%|-%:%`(s, C',
  ↪  [CONST_admininstr(INN_valtype(I32_inn), i)
  ↪  CALL_INDIRECT_admininstr(x)], ft)) => ... =>
  (($type(z, x) = $funcinst(z)[a].TYPE_funcinst) =>
  (($table(z, 0).REFS_tableinst[i] = ?(a)) =>
  ((a < |$funcinst(z)|) =>
  ((i < |$table(z, 0).REFS_tableinst|) => ... =>
  @(Admin_instrs_ok: `%;%|-%:%`(s, C', [CALL_ADDR_admininstr(a)], ft)))))))
```

Figure 3.68: Results of Coq translation after simplification pass

3.4.5 Coq Translation

Since template expansion is fully handled within the template middlend, the IL passed to downstream translation is free of template constructs. Consequently, the Coq backend functions as expected without requiring any modifications.

Figure 3.69 gives an example of the final translation to Coq. Note that the generated code may contain a mixture of Coq and SSReflect syntaxes.

```
Lemma Step_pure__preserves__br_if_true : forall (v_l : labelidx) (v_c : iN)
↪  (v_ft : functype) (v_C : context) (v_s : store), ((Admin_instrs_ok v_s
↪  v_C [(admininstr__CONST (valtype__INN (inn__I32 )) (v_c :
↪  val_));(admininstr__BR_IF v_l)] v_ft) -> ((Step_pure [(admininstr__CONST
↪  (valtype__INN (inn__I32 )) (v_c : val_));(admininstr__BR_IF v_l)]
↪  [(admininstr__BR v_l)]) -> (((v_c : val_) <> 0) -> (Admin_instrs_ok v_s
↪  v_C [(admininstr__BR v_l)] v_ft)))).
Proof. Admitted.
```

Figure 3.69: Results of Coq translation of template constructs

Chapter 4

Evaluation

4.1 Progress Proof

The key outcome of this section is the completion of the progress proof. We made the necessary modifications to the DSL, acknowledged the divergence of the DSL from the Wasm 1.0 specification [Wor19] and developed the progress proof in a SSReflect-native style [Inr18b].

Apart from the changes made in `Admin_instr_ok` and `Memory_instance_ok`, all modifications to the DSL concerned the parts authored by Andreas himself. These inconsistencies were not identified by the preservation proof nor by the implementations of the individual backends, highlighting the significance of completing the progress proof in this project.

Several lemmas were reformulated from scratch to accommodate changes in the DSL translation, particularly those resulting from the elimination of block and evaluation contexts. This demonstrates originality in the development of the proof structure, going beyond a direct porting process of WasmCert-Coq [WRPP+21].

The consistent use of SSReflect tactics in the progress proof also offers a major benefit in terms of readability and maintainability. Compared to standard Coq proofs. SSReflect enforces an explicit style in proof mode, requiring us to name every newly introduced variable and to specify the exact occurrence in rewrites, for example [Inr18b]. These requirements help make the proofs more robust to future changes.

This proof style is in contrast to that of WasmCert-Coq by Rao [BGP+25] and the preservation proof by Diego [Cup25]. WasmCert-Coq likely began with pure Coq proofs and gradually incorporated SSReflect constructs as the codebase evolved, resulting in a mixture of both styles [BGP+25]. This project was therefore a valuable opportunity not only to migrate the proofs but also to resolve these stylistic inconsistencies.

There are, however, several outstanding tasks that still need to be addressed. A notable issue is the introduction of the `val_wf` axiom – the proofs should be rewritten without this axiom once the monomorphisation feature is implemented. Another pending task is to update the preservation proof to accommodate the modifications made to the DSL. This should be addressed in the near future, ideally as part of a broader refactoring effort to update both IL2Coq and the preservation proof itself [Cup24] to a SSReflect-native style.

In terms of future work, the next milestone would be to upgrade the proofs of the soundness theorems from the Wasm 1.0 [Wor19] to Wasm 2.0 [Wor22a] specification. This task remains relatively high priority because other backends already support the Wasm 2.0 specification [YSL+25], and the adoption of SpecTec into the official WebAssembly repository is likely imminent. However, we did not attempt it within this project, as our primary focus was on extending the SpecTec DSL.

4.2 Decidable Equality Proofs

The key outcome of this section is the full automation of `EqType` instance generation. We have successfully extended Il2Coq's existing mechanisms [Cup24] to automatically generate `EqType` instances for non-recursive data types, and improved WasmCert-Coq's approach [BGP+25] for (mutually) recursive data types by using the `Fixpoint` technique, alongside a custom hint database to isolate proofs involving decidable equality of individual data types.

The primary benefits of these `EqType` instance generations are discussed in detail in the introduction of Chapter 3.2. Compared to WasmCert-Coq's original approach, our method considerably simplifies the decidable equality proofs, reducing their length from approximately 100 lines to 15 lines per definition [BGP+25].

An interesting extension to this section of the project would be to upgrade the MathComp version to 2.0 or later, which uses hierarchy builders (HB) [Mat25] instead of canonical structures for defining `EqType` instances.

Figure 4.1 demonstrates the use of HB for `EqType` instances, which is inspired by a recent change made by Rao in WasmCert-Coq [BGP+25]. As we can observe, HB offers high-level commands, such as the factory `hasDecEq.Build` in this case [Mat25], which help reduce boilerplate code compared to Canonical Structures.

```
From HB Require Import structures.
Definition func_eqb v1 v2 := is_left (func_eq_dec v1 v2).
Definition eqfuncP : Equality.axiom func_eqb :=
  eq_dec_Equality_axiom func_eq_dec.
HB.instance Definition func_eqMixin := hasDecEq.Build func eqfuncP.
```

Figure 4.1: `EqType` instance using hierarchy builder (HB) inspired by WasmCert-Coq [BGP+25]

Additionally, it is important to recognise that generating decidable equality proofs is not possible for every data type. For instance, equality between two real numbers is generally undecidable, and thus representing floating-point numbers in the specification as real numbers would prevent them from being valid instances of `EqType`. Instead, such floating-point values must either be modelled using specialised libraries such as the `Flocq` library [inr25b], or represented explicitly in the DSL as a syntax type.

4.3 First-Order Logic Extension

The key outcomes of this section include a discussion of the design of first-order logic constructs and the introduction of the necessary extensions to the DSL, EL, IL and MIL. We successfully integrated the frontend, middlend and backends to perform the required translation and validation, producing theorem statements in Coq, LaTeX and prose formats.

The primary benefits of the first-order logic extension, along with the justifications for the design choices, are presented in the introduction of Chapter 3.3. This section focuses on the remaining aspects of this extension, including potential issues, side effects and directions for future work.

Figure 4.2 presents the manually written theorem statements that correspond to the auto-translated Coq code shown in Figure 3.43. A visual comparison between the two confirms that they are indeed equivalent, thereby justifying the correctness of the translation.

As an additional step, we have successfully migrated the proofs of a few theorems to the auto-translated counterparts. This was achieved with minimal modifications, as the translated statements are nearly identical to the original statements. This not only provides an objective confirmation of correctness but also suggests that migrating the proofs of the remaining theorems should be a straightforward process.

```
Definition not_lf_br es :=
  forall vcs l es',
  es <> list__val__admininstr vcs ++ [:: admininstr__BR l] ++ es'.

Theorem t_progress: forall s f es ts,
  Config_ok (config__ (state__ s f) es) ts ->
  terminal_form es \/
  exists s' f' es', Step (config__ (state__ s f) es) (config__ (state__ s'
  ↪  f') es').
Proof. ... Qed.
```

Figure 4.2: Manually written statement of progress property in Coq

Additionally, Figure 4.3 presents the manually written statement corresponding to the auto-translated LaTeX formula and prose description shown in Figures 3.47 and 3.44. The manual counterpart of the LaTeX formula is unavailable, as theorems are specified only in prose format within the specification. Once again, a visual comparison confirms that the manually written and auto-translated versions are logically equivalent, although the manual statement has a more natural style.

```
**Theorem (Progress).**
If a :ref:`configuration <syntax-config>` :math:`S;T` is :ref:`valid
↪  <valid-config>` (i.e., :math:`\vdashconfig S;T : [t^\ast]` for some
↪  :ref:`result type <syntax-resulttype>` :math:`[t^\ast]`), then either it
↪  is terminal, or it can step to some configuration :math:`S';T'` (i.e.,
↪  :math:`S;T \stepto S';T'`).
```

Theorem (Progress). If a configuration $S;T$ is valid (i.e., $\vdash S;T : [t^*]$ for some result type $[t^*]$), then either it is terminal, or it can step to some configuration $S';T'$ (i.e., $S;T \hookrightarrow S';T'$).

Figure 4.3: Manually written statement of progress property in LaTeX and prose [Wor19]

These first-order logic constructs form a superset of the premises in the DSL in terms of expressiveness. In particular, rule expressions replace rule invocations in premises, while iterated formulas substitute iterated premises. As a result, the syntax of premises can be simplified to include only `var`, `if` and `otherwise`, as illustrated in Figure 4.4.

```
premise ::=
  "var" id ":" typ              local variable declaration
  "if" exp                      side condition
  "otherwise"                   fallback side condition

rule Step/ctxt-label:
  z; (LABEL_ n `{instr*} admininstr*)  ~>  z'; (LABEL_ n `{instr*} admininstr'*)
  -- if @(Step: z; admininstr* ~> z'; admininstr'*)
```

Figure 4.4: Potential simplification of premise syntax in DSL

In fact, these first-order logic constructs are more expressive than premises due to the inclusion of universal and existential quantifiers. For example, consider range types in the DSL, which are internally represented as type aliases of `nat` or `int`, with their bounds specified by inequalities in the premises [Spe25b]. By leveraging first-order quantifiers, it becomes possible to define more complex data types, such as the type of even integers or prime numbers, as shown in Figure 4.5.

The interpretation of these premises is not handled by the frontend or middlend, but is delegated to the individual backends [Spe25b]. Consequently, the introduction of such complex premises does not affect the semantics of the DSL itself, although it may increase the complexity of backend implementations.

```
syntax int32 = int
  -- if int >= $(-2^31) /\ int < $(+2^31)

syntax even = int
  -- if exists (n) int = $(n * 2)

syntax prime = nat
  -- if nat >= 1
  -- if forall (y, z) nat = $(y * z) => y = 1 \/ z = 1
```

Figure 4.5: Potential applications of first-order logic constructs in premises in DSL

Another notable consequence of lifting premises to expressions is that the `otherwise` premise can now be represented simply by negating the conjunction of all preceding premises at the IL level. Previously, we had to define new relations for each negation, since the DSL lacks syntax to directly negate premises. This also implies that the else pass in IL2Coq [Cup24] is no longer necessary, thereby simplifying its implementation.

Moreover, these first-order logic constructs could be useful for explicitly annotating the quantification of determinate variables in the DSL. Determinate variables are a subset of free variables used by the elaboration process to generate binds, defined as follows [YSL+25]:

- Free variables occuring as an iteration variable
- Free variables occuring in destructuring position on the LHS
- Free variables occuring in destructuring position on either side of an equational premise
- Free variables occuring in destructuring position as an indexing operand
- Free variables occuring in destructuring position as the last call arg

Determinate variables are typically universally quantified, such as `C`, `deftype_1` and `deftype_2` in the rule `Deftype_sub/super` from the Wasm 3.0 draft specification [Wor22b], as given in Figure 4.6. However, determinate variables may also be existentially quantified in some cases, such as `fin`, `typeuse`, `ct` and `i` in Figure 4.6. With the first-order logic constructs, we can clarify these implicit semantics by quantifying these variables explicitly, as demonstrated in Figure 4.7.

```
rule Deftype_sub/super:
  C |- deftype_1 <: deftype_2
  -- if $unrolldt(deftype_1) = SUB fin typeuse* ct
  -- Heaptype_sub: C |- typeuse*[i] <: deftype_2
```

Figure 4.6: Example of determinate variables without explicit quantification [YSL+25]

```
rule Deftype_sub/super:
  C |- deftype_1 <: deftype_2
  -- if exists (fin, typeuse*, ct) $unrolldt(deftype_1) = SUB fin typeuse* ct
  -- if exists (i) @(Heaptype_sub: C |- typeuse*[i] <: deftype_2)
```

Figure 4.7: Example of determinate variables with explicit quantification

However, a notable limitation of the current implementation is that we can only quantify variables of non-family types. This prevents us from quantifying over variables of type `val_(t)`, for instance, although such quantification was not required for our use cases. Ideally, it would be desirable to introduce a type-annotation syntax such as `forall (t, c : val_(t)) ...`, but this is complicated by the need for disambiguation again, since the colon `:` is an atom that can form part of any custom notations.

Another practical challenge is that any change in the DSL requires recompilation. This poses a significant issue for iterative proof development, because it overwrites any proofs manually added to the placeholder `Proof. Admitted.` within the auto-translated theorem definitions.

It is possible to circumvent this limitation by outputting theorem statements to a temporary file, allowing them to be copied into a separate file containing the manual proofs. However, this process is rather cumbersome, particularly during the initial stages of proof development. Alternatively, we could define theorem statements as zero-ary predicates in the DSL, and state the theorems like `Theorem t_progress : t_progress_statement.` in Coq. While this approach eliminates the need for copy-pasting, it lacks readability as the actual statements must be referenced in a separate file.

In general, there is no straightforward solution that can address all of these issues. As a consequence, we may need to resort to one of these suboptimal solutions when incorporating this mechanism for proof development in the future.

Another drawback of the first-order logic constructs is the frequent use of the `@` symbol for disambiguation. However, we emphasise that the purpose of the `@` symbol is primarily to serve as a clear visual reminder that the grammar of the DSL will likely require revision to address these issues, should this mechanism be adopted in the future.

The revised syntax will likely require reserving certain symbols, such as the colon `:`, rather than allowing them to be part of custom notations. These decisions are deferred in this project, as they are inherently subjective and will depend heavily on feedback from specification writers.

Regarding code quality, considerable effort was made to ensure that the new code is clear, concise and consistent with the style of the existing codebase. Slight variations in OCaml coding styles exist across the SpecTec toolchain, particularly among the frontend, middlend, interpreter backend and Coq backend [YSL+25]. To minimise disruption, the new code was adapted to align with the style of the surrounding code within each respective component. Addressing these inconsistencies may necessitate a comprehensive refactoring of the SpecTec toolchain in the future.

Finally, an important direction for future work is to extend these first-order logic constructs to other theorem provers such as Lean and Isabelle/HOL, should they be adopted. These constructs will become increasingly significant once this extension is made, as one of their primary motivations is to maintain uniformity and consistency across theorem provers.

4.4 Template Mechanism

The key outcomes of this section include a discussion of the design of the template constructs and the introduction of the necessary extensions to the DSL, EL and IL. We successfully integrated the corresponding frontend and middlend to perform the translation, validation and template expansion, producing the statements of dozens of auxiliary lemmas from a single template definition.

The primary advantages of this template mechanism, as well as the justifications for the design decisions, are discussed in the introduction of Chapter 3.4. This section will address the remaining aspects of the template mechanism.

Figure 4.8 presents the manually written statement of the auxiliary lemma corresponding to the auto-translated Coq code shown in Figure 3.43. A visual comparison between the two confirms that they are indeed equivalent, except that the manually written statement omits the premise `((v_c : val_) <> 0)`, as preservation still holds without this condition. Furthermore, we have successfully migrated proofs of a few theorems to their auto-translated counterparts, confirming the correctness of the translation.

```
Lemma Step_pure__br_if_true_preserves :
  forall v_S v_C (v_c : iN) (v_l : labelidx) v_func_type,
  Admin_instrs_ok v_S v_C [(admininstr__CONST (valtype__INN (inn__I32 )) (v_c
  ↪   : val_));(admininstr__BR_IF v_l)] v_func_type ->
^^IStep_pure [(admininstr__CONST (valtype__INN (inn__I32 )) (v_c :
↪  val_));(admininstr__BR_IF v_l)] [(admininstr__BR v_l)] ->
^^IAdmin_instrs_ok v_S v_C [(admininstr__BR v_l)] v_func_type.
Proof. ... Qed.
```

Figure 4.8: Manually written auxiliary lemma statements in preservation proof [Cup24]

A potential drawback of this template mechanism is code bloat, characterised by a significant increase in the size of the IL code following template expansion. This poses a problem because the SpecTec compiler is not optimised for performance. For instance, the elaboration and validation processes take approximately 0.25 seconds for Wasm 1.0, 1.5 seconds for Wasm 2.0 and 2.75 seconds for Wasm 3.0 specifications on a consumer laptop, which is not particularly fast.

Simple template definitions producing an $O(N)$ number of instances are unlikely to cause issues. However, template definitions involving nested slots or Cartesian products of slots, resulting in $O(N^2)$ instances or more, could significantly slow down compilation. This will not be a problem for auxiliary lemmas in most cases, however, as proofs involving $O(N^2)$ cases are likely to be poorly structured and would require reformulation.

Another subtle issue concerns potential conflicts among free variables introduced in the substituted expressions, although this has not been an issue in our current use cases. Such conflicts may arise either with the surrounding context within the template definition or between expressions substituted into different holes of the template. The latter is particularly likely in cases involving a Cartesian product of slots.

As a temporary solution, we implemented a mechanism that renames these free variables to unique identifiers prior to substitution. While this approach partially addresses the problem, the issue of determining the appropriate scopes beyond which conflicts must be resolved remains unresolved – for instance, conflicts between `relations.Step_pure.rules.*.before` and `relations.Step_read.rules.*.before` should be resolved, but not between `relations.Step_pure.rules.*.before` and `relations.Step_pure.rules.*.after`, since the latter should share the same free variables. Consequently, this implementation has been abandoned and remains a task for future work.

Additionally, a similar concern arises regarding limitations on iterative proof development, which requires suboptimal workarounds as previously discussed. This is particularly relevant in the context of the template mechanism, as the auto-generated auxiliary lemmas often reflect the internal structure of proofs, which are subject to frequent updates.

Furthermore, it is worth noting that, unlike key theorems such as progress and preservation, these auxiliary lemmas are intended to be specified by verification engineers rather than specification writers. This imposes a steep learning curve on verification engineers, as they are required to learn the SpecTec DSL for proof development.

Regarding code quality, significant attention was paid to ensuring consistency with the existing codebase, as discussed in the previous section. The template middlend is relatively isolated from the rest of the code, which offered us greater flexibility in terms of coding style. We chose to adopt the style of the frontend code, primarily authored by Andreas, as this part of the SpecTec toolchain generally has the cleanest and most consistent coding style [YSL+25].

Futhermore, future work could explore restructuring the progress proof by extracting each inductive step into a standalone lemma, adopting a structure similar to that of the preservation proof. We did not take this approach for the progress proof, as manually writing such auxiliary lemmas would be laborious. However, this manual effort is no longer a concern with the introduction of the template mechanism.

Finally, an ambitious direction for future work could be to design a sophisticated mechanism capable of auto-generating not only the statements but also the proofs of auxiliary lemmas. Unlike automated theorem provers, this mechanism could use proof templates to guide proof search, drawing inspiration from logical frameworks [Gar92]. This work would be highly ambitious but can be seen as the ultimate goal for the theorem prover backend, as proof automation will inevitably become necessary at some point due to the rapid evolution of Wasm.

Chapter 5

Conclusion

In this project, we successfully completed the mechanised proof of the progress property in the Wasm 1.0 specification [Wor19], building upon Diego's prior work on IL2Coq and the preservation proof [Cup24]. We went beyond simply porting the proofs from WasmCert-Coq [WRPP+21] and introduced improvements in proof style and structure to enhance maintainability. This process led to the identification of key inconsistencies in the SpecTec DSL authored by Andreas [YSL+24], all of which were subsequently resolved.

The development of the progress proof highlighted the limitations of IL2Coq and the broader SpecTec toolchain. This motivated us to implement improvements in SpecTec itself, including the auto-generation of decidable equality proofs, the extension with the first-order logic constructs and the introduction of the template mechanism — which we collectively refer to as "Lemmagen", the title of this project.

Some of these improvements, namely the first-order logic constructs and the template mechanism, serve as proofs of concept, with their adoption left to the WebAssembly community. These extensions will nevertheless remain a significant contribution, as the difficulties and limitations that motivated their development persist and must be addressed in the future. Therefore, the focus of these extensions is not only on their implementation but also on documenting the design choices for future reference.

In conclusion, the contributions made in this project represent another milestone in the ongoing evolution of the SpecTec toolchain and the WebAssembly language, building upon WasmCert, IL2Coq and related efforts. We hope this work will offer meaningful value to future contributors who continue the development of the WebAssembly language.

Bibliography

[BDP24] Jasmin Blanchette, Martin Desharnais, and Lawrence C. Paulson. A user's guide to sledgehammer for isabelle/hol, 2024. Last accessed: 20 January 2025.

[BGP$^+$25] M. Bodin, P. Gardner, J. Pichon, C. Watt, and X. Rao. Wasmcert-coq. GitHub repository, 2025. Last accessed: 20 January 2025.

[Buz] David J. P. L. Buzzard. An introduction to lean. Last accessed: 20 January 2025.

[Con24] P4 Language Consortium. P4_16 language specification (working draft). https://p4.org/p4-spec/docs/p4-16-working-draft.html, 2024. Accessed: 2025-06-07; descriptive title and author inferred.

[cpp] cppreference.com. virtual function specifier. Last accessed: 20 January 2025.

[cpp25] cppreference.com. Undefined behavior, 2025. Last accessed: 20 January 2025.

[Cup24] Diego Cupello. Il2coq: Automatic translation of inductive logic programming concepts into coq. 2024. Accessed: 2025-01-13.

[Cup25] Diego Cupello. Il2coq. GitHub repository, 2025. Last accessed: 20 January 2025.

[dReIeeA23] Institut National de Recherche en Informatique et en Automatique. *Lexer and parser generators (ocamllex, ocamlyacc)*, 2023. Version 5.3.

[Fog18] Agner Fog. Nan payload propagation - unresolved issues, April 2018. Accessed: 2025-06-07.

[Gar92] Philippa Gardner. *Representing Logics in Type Theory*. PhD thesis, University of Edinburgh, 1992. Doctor of Philosophy.

[Gon08] Georges Gonthier. Formal proofthe four-color theorem. *American Mathematical Society Notices*, 2008. Last accessed: 20 January 2025.

[GR09] Georges Gonthier and Stéphane Le Roux. An ssreflect tutorial, 2009.

[HRS$^+$17] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. *SIGPLAN Not.*, 52(6):185200, June 2017.

[HSW$^+$24] Justin Hileman, Bruno Sutic, Chris Wanstrath, Ricardo Mendes, and Bruno Michel. mustache - logic-less templates., 2024. Accessed: 2025-06-08.

[IEE19] IEEE. Ieee standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019.

[Inr18a] Inria. The gallina specification language coq 8.9.1 documentation, 2018. Last accessed: 12 June 2025.

[Inr18b] Inria. The ssreflect proof language coq 8.19 documentation, 2018. Last accessed: 12 June 2025.

[Inr18c] Inria. The tactic language coq 8.9.1 documentation, 2018. Last accessed: 12 June 2025.

[Inr18d] Inria. Vernacular commands coq 8.9.1 documentation, 2018. Last accessed: 12 June 2025.

[Inr21a] Inria. Canonical structures coq 8.19 documentation, 2021. Last accessed: 12 June 2025.

[Inr21b] Inria. Tactics coq 8.19 documentation, 2021. Last accessed: 12 June 2025.

[Inr25a] Inria. Coq documentation, 2025. Last accessed: 12 June 2025.

[inr25b] inria. Flocq. https://flocq.gitlabpages.inria.fr, 2025. Accessed: 2025-06-09.

[Inr25c] Inria. Mathematical components, 2025. Last accessed: 20 January 2025.

[Isa] Isabelle Community. Isabelle documentation. Last accessed: 20 January 2025.

[ISO11] ISO/IEC. Programming languages - c, iso/iec 9899:2011 (c11), 2011. Last accessed: 20 January 2025.

[Lea] Lean Community. Lean documentation. Last accessed: 20 January 2025.

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107115, July 2009.

[LR24] Jae Hyun Lee and Sukyoung Ryu. P4-spectec: Mechanized language definition for p4, October 2024. Accessed: 2025-06-07.

[Mat25] MathComp. Hierarchy builder. https://github.com/math-comp/hierarchy-builder, 2025. Accessed: 2025-06-09.

[Moznd] Mozilla. Xpath. https://developer.mozilla.org/en-US/docs/Web/XML/XPath, n.d. Accessed: 2025-06-09.

[nLaa] nLab contributors. Dependent type theory. Last accessed: 20 January 2025.

[nLab] nLab contributors. Higher-order logic. Last accessed: 20 January 2025.

[Ora23] Oracle Corporation. The java language specification, java se 23 edition, 2023. Last accessed: 20 January 2025.

[Pal24] Pallets. *Jinja Documentation*, 2024. Version 3.1 (stable).

[PdAC$^+$25] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Ctlin Hricu, Vilhelm Sjöberg, and Brent Yorgey. *Logical Foundations*, volume 1 of *Software Foundations*. Electronic textbook, 2025.

[PRG24] François Pottier and Yann Régis-Gianas. *Menhir Reference Manual*, 2024. Accessed: 2025-06-08.

[Ros25] Andreas Rossberg. Spectec has been adopted, March 2025. Accessed: 2025-06-07.

[Sou14] Jean Souyiris. Industrial use of compcert on a safety-critical software product, 2014. Last accessed: 20 January 2025.

[Spe25a] SpecTec Community. Il. https://github.com/Wasm-DSL/spectec/blob/main/spectec/doc/IL.md, 2025. Accessed: 2025-06-07.

[Spe25b] SpecTec Community. Language. https://github.com/Wasm-DSL/spectec/blob/main/spectec/doc/Language.md, 2025. Accessed: 2025-06-07.

[Wat18] Conrad Watt. Mechanising and verifying the webassembly specification. pages 53–65, 2018.

[Wor19] World Wide Web Consortium (W3C). Webassembly specification release 1.0, 2019. Last accessed: 20 January 2025.

[Wor22a] World Wide Web Consortium (W3C). Webassembly specification release 2.0, 2022. Last accessed: 20 January 2025.

[Wor22b] World Wide Web Consortium (W3C). Webassembly specification release 3.0 (draft 2025-05-15), 2022. Last accessed: 20 January 2025.

[WRPP$^+$21] Conrad Watt, Xiaojia Rao, Jean Pichon-Pharabod, Martin Bodin, and Philippa Gardner. Two mechanisations of webassembly 1.0. 2021.

[YSL$^+$24] Dongjun Youn, Wonho Shin, Jaehyun Lee, Sukyoung Ryu, Joachim Breitner, Philippa Gardner, Sam Lindley, Matija Pretnar, Xiaojia Rao, Conrad Watt, and Andreas Rossberg. Bringing the webassembly standard up to speed with spectec. *Proc. ACM Program. Lang.*, 8(PLDI), June 2024.

[YSL$^+$25] Dongjun Youn, Wonho Shin, Jaehyun Lee, Sukyoung Ryu, Joachim Breitner, Philippa Gardner, Sam Lindley, Matija Pretnar, Xiaojia Rao, Conrad Watt, and Andreas Rossberg. Spectec. GitHub repository, 2025. Last accessed: 20 January 2025.

Errata

The following notable corrections have been made to this report since its submission, in chronological order:

Location	**Error**	**Correction**
p.32-33	Inappropriate use of `repeat` tactical in Figures 3.27 and 3.28	Replaced with `do ?` tactical to match SSReflect-native style
p.30, par 5, line 2	Inaccurate description of the proof in Figure 3.25	Clarified that the proof proceeds by establishing the `Config_sound` co-recursively
p.65, par 3, line 2	Unclear discussion of the limitation imposed on iterative proof development	Clarified that this limitation is particularly relevant due to frequent updates
p.21, par.3	Missing description of the modifications to the `Step/ctxt-frame` rule	Clarified that this rule has been modified to allow changes in the frame state `f`

Declarations

Ethical Considerations

There are no specific ethical considerations associated with this project.

However, it is important to acknowledge that the design choices made in the progress proof and the extensions to the SpecTec toolchain will significantly influence future developments of SpecTec, should they be adopted.

In this regard, we believe that we have provided thorough discussions justifying the design choices made in each section. Nevertheless, we must take responsibility for any outstanding tasks before the end of this project and ensure that any future directions are clearly communicated to the SpecTec contributors and the wider WebAssembly community.

Use of Generative AI

We acknowledge the use of ChatGPT-4o (OpenAI) to detect grammatical errors as part of the final review. We confirm that no AI-generated content has been presented as original work.

Sustainability

There are no specific sustainability issues associated with this project.

Availability of Data and Materials

The source code for this project is publicly available in the `coq-il-generation` branch as a fork of the Wasm SpecTec repository.

Appendix A

WebAssembly

The following presents a subset of the structure, typing rules and reduction rules of the WebAssembly language, as specified in the Wasm 2.0 specification [Wor22a], which are referenced in this report:

A.1 Structure

$$\begin{aligned} uN &::= 0 \mid 1 \mid \cdots \mid 2^N - 1 && \text{(value)} \\ sN &::= -2^N - 1 \mid \cdots \mid -1 \mid 0 \mid 1 \mid \cdots \mid 2^{N-1} - 1 \\ iN &::= uN \\ fN &::= \cdots \end{aligned}$$

$$\begin{aligned} \mathit{numtype} &::= \mathsf{i32} \mid \mathsf{i64} \mid \mathsf{f32} \mid \mathsf{f64} && \text{(type)} \\ \mathit{vectype} &::= \mathsf{v128} \\ \mathit{reftype} &::= \mathsf{funcref} \mid \mathsf{externref} \\ \mathit{valtype} &::= \mathit{numtype} \mid \mathit{vectype} \mid \mathit{reftype} \\ \mathit{resulttype} &::= [\mathit{vec}(\mathit{valtype})] \\ \mathit{functype} &::= \mathit{resulttype} \to \mathit{resulttype} \\ \mathit{blocktype} &::= \mathit{typeidx} \mid \mathit{valtype} \end{aligned}$$

$$\begin{aligned} \mathit{limits} &::= \{\mathsf{min}\ \mathit{u32}, \mathsf{max}\ \mathit{u32}^?\} \\ \mathit{memtype} &::= \mathit{limits} \\ \mathit{tabletype} &::= \mathit{limits}\ \mathit{reftype} \\ \mathit{globaltype} &::= \mathit{mut}\ \mathit{valtype} \\ \mathit{mut} &::= \mathsf{const} \mid \mathsf{var} \\ \mathit{externtype} &::= \mathsf{func}\ \mathit{functype} \\ &\mid \mathsf{table}\ \mathit{tabletype} \\ &\mid \mathsf{mem}\ \mathit{mathtype} \\ &\mid \mathsf{global}\ \mathit{globaltype} \end{aligned}$$

$$\mathit{nn}, \mathit{mm} ::= \mathsf{32} \mid \mathsf{64} \qquad \text{(instr)}$$

$$
\begin{aligned}
\mathit{sx} &::= \mathsf{u} \mid \mathsf{s} \\
\mathit{iunop} &::= \mathsf{clz} \mid \mathsf{ctz} \mid \cdots \\
\mathit{ibinop} &::= \mathsf{add} \mid \mathsf{sub} \mid \mathsf{mul} \mid \\
\\
\mathit{instr} &::= \mathsf{i}\mathit{nn}.\mathsf{const}\ \mathit{unn} \mid \mathsf{f}\mathit{nn}.\mathsf{const}\ \mathit{fnn} \\
&\quad \mid \mathsf{i}\mathit{nn}.\mathit{iunop} \mid \mathsf{f}\mathit{nn}.\mathit{funop} \\
&\quad \mid \mathsf{i}\mathit{nn}.\mathit{ibinop} \mid \mathsf{i}\mathit{nn}.\mathit{ibinop} \\
&\quad \ldots \\
&\quad \mid \mathsf{ref.null}\ \mathit{reftype} \mid \cdots \\
&\quad \mid \mathsf{drop} \mid \mathsf{select}\ (\mathit{valtype}^*)^? \\
&\quad \mid \mathsf{local.get}\ \mathit{localidx} \mid \mathsf{local.set}\ \mathit{localidx} \\
&\quad \mid \mathsf{global.get}\ \mathit{globalidx} \mid \mathsf{global.set}\ \mathit{globalidx} \\
&\quad \ldots \\
&\quad \mid \mathsf{nop} \mid \mathsf{unreachable} \\
&\quad \mid \mathsf{block}\ \mathit{blocktype}\ \mathit{instr}^*\ \mathsf{end} \\
&\quad \mid \mathsf{loop}\ \mathit{blocktype}\ \mathit{instr}^*\ \mathsf{end} \\
&\quad \mid \mathsf{if}\ \mathit{blocktype}\ \mathit{instr}^*\ \mathsf{else}\ \mathit{instr}^*\ \mathsf{end} \\
&\quad \mid \mathsf{br}\ \mathit{labelidx} \mid \cdots \mid \mathsf{return} \\
&\quad \mid \mathsf{call}\ \mathit{funcidx} \mid \mathsf{call\_indirect}\ \mathit{tableidx}\ \mathit{typeidx} \\
&\quad \ldots
\end{aligned}
$$

$$
\begin{aligned}
\mathit{instr} &::= \cdots && \text{(admininstr)} \\
&\quad \mid \mathsf{trap} \mid \mathsf{ref}\ \mathit{funcaddr} \mid \mathsf{ref.extern}\ \mathit{externaddr} \\
&\quad \mid \mathsf{invoke}\ \mathit{funcaddr} \\
&\quad \mid \mathsf{label}_n\{\mathit{instr}^*\}\ \mathit{instr}^*\ \mathsf{end} \\
&\quad \mid \mathsf{frame}_n\{\mathit{framestate}\}\ \mathit{instr}^*\ \mathsf{end}
\end{aligned}
$$

$$
\mathit{expr} ::= \mathit{instr}^*\ \mathsf{end} \qquad \text{(exp)}
$$

$$
\begin{aligned}
\mathit{module} ::= \{ & && \text{(module)} \\
& \mathsf{types}\ \ \mathit{vec}(\mathit{functype}), \\
& \mathsf{funcs}\ \ \mathit{vec}(\mathit{func}), \\
& \mathsf{tables}\ \ \mathit{vec}(\mathit{table}), \\
& \mathsf{mems}\ \ \mathit{vec}(\mathit{mem}), \\
& \mathsf{globals}\ \ \mathit{vec}(\mathit{global}), \\
& \mathsf{elems}\ \ \mathit{vec}(\mathit{elem}), \\
& \mathsf{datas}\ \ \mathit{vec}(\mathit{data}), \\
& \mathsf{start}\ \ \mathit{start}^?, \\
& \mathsf{imports}\ \ \mathit{vec}(\mathit{import}), \\
& \mathsf{exports}\ \ \mathit{vec}(\mathit{export})\ \ \}
\end{aligned}
$$

$$
\mathit{func} ::= \{\ \mathsf{type}\ \mathit{typeidx},\ \mathsf{locals}\ \mathit{vec}(\mathit{valtype}),\ \mathsf{body}\ \mathit{expr}\ \} \qquad \text{(func)}
$$

$$\mathit{global} ::= \{\ \mathsf{type}\ \mathit{memtype},\ \mathsf{init}\ \mathit{expr}\ \} \tag{global}$$

$$\mathit{table} ::= \{\ \mathsf{type}\ \mathit{tabletype}\ \} \tag{table}$$

$$\mathit{mem} ::= \{\ \mathsf{type}\ \mathit{memtype}\ \} \tag{mem}$$

$$\begin{aligned} \mathit{context} ::= \{& \\ & \mathsf{types}\ \ \mathit{functype}^*, \\ & \mathsf{funcs}\ \ \mathit{functype}^*, \\ & \mathsf{tables}\ \ \mathit{tabletype}^*, \\ & \mathsf{mems}\ \ \mathit{memtype}^*, \\ & \mathsf{globals}\ \ \mathit{globaltype}^*, \\ & \mathsf{elems}\ \ \mathit{reftype}^*, \\ & \mathsf{datas}\ \ \mathsf{ok}^*, \\ & \mathsf{locals}\ \ \mathit{valtype}^*, \\ & \mathsf{labels}\ \ \mathit{resulttype}^*, \\ & \mathsf{return}\ \ \mathit{resultytpe}^?, \\ & \mathsf{refs}\ \ \mathit{funcidx}^*\ \} \end{aligned} \tag{context}$$

$$\begin{aligned} \mathit{store} ::= \{& \\ & \mathsf{funcs}\ \ \mathit{funcinst}^*, \\ & \mathsf{tables}\ \ \mathit{tableinst}^*, \\ & \mathsf{mems}\ \ \mathit{meminst}^*, \\ & \mathsf{globals}\ \ \mathit{globalinst}^*, \\ & \mathsf{elems}\ \ \mathit{eleminst}^*, \\ & \mathsf{datas}\ \ \mathit{datainst}^*\ \ \} \end{aligned} \tag{store}$$

$$\begin{aligned} \mathit{funcinst} &::= \{\ \mathsf{type}\ \mathit{functype},\ \mathsf{module}\ \mathit{moduleinst},\ \mathsf{code}\ \mathit{func}\ \} \\ &\quad\ \mid \{\ \mathsf{type}\ \mathit{functype},\ \mathsf{hostfunc}\ \mathit{hostfunc}\ \} \\ \mathit{hostfunc} &::= \cdots \\ \mathit{tableinst} &::= \{\ \mathsf{type}\ \mathit{tabletype},\ \mathsf{elem}\ \mathit{vec}(\mathit{ref})\ \} \\ \mathit{meminst} &::= \{\ \mathsf{type}\ \mathit{memtype},\ \mathsf{data}\ \mathit{vec}(\mathit{byte})\ \} \\ \mathit{globalinst} &::= \{\ \mathsf{type}\ \mathit{globaltype},\ \mathsf{value}\ \mathit{val}\ \} \\ \mathit{eleminst} &::= \{\ \mathsf{type}\ \mathit{reftype},\ \mathsf{elem}\ \mathit{vec}(\mathit{ref})\ \} \\ \mathit{datainst} &::= \{\ \mathsf{data}\ \mathit{vec}(\mathit{byte})\ \} \\ \mathit{exportinst} &::= \{\ \mathsf{name}\ \mathit{name},\ \mathsf{value}\ \mathit{externval}\ \} \end{aligned} \tag{inst}$$

$$\mathit{framestate} ::= \{\ \mathsf{locals}\mathit{val}^*,\ \mathsf{module}\mathit{moduleinst}^*\ \} \tag{framestate}$$

$$\begin{aligned} \mathit{moduleinst} ::= \{& \\ & \mathsf{type}\mathit{functype}^*, \end{aligned} \tag{moduleinst}$$

$\mathsf{funcaddrs}\, \mathit{funcaddr}^*,$
$\mathsf{tableaddrs}\, \mathit{tableaddr}^*,$
$\mathsf{memaddrs}\, \mathit{memaddr}^*,$
$\mathsf{globaladdrs}\, \mathit{globaladdr}^*,$
$\mathsf{elemaddrs}\, \mathit{elemaddr}^*,$
$\mathsf{dataaddrs}\, \mathit{dataaddr}^*,$
$\mathsf{exports}\, \mathit{exportinst}^* \quad \}$

$$B^0 ::= \mathit{val}^*\ [\_]\ \mathit{instr}^* \qquad \text{(blockcontext)}$$
$$B^{k+1} ::= \mathit{val}^*\ \mathsf{label}_n\{\mathit{instr}^*\}\ B^k\ \mathsf{end}\ \mathit{instr}^*$$

$$E ::= [\_] \mid \mathit{val}^*\ E\ \mathit{instr}^* \mid \mathsf{label}_n\{\mathit{instr}^*\}\ E\ \mathsf{end} \qquad \text{(evalcontext)}$$

A.2 Validation

$$\frac{}{C \vdash t.\mathsf{const}\ c : [] \to [t]} \qquad \text{(T-const)}$$

$$\frac{}{C \vdash t.\mathit{unop} : [t] \to [t]} \qquad \frac{}{C \vdash t.\mathit{binop} : [t\ t] \to [t]} \qquad \text{(T-numeric)}$$

$$\frac{}{C \vdash \mathsf{drop} : [t] \to []} \qquad \frac{}{C \vdash \mathsf{select} : [t\ t\ \mathsf{i32}] \to [t]} \qquad \text{(T-parametric)}$$

$$\frac{C.\mathsf{locals}[x] = t}{C \vdash \mathsf{local.get} : [] \to [t]} \qquad \frac{C.\mathsf{locals}[x] = t}{C \vdash \mathsf{local.set} : [t] \to []} \qquad \text{(T-variable)}$$

$$\frac{C.\mathsf{globals}[x] = \mathit{mut}\ t}{C \vdash \mathsf{global.get} : [] \to [t]} \qquad \frac{C.\mathsf{globals}[x] = \mathsf{var}\ t}{C \vdash \mathsf{global.set} : [t] \to []}$$

$$\frac{}{C \vdash \mathsf{nop} : [] \to []} \qquad \frac{}{C \vdash \mathsf{unreachable} : [t_1^*] \to [t_2^*]} \qquad \text{(T-control)}$$

$$\frac{C \vdash \mathit{blocktype} : [t_1^*] \to [t_2^*] \quad C, \mathsf{labels}\ [t_2^*] \vdash \mathit{instr}^* : [t_1^*] \to [t_2^*]}{C \vdash \mathsf{block}\ \mathit{blocktype}\ \mathit{instr}^*\ \mathsf{end} : [t_1^*] \to [t_2^*]}$$

$$\frac{C \vdash \mathit{blocktype} : [t_1^*] \to [t_2^*] \quad C, \mathsf{labels}\ [t_2^*] \vdash \mathit{instr}^* : [t_1^*] \to [t_2^*]}{C \vdash \mathsf{loop}\ \mathit{blocktype}\ \mathit{instr}^*\ \mathsf{end} : [t_1^*] \to [t_2^*]}$$

$$\frac{C.\mathsf{labels}[l] = [t^*]}{C \vdash \mathsf{br}\ l : [t_1^*\ t^*] \to [t_2^*]} \qquad \frac{C.\mathsf{return} = [t^*]}{C \vdash \mathsf{return} : [t_1^*\ t^*] \to [t_2^*]}$$

$$\frac{\vdash \mathit{limits} : 2^{32} - 1}{\vdash \mathit{limits}\ \mathit{reftype}\ \mathsf{ok}} \qquad \frac{\vdash \mathit{limits} : 2^{16}}{\vdash \mathit{limits}\ \mathit{reftype}\ \mathsf{ok}} \qquad \text{(T-type)}$$

$$\frac{n \leq k \quad (m \leq k)^? \quad (n \leq m)^?}{\vdash \{\mathsf{min}\ n, \mathsf{max}\ m^?\} : k}$$

A.3 Execution

$$\begin{aligned}
(t.\mathsf{const}\ c_1)\ t.\mathit{unop} &\hookrightarrow (t.\mathsf{const}\ c) && \text{(R-numeric)}\\
&\quad (\text{if } c \in \mathit{unop}_t(c_1))\\
(t.\mathsf{const}\ c_1)\ t.\mathit{unop} &\hookrightarrow \mathsf{trap}\\
&\quad (\text{if } \mathit{unop}_t(c_1) = \emptyset)\\
(t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ t.\mathit{binop} &\hookrightarrow (t.\mathsf{const}\ c)\\
&\quad (\text{if } c \in \mathit{binop}_t(c_1, c_2))\\
(t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ t.\mathit{binop} &\hookrightarrow \mathsf{trap}\\
&\quad (\text{if } \mathit{binop}_t(c_1, c_2) = \emptyset)\\
(t.\mathsf{const}\ c_1)\ t.\mathit{testop} &\hookrightarrow (t.\mathsf{const}\ c)\\
&\quad (\text{if } c \in \mathit{testop}_t(c_1))\\
(t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ t.\mathit{relop} &\hookrightarrow (t.\mathsf{const}\ c)\\
&\quad (\text{if } c \in \mathit{relop}_t(c_1, c_2))
\end{aligned}$$

$$\begin{aligned}
\mathit{val}\ \mathsf{drop} &\hookrightarrow \epsilon && \text{(R-parametric)}\\
\mathit{val}_1\ \mathit{val}_2\ (\mathsf{i32.const}\ c)\ (\mathsf{select}\ t^?) &\hookrightarrow \mathit{val}_1\\
&\quad (\text{if } c \neq 0)\\
\mathit{val}_1\ \mathit{val}_2\ (\mathsf{i32.const}\ c)\ (\mathsf{select}\ t^?) &\hookrightarrow \mathit{val}_2\\
&\quad (\text{if } c = 0)
\end{aligned}$$

$$\begin{aligned}
F; (\mathsf{local.get}\ x) &\hookrightarrow F; \mathit{val} && \text{(R-variable)}\\
&\quad (\text{if } F.\mathsf{locals}[x] = \mathit{val})\\
F; (\mathsf{local.set}\ x) &\hookrightarrow F'; \mathit{val}\\
&\quad (\text{if } F' = F \text{ with } \mathsf{locals}[x] = \mathit{val})\\
S; F; (\mathsf{global.get}\ x) &\hookrightarrow S; F; \mathit{val}\\
&(\text{if } S.\mathsf{globals}[F.\mathsf{module.globaladdrs}[x]].\mathsf{value} = \mathit{val})\\
S; F; (\mathsf{global.set}\ x) &\hookrightarrow S'; F; \mathit{val}\\
&(\text{if } S' = S \text{ with } \mathsf{globals}[F.\mathsf{module.globaladdrs}[x]].\mathsf{value} = \mathit{val})
\end{aligned}$$

$$\begin{aligned}
\mathsf{nop} &\hookrightarrow \epsilon && \text{(R-control)}\\
\mathsf{unreachable} &\hookrightarrow \mathsf{trap}
\end{aligned}$$

$$\begin{aligned}
F; \mathit{val}^m\ \mathsf{block}\ \mathit{bt}\ \mathit{instr}^*\ \mathsf{end} &\hookrightarrow F; \mathsf{label}_n\{\epsilon\}\ \mathit{val}^m\ \mathit{instr}^*\ \mathsf{end}\\
&\quad (\text{if } \mathsf{expand}_F(\mathit{bt}) = [t_1^m] \to [t_2^n])\\
F; \mathit{val}^m\ \mathsf{loop}\ \mathit{bt}\ \mathit{instr}^*\ \mathsf{end} &\hookrightarrow F; \mathsf{label}_n\{\mathsf{loop}\ \mathit{bt}\ \mathit{instr}^*\ \mathsf{end}\}\ \mathit{val}^m\ \mathit{instr}^*\ \mathsf{end}\\
&\quad (\text{if } \mathsf{expand}_F(\mathit{bt}) = [t_1^m] \to [t_2^n])
\end{aligned}$$

$$\begin{aligned}\mathsf{expand}_F(\mathit{typeidx}) &= F.\mathsf{module}.\mathsf{types}[\mathit{typeidx}]\\ \mathsf{expand}_F([\mathit{valtype}^?]) &= [] \to [\mathit{valtype}^?]\end{aligned}$$

$$\begin{aligned}\mathsf{label}_n\{\mathit{instr}^*\}\ \mathit{val}^n\ \mathsf{end} &\hookrightarrow \mathit{val}^* && \text{(R-block)}\\ \mathsf{label}_n\{\mathit{instr}^*\}\ B^l[\mathit{val}^n\ (\mathsf{br}\ l)]\ \mathsf{end} &\hookrightarrow \mathit{val}^n\ \mathit{instr}^*\end{aligned}$$

$$\begin{aligned}S; F; E[\mathit{instr}^*] &\hookrightarrow S'; F'; E[\mathit{instr}'^*] && \text{(R-eval)}\\ &\quad (\text{if } S; F; \mathit{instr}^* \hookrightarrow S'; F'; \mathit{instr}'^*)\\ S; F; \mathsf{frame}_n\{F'\}\ \mathit{instr}^*\ \mathsf{end} &\hookrightarrow S'; F; \mathsf{frame}_n\{F''\}\ \mathit{instr}'^*\ \mathsf{end}\\ &\quad (\text{if } S; F'; \mathit{instr}^* \hookrightarrow S'; F''; \mathit{instr}'^*)\end{aligned}$$

$$\begin{aligned}S; F; E[\mathsf{trap}] &\hookrightarrow S; F; \mathsf{trap}\\ S; F; \mathsf{frame}_n\{F'\}\ \mathsf{trap}\ \mathsf{end} &\hookrightarrow S; F; \mathsf{trap}\end{aligned}$$

Appendix B

SpecTec

The following provides the complete grammars of the SpecTec DSL and IL for reference. Details of their semantics can be found in their respective documentations [Spe25b, Spe25a].

B.1 DSL Grammar

```
x*sep ::=
  eps
  x
  x sep x*sep

digit ::= "0" | ... | "9"
hex ::= digit | "A" | ... | "F"

num ::= digit+ | "0x" hex+ | "U+" hex+ | "`" digit+
bool ::= "true" | "false"
text ::= """ utf8* """

upletter ::= "A" | ... | "Z"
loletter ::= "a" | ... | "z"

upid ::= (upletter | "`" loletter | "_") (upletter | digit | "_" | "." | "'")*
loid ::= (loletter | "`" upletter | "`_") (loletter | digit | "_" | "'")*
id ::= upid | loid

atomid ::= upid | "infinity" | "_|_" | "^|^"
varid ::= loid
gramid ::= id
defid ::= id
relid ::= id
ruleid ::= id
subid ::= ("/" | "-") ruleid

atomop ::=
  "in" | ":" | ";" | "\" | <:"
  "<<" | ">>"
  "|-" | "-|"
  ":=" | "~~" | "~~_"
  "->" | "~>" | "~>*" | "=>"
  "`." | ".." | "..."
  "`?" | "`+" | "`*"
  "(/\)" | "(\/)" | "(+)" | "(*)" | "(++)"
  ":_" | "=_" | "==_" | "->_" | "=>_" | "~>_" | "~>*_" | "|-_" | "-|_"

numtyp ::=
  "nat"                                natural numbers
  "int"                                integer numbers
  "rat"                                rational numbers
  "real"                               real numbers

typ ::=
  varid args                           type name
  "bool"                               booleans
  "text"                               text strings
  numtyp                               numbers
  typ iter                             iteration
  "(" typ*"," ")"                      parentheses or tupling

iter ::=
  "?"                                  optional
  "*"                                  list
  "+"                                  non-empty list
  "^" arith                            list of specific length
  "^" "(" id "<" arith ")"             list of specific length with index

deftyp ::=
  typ                                       alias
  contyp                                    constructor
  ("..."? "|")? casetyp+"|" ("|" "...")?    variant
  rangetyp+"|"                              range / enumeration
  "{" fieldtyp+"," ","? "}"                 record

contyp   ::= nottyp hint* ("--" premise)*
casetyp  ::= nottyp hint* ("--" premise)*
fieldtyp ::= atom typ hint* ("--" premise)*
rangetyp ::= exp | "..."

nottyp ::=
  typ                                       plain type
  atomid                                    atom
  atomop nottyp                             infix atom
  nottyp atomop nottyp                      infix atom
  nottyp nottyp                             sequencing
  "(" nottyp ")"                            parentheses
  "`" "(" nottyp ")"                        custom brackets
  "`" "[" nottyp "]"
  "`" "{" nottyp "}"
  nottyp iter                               iteration

notop ::= "~"
logop ::= "/\" | "\/" | "=>"
cmpop ::= "=" | "=/=" | "<" | ">" | "<=" | ">="
exp ::=
  varid                                meta variable
  bool                                 Boolean literal
  num                                  natural number literal
  text                                 text literal
  notop exp                            logical negation
  exp logop exp                        logical connective
  exp cmpop exp                        comparison
  "eps"                                empty sequence
  exp exp                              sequencing
  exp iter                             iteration
  "[" exp* "]"                         list
  exp "[" arith "]"                    list indexing
  exp "[" arith ":" arith "]"          list slicing
  exp "[" path "=" exp "]"             list update
  exp "[" path "=++" exp "]"           list extension
  "{" (atom exp)*"," "}"               record
  exp "." atom                         record access
  exp "," exp                          record extension
  exp "++" exp                         list and record composition
  exp "<-" exp                         list membership
  "|" exp "|"                          list length
  "||" gramid "||"                     expansion length
  "(" exp*"," ")"                      parentheses or tupling
  "$" defid exp?                       function invocation
  atom                                 custom token
  atomop exp                           custom operator
  exp atomop exp
  "`" "(" exp ")"                      custom brackets
  "`" "[" exp "]"
  "`" "{" exp "}"
  "$" "(" arith ")"                    escape to arithmetic syntax
  "$" numtyp "$" "(" arith ")"         numeric conversion
  hole                                 hole
  exp "#" exp                          textual concatenation
  "##" exp                             remove possible parentheses

unop  ::= notop | "+" | "-"
binop ::= logop | "+" | "-" | "*" | "/" | "\" | "^"
arith ::=
  varid                                meta variable
  atom                                 token
  num                                  natural number literal
  unop arith                           unary operator
  arith binop arith                    binary operator
  arith cmpop arith                    comparison
  exp "[" arith "]"                    list indexing
  "(" arith ")"                        parentheses
  "(" arith iter ")"                   iteration (must not be "^exp")
  "|" exp "|"                          list length
  "$" defid args                       function invocation
  "$" "(" exp ")"                      escape back to general expression syntax
  "$" numtyp "$" "(" arith ")"         numeric conversion

path ::=
  path? "[" arith "]"                  list element
  path? "[" arith ":" arith "]"        list slice
  path? "." atom                       record element


hole ::=
  "%"                                  use next operand
  "%"digit*                            use numbered operand
  "%%"                                 use all operands
  "!%"                                 empty expression
  "%latex" "(" text* ")"               literal latex

sym ::=
  gramid args
  text
  num
  "$" "(" arith ")"
  "eps"
  "(" sym*"," ")"
  sym iter
  exp ":" sym
  sym sym
  sym "|" sym
  sym "|" "..." "|" sym

prod ::=
  sym "=>" exp ("--" premise)*

gram ::=
  ("..."? "|")? prod+"|" ("|" "...")?

args ::= ("(" arg*"," ")")?
arg ::=
  exp
  "syntax" typ
  "grammar" sym
  "def" defid

params ::= ("(" param*"," ")")?
param ::=
  (varid ":") typ
  "syntax" synid
  "grammar" gramid ":" typ
  "def" "$" defid params ":" typ

def ::=
  "syntax" varid params hint*                             syntax declaration
  "syntax" varid subid* params hint* "=" deftyp           syntax definition
  "grammar" gramid subid* params ":" typ hint* "=" gram   grammar definition
  "relation" relid hint* ":" nottyp                       relation declaration
  "rule" relid subid* hint* ":" exp ("--" premise)*  rule
  "var" varid ":" typ hint*                               variable declaration
  "def" "$" defid params ":" typ hint*                    function declaration
  "def" "$" defid args "=" exp ("--" premise)*            function clause
  "syntax" varid subid* atom? hint+                       outlined hints
  "grammar" gramid subid* hint*
  "relation" relid hint+
  "rule" relid subid* hint+
  "var" varid hint+
  "def" "$" defid hint+

premise ::=
  "var" id ":" typ                                        local variable
  ↪  declaration
  relid ":" exp                                           relational premise
  "if" exp                                                side condition
  "otherwise"                                             fallback side
  ↪  condition
  "(" premise ")" iter*                                     iterated relational
  ↪  premise
  "--"                                                      separator

hint ::=
  "hint" "(" hintid exp ")"                                 hintargs ::= ("("
  ↪  arg*"," ")")?

script ::=
  def*
```

B.2 IL Grammar

```
bool ::= "true" | "false"
text ::= """ char* """
id   ::= text
mixop ::= text

sign ::= "+" | "-"
nat  ::= digit+
int  ::= sign nat
rat  ::= sign? nat "/" nat
real ::= sign? nat "." nat
num  ::= "nat" nat | "int" int | "rat" rat | "real" real

unop  ::= "not" | "plus" | "minus" | "plusminus" | "minusplus"
binop ::= "and" | "or" | "impl" | "equiv" | "add" | "sub" | "mul" | "div" | "mod"
↪  | "pow"
cmpop ::= "eq" | "ne" | "lt" | "gt" | "le" | "ge"

iter ::=
  "opt"                       ?
  "list"                      *
  "list1"                     +
  "listn" exp id?             ^n, ^(i<n)

booltyp ::= "bool"
numtyp  ::= "nat" | "int" | "rat" | "real"
texttyp ::= "text"
optyp   ::= booltyp | numtyp

typ ::=
  "var" id                    t
  booltyp                     bool
  numtyp                      nat, int, ...
  texttyp                     text
  "tup" typbind*              ( typ , ... , typ )
  "iter" typ iter             typ*, typ+, ...

deftyp ::=
  "alias" typ                 typ
  "struct" typfield*          { field , ... , field }
  "variant" typcase*          case | ... | case

typbind  ::= "bind" exp typ
typfield ::= "field" mixop bind* typ prem*
typcase  ::= "case" mixop bind* typ prem*
```

```
exp ::=
  "var" id                        x
  "bool" bool                     true, false
  "num"  num                      0, -2
  "text" text                     "text"
  "un" unop optyp exp             <op> exp
  "bin" binop optyp exp exp       exp <op> exp
  "cmp" cmpop optyp exp exp       exp <cmp> exp
  "idx" exp exp                   exp[exp]
  "slice" exp exp exp             exp[exp : exp]
  "upd" exp path exp              exp[path = exp]
  "ext" exp path exp              exp[path =++ exp]
  "struct" expfield*              { atom exp, ... , atom exp }
  "dot" exp mixop                 exp.atom
  "comp" exp exp                  exp ++ exp  (on records)
  "mem" exp exp                   exp <- exp
  "len" exp                       |exp|
  "tup" exp*                      (exp, ..., exp)
  "call" id arg*                  $x(arg, ..., arg)?
  "iter" exp iter dom*            exp?, exp*, ...
  "case" mixop exp                atom exp
  "list" exp?                     exp ... exp or [exp ... exp]
  "cat" exp exp                   exp ++ exp  (on lists)

expfield ::= "field" mixop exp e

path ::=
  "root"                          .
  "idx" path exp                  path[exp]
  "slice" path exp exp            path[exp : exp]
  "dot" path mixop                path.atom

exp ::= ...
  "proj" exp nat                  tuple projection exp.i
  "uncase" exp mixop              inverse of "case"
  "opt" exp?                      option value (eps or singletong value)
  "unopt" exp                     inverse of "opt"
  "lift" exp                      conversion from t? to t*
  "cvt" numtyp numtyp exp         conversion from first to second numeric type
  "sub" typ typ exp               subsumption from first to second type

dom ::= "dom" id exp              x <- exp

sym ::=
  "var" id arg*                   x(arg, ..., arg)
  "num" nat                       0x12
  "text" text                     "text"
  "eps"                           eps
  "seq" sym*                      sym ... sym
  "alt" sym*                      sym | ... | sym
  "range" sym sym                 sym | "..." | sym
  "iter" sym iter dom             sym?, sym+, ...
  "attr" exp sym                  exp:sym

prem ::=
  "rule" id mixop exp             -- id: mixop-exp
  "if" exp                        -- if exp
  "else"                        -- otherwise
  "let" exp exp                 -- if exp = exp (when one side introduces
  ↪  variables)
  "iter" prem iter dom          -- prem*

def ::=
  "typ" id param* inst*         syntax x(param*) with instance definitions
  "rel" id mixop typ rule*      relation x: mixop-typ with rules
  "def" id param* typ clause*   def $x(param*) : typ with clauses
  "gram" id param* typ prod*    grammar x(param*) : typ with productions
  "rec" def*                    inferred recursion group

inst   ::= "inst" bind* arg* deftyp dt       syntax _(arg*) = deftyp
rule   ::= "rule" id bind* mixop exp prem*   rule _/x mixop-exp -- prem*
clause ::= "clause" bind* arg* exp* prem*    def $_(arg*) = exp -- prem*
prod   ::= "prod" bind* sym exp prem*        | sym => exp -- prem*

param ::=
  "exp", id typ                 x : typ
  "typ", id                     syntax x
  "def", id param* typ          def $x(param*) : typ
  "gram", id typ                grammar x : typ

arg ::=
  "exp" exp                     exp
  "typ" typ                     syntax typ
  "def" id                      def $x
  "gram" sym                    grammar sym

bind ::=
  "exp" id typ
  "typ" id
  "def" id param* typ
  "gram" id param* typ

script ::= def*
```