# Verifying Verus: A Lean 4 Formalization of the SST-to-AIR Expression Translation

Shuge Rong

May 2026

Committee:
Jeremy Avigad (Chair)
Professor of Philosophy and Mathematical Sciences, Carnegie Mellon University
Bryan Parno
Kavčić-Moura Professor of Electrical & Computer Engineering and Computer Science,
Carnegie Mellon University

*A thesis submitted in partial fulfillment of the requirements for the degree of*
*Master of Science in Logic, Computation, and Methodology*
*to the Department of Philosophy, Carnegie Mellon University*

**Abstract**

Verus is an SMT-based verification tool that checks specifications and proofs written alongside Rust source code. Like any verification tool built around a compilation pipeline, the guarantees it provides depend on the correctness of each translation pass that takes the user's program down to the queries dispatched to an SMT solver. This thesis addresses one such pass: the translation from Verus's Statement-oriented Syntax Tree (SST) to its Assertion Intermediate Representation (AIR), the last language-specific stage before the SMT-LIB encoding.

We formalize this translation in the Lean 4 theorem prover. On the source side, we model a substantial fragment of SST, including its types, expressions, static type system, and denotational semantics. We then give it an operational semantics that mirrors the Verus interpreter. We claim that we are able to prove in this framework the interpreter sound and type-preserving with respect to the denotational semantics. On the target side, we formalize AIR as an instance of a generic multi-sorted first-order logic framework, instantiating it with AIR's sorts, function symbols, and relation symbols. We also pin down the definition of a model of the AIR language. The translation is defined as a Lean function that produces, for each typed SST expression, an AIR term or formula together with a set of accumulated axioms. We then state a soundness theorem: if the translations of two SST expressions are equivalent in every AIR model satisfying the axioms, and the SST and AIR variable valuations are coherent, then the original expressions are denotationally equal. We give proofs of representative cases and identify the remaining cases and lemmas needed to complete the mechanization.

This development demonstrates that the end-to-end, proof-assistant-based approach to compiler verification scales to a real SMT-based verification tool, and provides infrastructure on which a fuller mechanized correctness proof for Verus can be built.

# Contents

# 1 Introduction

Formal verification of systems software has made remarkable progress in recent years. Tools such as Verus [11] allow developers to write Rust code annotated with specifications and to obtain, at compile time, machine-checked guarantees that the code meets those specifications for all possible executions. The strength of these guarantees, however, rests on a critical assumption: that the verification tool itself is correct. Verus, like any verification tool built around a compiler pipeline, transforms the user's program through a sequence of intermediate representations before producing queries for an SMT solver. A bug in any of these translation passes could silently invalidate the verification result, causing Verus to accept a program that does not actually satisfy its specification.

This thesis addresses that concern by formally verifying one of the translation passes inside the Verus compiler. Specifically, we target the translation from the Statement-oriented Syntax Tree (SST) to the Assertion Intermediate Representation (AIR), the pass that generates the verification conditions ultimately dispatched to the Z3 SMT solver [7]. We formalize the syntax and denotational semantics of both SST and AIR in the Lean 4 theorem prover [8], define the translation as a Lean function, and state a soundness theorem asserting that the translation preserves semantic equivalence. Our development builds on a multi-sorted first-order logic framework to model AIR's sort structure and uses dependent types to ensure that only well-typed SST expressions are interpreted.

The remainder of this chapter provides the necessary background. Section 1.1 introduces the Verus verification tool and its internal compilation pipeline. Section 1.2 surveys the field of compiler verification and the two main approaches—end-to-end proof-assistant-based verification and lightweight SMT-based verification. Section 1.3 describes the Lean 4 theorem prover and the specific features we rely on in our formalization.

## 1.1 Verus and its Compilation Pipeline

Verus [11] is an SMT-based verification tool for the Rust programming language. Given executable Rust source code annotated with specifications and proofs, Verus statically checks

that the executable code satisfies these specifications for all possible executions. Rather than inserting run-time checks, Verus generates verification conditions (VCs) from specifications and proofs and discharge them with the Z3 SMT solver [7] at compile time.

A distinguishing feature of Verus is that specifications and proofs are themselves written in Rust syntax. The `verus!` macro extends Rust with specification constructs such as `forall`, `exists`, mathematical integer types (`int`, `nat`), and ghost code, while remaining compatible with Rust's type checker. This design avoids the need for systems programmers to learn a separate verification language, and allows specifications and proofs to exploit Rust's existing features: algebraic datatypes, pattern matching, traits, closures, and, crucially, its linear ownership and borrowing discipline.

Another feature of Verus is that it organizes all code into three *modes* [1]:

- **Spec**: purely functional ghost code that describes properties about programs. Spec code is not checked for linearity or borrowing, and is always erased before compilation.
- **Proof**: ghost code proving that programs satisfy properties. Proof code is checked for linearity and borrowing but is also erased before compilation.
- **Exec**: ordinary Rust code that can be compiled and run. Exec code is checked for linearity and borrowing, and is compiled to machine code.

This design enables rich specifications and proofs that incur *zero* run-time overhead, since all ghost code is erased after verification succeeds. Since Verus trusts the results of Rust's borrow checker, it doesn't recheck the results, and can rely on the properties of borrowing in its SMT encoding.

**Compilation pipeline.** Verus is implemented as an extension of the Rust compiler. Internally, it compiles user-written code through a sequence of intermediate representations (IRs), each of which progressively simplifies the program for verification. The pipeline proceeds as follows [9]:

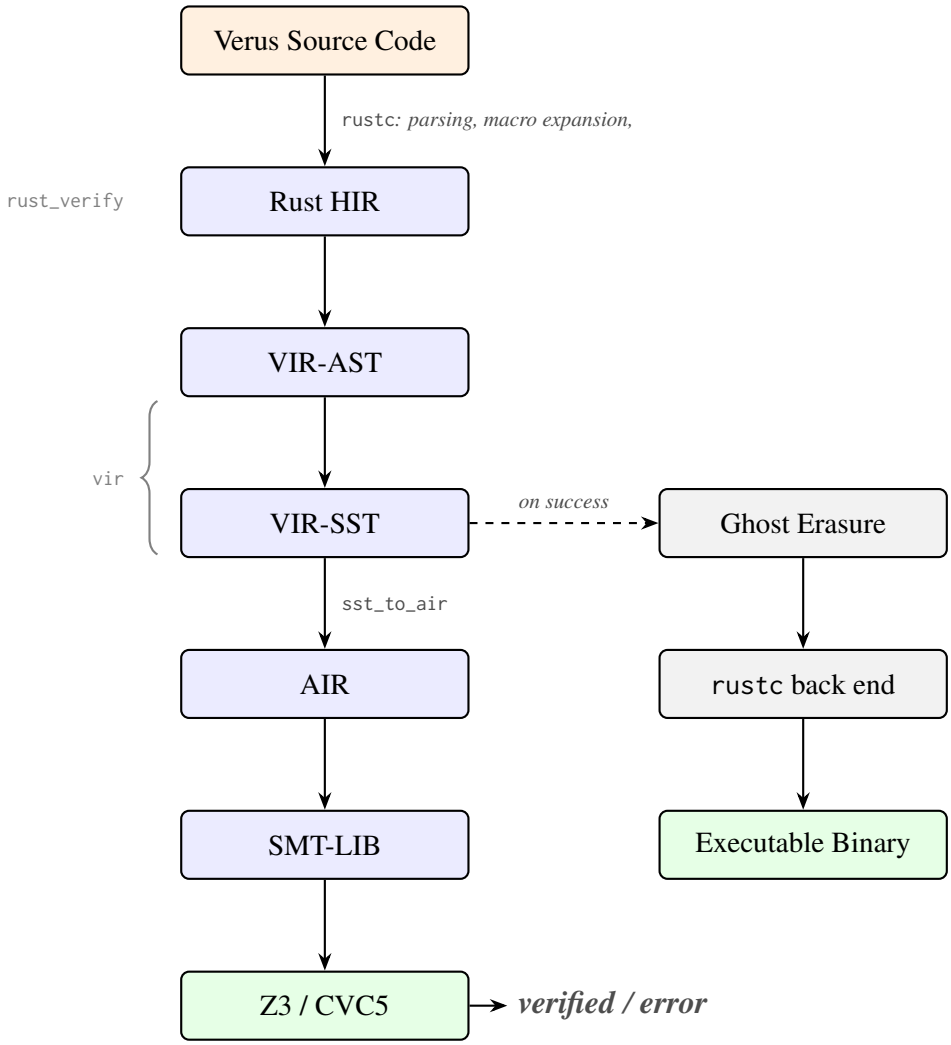


Figure 1: The Verus verification pipeline. Annotated Rust source code is translated through a sequence of IRs before VCs are dispatched to an SMT solver. Upon successful verification, ghost code is erased and the remaining executable Rust code is compiled by the standard `rustc` back end.

1. **Rust HIR.** The standard Rust compiler front end performs parsing and macro expansion to produce a *High-level Intermediate Representation* (HIR).

2. **HIR → VIR-AST.** HIR is then translated into the *Verification Intermediate Representation* (VIR), an abstract syntax that represents the elements of Rust code relevant to verification. Compared to the original Rust code, VIR focuses on values, rather than on how values are stored in memory. For example, `Box`, `Rc`, and `Arc` are irrelevant to

VIR and are not present in VIR.

VIR consists of two distinct abstract syntax trees, *VIR Abstract Syntax Tree* (VIR-AST) and *VIR Statement-oriented Syntax Tree* (VIR-SST). VIR-AST keeps the original tree structure of mutually nested Rust HIR expressions and statements. VIR-SST, on the other hand, disallows statements inside expressions and disallows side effects inside expressions.

3. **VIR-AST → VIR-SST.** The VIR-AST is converted into VIR-SST by moving statements out of expressions, and carrying out other transformations.

4. **VIR-SST → AIR.** The SST is translated into the *Assertion Intermediate Representation* (AIR), a language of `assert` and `assume` statements together with mutable updates to local variables. AIR's structure closely mirrors SMT-LIB [4].

5. **AIR → SMT-LIB.** The AIR is encoded into SMT-LIB format and dispatched to Z3. There is a verifier function that partitions SMT queries into independent *buckets* that can be verified in parallel across multiple cores, enabling fast turnaround even on large codebases.

After verification succeeds, Verus erases all ghost code (spec and proof mode) and hands the remaining executable Rust code to the standard Rust compiler back end for compilation to machine code. This two-phase approach—verify with ghost annotations, then compile without them—ensures that verified programs integrate seamlessly with the broader Rust ecosystem.

As described above, the Verus pipeline transforms programs through a chain of intermediate representations before producing verification conditions for the SMT solver. The correctness of the final verification result depends on the correctness of each translation step in this chain: a bug in any pass could cause Verus to accept a program that does not actually satisfy its specification, or reject one that satisfies its specification, undermining the guarantees it provides. This concern is not unique to Verus; it is an instance of the broader and well-studied problem of *compiler verification*, to which we now turn.

## 1.2 Compiler Verification

The correctness of a compiler is a critical foundation for the trust we place in compiled software. A bug in a compiler can transform a correct source program into an incorrect executable, leading to errors that are notoriously difficult to diagnose. This has motivated the field of compiler verification, which aims to provide mathematical guarantees that a compiler preserves the semantics of programs it translates. The end-to-end, proof-assistant-based verification and the lightweight, SMT-based verification are two standard approaches in compiler verification.

A landmark achievement for the end-to-end, proof-assistant-based approach is the CompCert C compiler [13], the first realistically usable compiler with a mechanically checked proof of correctness. For every translation from a source program to a target assembly, a machine-checkable proof in the Rocq proof assistant is constructed to show that the observable behavior of the target code refines the behavior of the source code. Another verified compiler is developed in the CakeML project [10]. CakeML is a fully verified ML stack with a formally specified semantics for the source language and a verified compiler down to machine code, all developed in the HOL4 theorem prover.

In parallel, a more automated, lightweight verification approach based on SMT-solvers has been developed. This line of work often targets intermediate representations (IRs), and a significant body of work has focused on the formalization of LLVM. These formalizations, such as Vellvm [22] and K-LLVM [14], which capture the semantics of core LLVM IR constructs, have served as a foundational bedrock for automated verification tools. Building upon these formal semantics, tools like Alive [16] and its successor Alive 2 [15] leverage SMT solvers to provide push-button verification to prove the correctness of LLVM optimizations.

In this thesis, we adopt the end-to-end, proof-assistant-based approach to verify a translation pass within the Verus compiler. Specifically, we target part of the SST-to-AIR translation (step 4 in the pipeline described in §1.1), formalizing both the source and target languages, the translation function, and a soundness proof entirely within the Lean 4 theorem prover [8]. We describe the relevant features of Lean 4 in the next subsection.

## 1.3 The Lean Theorem Prover

Lean 4 [8] is an open-source theorem prover and programming language developed principally by Leonardo de Moura and his collaborators. Built on the Calculus of Inductive Constructions (CIC), Lean 4 provides a dependently typed foundation in which propositions are types and proofs are terms, in the tradition of the Curry–Howard correspondence. Unlike its predecessor Lean 3, Lean 4 is also designed as a practical, compiled functional programming language with an efficient runtime, making it well suited for projects that combine formalisation with executable artefacts. We now highlight the features of Lean 4 that are most relevant to our formalisation.

**The `grind` tactic.** Lean 4's `grind` tactic is an automated proof procedure inspired by modern SMT solvers. It works by incrementally collecting facts and deriving new ones through a set of cooperating engines, including congruence closure, constraint propagation, E-matching, guided case analysis, and theory solvers for linear integer arithmetic and commutative rings. We rely on `grind` extensively in our substitution infrastructure for SST expressions, where it discharges the numerous equational obligations that arise when proving lemmas about parallel substitutions and de Bruijn index manipulation. Many of these goals involve chains of equalities, case splits on natural-number comparisons, and congruence reasoning over recursive syntax constructors—precisely the class of obligations that `grind`'s combination of engines handles well, often closing goals that would otherwise require several lines of manual tactic invocations.

**Noncomputable definitions.** Lean is designed to support both computational and classical reasoning. By default, it requires every definition to have computational content, which guarantees that closed expressions in the system evaluate to canonical normal forms. The `noncomputable` modifier relaxes this requirement, instructing Lean to skip code generation for a definition while still accepting it as a well-typed term. This is necessary whenever a definition depends on axioms that lack a computational interpretation, most notably the classical choice principle (`Classical.choice`) and its consequences such as the law of the excluded middle derived via Diaconescu's theorem [3]. We make use of `noncomputable` definitions in `exp_rep.lean`, where we define the denotational semantics of SST expressions. Because our interpretation functions rely on classical reasoning—for example, to decide equality on

types or to select witnesses from existential hypotheses—they cannot be compiled. But this is immaterial: the semantic functions serve exclusively as the reference specification against which we state and prove the soundness of the translation, and are never intended to be executed.

**Nested induction.** When an inductive type contains constructors whose arguments are themselves wrapped in another type constructor, the type is said to involve *nested* recursion. Lean 4's `induction` tactic does not yet generate induction principles that account for such nesting: the default eliminator provides an inductive hypothesis only for recursive occurrences that are not nested. In our formalisation, the SST expression type `Exp` contains several nested constructors, putting `Exp` itself inside a `List` or a product type. To reason by induction over `Exp`, we therefore define an induction principle (`Exp.induct`) manually. We register this principle with the `@[induction_eliminator]` attribute so that Lean's `induction` tactic uses it by default in place of the auto-generated recursor.

**External libraries.** Our development builds on two external Lean 4 libraries. We use the `Many-Sorted-Model-Theory` [21] library to provide a generic framework for multi-sorted first-order logic, which we instantiate with AIR's sort structure to obtain terms, formulas, structures, and satisfaction judgments. We also use selected components of Mathlib [17], primarily fortactics and basic algebraic lemmas.

# 2 SST: Statement-Oriented Syntax Tree

This chapter presents our formalization of the expression fragment of Verus's Statement-oriented Syntax Tree (SST). We describe the subset of SST that we model, define its syntax and typing rules, and give a denotational semantics in the style of [6]. After that, we discuss the operational semantics presented in the Verus interpreter.

## 2.1 Scope of the Formalization

The full SST defined in the Verus codebase[1] is an intermediate representation that covers both *expressions* (`ExpX`) and *statements* (`StmX`). Expressions are pure, side-effect-free terms; statements include function calls with side effects, assertions, assumptions, assignments, loops, and control flow. In this thesis we restrict attention to the expression fragment. Furthermore, we consider only expressions that arise in ghost mode (spec and proof mode).

Even within the ghost-mode expression fragment, the full `ExpX` in the Verus codebase comprises over twenty constructors, several of which deal with Rust-specific concerns such as mutable borrows (`VarLoc`, `Loc`), snapshot references (`Old`), fuel constants (`FuelConst`), and internal interpreter values (`Interp`). Our formalization covers the most fundamental variants, including constants, variables, boolean and arithmetic operations, conditionals, let-bindings, quantifiers, lambdas, function calls, and array literals. We leave more complex constructs to future work.

## 2.2 Syntax

We begin by formalizing the SST type system. In Verus, types are defined by the `Typ` enum in the VIR crate and include booleans, integers of various widths, floats, arrays, string slices, type parameters, spec-mode function types, type decorations (references, boxes, ghost wrappers), tuples, structs, and enums.

[1] https://github.com/verus-lang/verus/blob/main/source/vir/src/sst.rs

**Definition 2.1 – SST Types**

The syntax of SST types is given by the following grammar:

$$t ::= \alpha \mid \texttt{Bool} \mid \texttt{Int} \mid \texttt{Float32} \mid \texttt{Float64} \mid \texttt{String} \mid \texttt{Char}$$
$$\mid t_1 \to t_2 \mid \texttt{Array}(t) \mid \texttt{Box}(t) \mid \texttt{Struct}(s)(t_1, \ldots, t_n) \mid \texttt{Enum}(s)(t_1, \ldots, t_n)$$

Here $\alpha$ is a type parameter, $s$ ranges over names (strings).

Our Lean formalisation mirrors this structure closely:

```
inductive Typ where
  | _Bool
  | Int (i: IntRange)
  | Float (width: UInt32)
  | Array (t : Typ)
  | StrSlice
  | TypParam (i : String)
  | SpecFn (params : List Typ) (ret : Typ)
  | Decorated (dec : TypDecoration) (ty : Typ)
  | Tuple (t₁ t₂ : Typ)
  | Struct (name : Ident) (fields : List Typ)
  | Enum (name : Ident) (params : List Typ)
  ...
```

The expression grammar is given below. Constants $c$ include booleans, arbitrary-precision integers, characters, string slices, and floating-point bit patterns. Variables $x$ are represented as de Bruijn indices. Unary operators (uop) include boolean and bitwise negation, integer clipping, and float-to-bits coercion; binary operators (bop) include boolean connectives, equality, arithmetic, comparisons, bitwise operations, and array indexing.

**Definition 2.2 – SST Expressions**

> The syntax of SST expressions is defined as follows:
>
> $$e ::= c \mid x \mid \mathsf{uop}(e) \mid \mathsf{bop}(e_1, e_2) \mid \langle e_1, e_2 \rangle \mid \texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2$$
> $$\mid e_f(e_1, \ldots, e_n) \mid [e_1, \ldots, e_n] \mid \texttt{array } e_1[e_2] \mid \texttt{str } e_1[e_2]$$
> $$\mid \lambda x : t.e \mid \forall x : t.e \mid \exists x : t.e \mid \texttt{let } x = e_1 \texttt{ in } e_2 \mid \mathsf{box}(e) \mid \mathsf{unbox}(e) \mid \mathsf{has\_type}\ (e, t)$$
> $$\mid \texttt{struct } s\ \{f_1 = e_1, \ldots, f_n = e_n\} \mid \mathsf{enum}\ s\ \{f_1 = e_1, \ldots, f_n = e_n\} \mid \texttt{struct } s.f \mid \mathsf{enum}\ s.f$$
>
> Here $c$ ranges over constants, $x$ over variables, $s$ over names, and $f$ over field names.

Our Lean formalisation mirrors this structure closely:

```
inductive Exp where
  | Const (c : Const)
  | Var (x : Nat)
  | TupleCtor (e₁ e₂ : Exp)
  | Unary (op : UnaryOp) (arg : Exp)
  | Binary (op : BinaryOp) (arg₁ arg₂ : Exp)
  | If (cond branch₁ branch₂ : Exp)
  | Call (fn : CallFun) (typs : List Typ) (exps : List Exp) (ret : Typ)
  | ArrayLiteral (elems : List Exp)
  | ArrayIndex (e₁ e₂ : Exp)
  | StrIndex (e₁ e₂ : Exp)
  | Lambda (var : Typ) (exp : Exp)
  | Quant (q : Quant) (var : Typ) (body : Exp)
  | Let (tys : List Typ) (es : List Exp) (body : Exp)
  | Box (e : Exp)
  | Unbox (e : Exp)
  | StructCtor (dt : Ident) (fields : List (String × Exp))
  | EnumCtor (dt : Ident) (data : List (String × Exp))
  ...
```

## 2.3 Statics

We define a *type variable context* $\Delta$ that tracks which type variables are in scope:

**Definition 2.3 – Type Variable Context**

A type variable context $\Delta$ is a list of type variables:

$$\Delta ::= \cdot \mid \Delta,\ \alpha \text{ type}$$

The judgment $\Delta \vdash A$ type asserts that type $A$ is well-formed in context $\Delta$. Base types such as `Bool`, `Int`, `Float32`, `Float64`, `String`, and `Char` are always well-formed. Some illustrative rules are as follows:

$$\frac{\alpha \text{ type} \in \Delta}{\Delta \vdash \alpha \text{ type}}\ \textsc{Wf-Param} \qquad \frac{\Delta \vdash A \text{ type}}{\Delta \vdash \texttt{Array}(A) \text{ type}}\ \textsc{Wf-Array}$$

$$\frac{\forall t \in \overline{\tau},\ \Delta \vdash t \text{ type}}{\Delta \vdash \texttt{Struct}(n)(\overline{\tau}) \text{ type}}\ \textsc{Wf-Struct} \qquad \frac{\forall t \in \overline{\tau},\ \Delta \vdash t \text{ type}}{\Delta \vdash \texttt{Enum}(n)(\overline{\tau}) \text{ type}}\ \textsc{Wf-Enum}$$

**Definition 2.4 – Typing Context**

A typing context $\Gamma_\Delta$ with respect to a type variable context $\Delta$ is defined by:

$$\Gamma_\Delta ::= \cdot \mid \Gamma_\Delta,\ x : A$$

where $\Delta \vdash A$ type.

The judgment $\Gamma_\Delta \vdash e : A$ asserts that $e$ has type $A$ in the context $\Gamma_\Delta$. We list representative typing rules below. We omit the subscript $\Delta$ when the type variable context is clear from the surrounding discussion.

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ \text{T-Var} \qquad \frac{\Gamma \vdash e : \mathsf{Bool}}{\Gamma \vdash \neg e : \mathsf{Bool}}\ \text{T-Not} \qquad \frac{\Gamma \vdash e_1 : \mathsf{Bool} \qquad \Gamma \vdash e_2 : \mathsf{Bool}}{\Gamma \vdash e_1 \wedge e_2 : \mathsf{Bool}}\ \text{T-And}$$

$$\frac{\Gamma \vdash e_1 : A \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1 = e_2 : \mathsf{Bool}}\ \text{T-Eq} \qquad \frac{\Gamma \vdash e_1 : \mathsf{Int} \qquad \Gamma \vdash e_2 : \mathsf{Int}}{\Gamma \vdash e_1\, op\, e_2 : \mathsf{Int}}\ \text{T-Arith-Int}$$

$$\frac{\Gamma \vdash e : A}{\Gamma \vdash \mathsf{box}_A\ e : \mathsf{Box}(A)}\ \text{T-Box} \qquad \frac{\Gamma \vdash e : \mathsf{Box}(A)}{\Gamma \vdash \mathsf{unbox}\ e : A}\ \text{T-Unbox}$$

$$\frac{\Delta \vdash A\ \text{type} \qquad \Gamma \vdash e : B}{\Gamma \vdash \mathsf{hasType}_A\ e : \mathsf{Bool}}\ \text{T-HasType} \qquad \frac{\Gamma \vdash a : \mathsf{Array}(A) \qquad \Gamma \vdash i : \mathsf{Nat}}{\Gamma \vdash \mathsf{array}\ a[i] : A}\ \text{T-ArrayIndex}$$

$$\frac{\Gamma \vdash s : \mathsf{String} \qquad \Gamma \vdash i : \mathsf{Nat}}{\Gamma \vdash \mathtt{str}\ s[i] : \mathsf{Char}}\ \text{T-StrIndex} \qquad \frac{\Gamma \vdash e_1 : A \qquad \Gamma \vdash e_2 : B}{\Gamma \vdash \langle e_1, e_2 \rangle : A \times B}\ \text{T-Tuple}$$

$$\frac{\Gamma \vdash c : \mathsf{Bool} \qquad \Gamma \vdash e_1 : A \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash \mathtt{if}\ c\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : A}\ \text{T-If} \qquad \frac{\forall e \in \overline{e},\, \Gamma \vdash e : A}{\Gamma \vdash [\overline{e}] : \mathsf{Array}(A)}\ \text{T-Array}$$

$$\frac{\forall 0 \leq i < n,\, \Gamma \vdash e_i : t_i}{\Gamma \vdash \mathtt{struct}\ s\ \{f_1 = e_1, ..., f_n = e_n\} : \mathsf{Struct}(s)(t_1, ..., t_n)}\ \text{T-StructCtor}$$

$$\frac{\Gamma \vdash e_1 : A \qquad \Gamma, x : A \vdash e_2 : B}{\Gamma \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : B}\ \text{T-Let} \qquad \frac{\Gamma, x : A \vdash e : \mathsf{Bool}}{\Gamma \vdash Qx : A.\ e : \mathsf{Bool}}\ \text{T-Quant}$$

$$\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x : A.\ e : A \to B}\ \text{T-Lambda} \qquad \frac{\Gamma \vdash e_1 : A \to B \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash \mathsf{ap}(e_1, e_2) : B}\ \text{T-App}$$

$$\frac{\Gamma \vdash \mathtt{struct}\ s\ \{f_1 = e_1, ..., f_n = e_n\} : \mathsf{Struct}(s)(t_1, ..., t_n)}{\Gamma \vdash \mathtt{struct}\ s.f_i : t_i}\ \text{T-Proj}$$

$$\frac{\Gamma \vdash e : A}{\Gamma \vdash \mathsf{has\_type}\ (e, B) : \mathsf{Bool}}\ \text{T-HasType}$$

where $op \in \{+, -, \times, /, \%\}$, and $Q \in \{\forall, \exists\}$ .

## 2.4 Denotational Semantics

Following the approach of [6], we give a denotational semantics for SST expressions by defining a Lean function `exp_rep` that maps each well-typed expression to a value in the corresponding Lean type. The semantic interpretation is composed of four layers:

1. a type variable interpretation `type_env` that maps type parameters to Lean types;
2. a `typ_rep` function that assigns a Lean type to each SST type;
3. a variable valuation that assigns each variable a value in the domain of its type;
4. a function `exp_rep` that provides interpretations for expressions in the domain of their types.

**Type variable interpretation.** The type variable interpretation

```
type_env: String → Type
```

is a function that maps type parameters to Lean types.

**Interpretation of types.** The function `typ_rep` assigns a Lean type to each SST type. `Bool` is mapped to Lean's `Bool`, `Int` to Lean's `Int`, `Nat` to Lean's `Nat`, `Char` to Lean's `Char`, and `String` to Lean's `String`, etc. For types that are not yet handled (structs, enums, spec-mode functions, AIR sorts), the function delegates to an auxiliary interpretation `dom_aux` that depends on `type_env`.

```
def typ_rep (type_env : String → Type) (dom_aux : (String → Type) → Typ →
    Type) (t: Typ): Type :=
  match t with
  | ._Bool => Bool
  | .Int i =>
```

```
    match i with
    | IntRange.Int => Int
    | IntRange.Nat => Nat
    | IntRange.Char => Char
    ...
  | .Float w =>
    if w = 32 then UInt32
    else if w = 64 then UInt64
    else TypeError
  | .TypParam p => type_env p
  | .Array t' =>
    List (typ_rep type_env dom_aux t')
  | .StrSlice => String
  | .Tuple t1 t2 =>
    (typ_rep type_env dom_aux t1) × (typ_rep type_env dom_aux t2)
  ...
termination_by t
```

Floating-point types are interpreted via their bit-pattern representation: `Float32` is mapped to Lean's `UInt32` and `Float64` to `UInt64`. Lean's built-in `Float` type does not carry a width parameter, so it cannot distinguish 32-bit from 64-bit floats. Mapping each Verus float type to the unsigned integer of matching width preserves this distinction. The function `dom_aux` is intended to handle the interpretation of structs and enums. In a complete interpretation, `dom_aux` would take a type environment `type_env` as a parameter and assign a Lean type to each struct or enum. In our current formalization, however, structs and enums are out of scope of this thesis, so `dom_aux` is left as an abstract parameter.

In the subsequent parts, we omit `type_env`, `dom_aux` and write the interpretation of type $t$ as $[\![t]\!]$.

**Variable valuations.** A variable valuation `val_vars` for a typing context $\Gamma$ is a dependent function that assigns to each de Bruijn index $i < |\Gamma|$ a value of type (`typ_rep type_env dom_aux` $\Gamma[i]$):

```
def val_vars type_env (Γ: context) dom_aux :=  (i : Nat) → (_ : i < Γ.length
    ) → typ_rep type_env dom_aux Γ[i]
```

We write $v$ instead of `val_vars type_env Γ dom_aux` to mean variable valuations in the following parts.

**Interpretation of expressions.** The core of the semantics is the function `exp_rep`.

```
def exp_rep Γ type_env (venv: val_vars tenv Γ dom_aux) (t : Typ) (e : Exp) (
    hty : Γ ⊢ e : t): typ_rep type_env dom_aux t:=
  match e with
  ...
```

It takes a typing context $\Gamma$, a type parameter interpretation `type_env`, a variable valuation $v$, a target type $t$, an expression $e$, and a proof that $e$ has type $t$ in $\Gamma$, and produces a value of the appropriate semantic type. The function is defined by structural recursion on $e$.

We write $[\![e]\!]_v$ for the denotational interpretation of expression $e$ under variable valuation $v$. For constants, the subscript on the expression (e.g., $b_{\texttt{Bool}}$, $i_{\texttt{Int}}$) indicates the SST type of the constant, distinguishing boolean, integer, character, string, and floating-point literals. The notation $v[0 \mapsto d]$ denotes extending $v$ with a new de Bruijn binding that maps index 0 to $d$ and shifts all existing bindings by one. Division and modulus are totalised: when the divisor is zero, an unspecified value of the appropriate type is returned. Equality and disequality use classical decidability (`Classical.propDecidable`) to convert propositions $[\![e_1]\!]_v = [\![e_2]\!]_v$ and $[\![e_1]\!]_v \neq [\![e_2]\!]_v$ into booleans.

> **Definition 2.5 – Denotational Interpretation of Expressions**
>
> The denotational interpretation $[\![e]\!]_v$ is defined by recursion on $e$ as follows:
>
> $$[\![b_{\texttt{Bool}}]\!]_v = b$$
> $$[\![i_{\texttt{Int}}]\!]_v = i$$
> $$[\![n_{\texttt{Nat}}]\!]_v = n$$

$$
\begin{aligned}
[\![c_{\texttt{Char}}]\!]_v &= c \\
[\![s_{\texttt{String}}]\!]_v &= s \\
[\![f_{\texttt{Float32}}]\!]_v &= f \\
[\![x]\!]_v &= v(x) \\
[\![\neg e]\!]_v &= \neg [\![e]\!]_v \\
[\![e_1 \otimes e_2]\!]_v &= [\![e_1]\!]_v \otimes [\![e_2]\!]_v, \quad \otimes \in \{\wedge, \vee, \rightarrow\} \\
[\![e_1 = e_2]\!]_v &= [\![e_1]\!]_v = [\![e_2]\!]_v \\
[\![e_1 \neq e_2]\!]_v &= [\![e_1]\!]_v \neq [\![e_2]\!]_v \\
[\![e_1\, op\, e_2]\!]_v &= [\![e_1]\!]_v\, op\, [\![e_2]\!]_v, \quad op \in \{+, -, \times, \mathsf{div}, \mathsf{mod}\} \\
[\![\texttt{array}\ a[i]]\!]_v &= [\![a]\!]_v[[\![i]\!]_v] \\
[\![\texttt{str}\ s[i]]\!]_v &= [\![s]\!]_v[[\![i]\!]_v] \\
[\![\texttt{if}\ c\ \texttt{then}\ e_1\ \texttt{else}\ e_2]\!]_v &= \texttt{if}\ [\![c]\!]_v\ \texttt{then}\ [\![e_1]\!]_v\ \texttt{else}\ [\![e_2]\!]_v \\
[\![\langle e_1, e_2 \rangle]\!]_v &= ([\![e_1]\!]_v,\ [\![e_2]\!]_v) \\
[\![[e_1, \ldots, e_n]]\!]_v &= [[\![e_1]\!]_v, \ldots, [\![e_n]\!]_v] \\
[\![\texttt{let}\ x{=}e_1\ \texttt{in}\ e_2]\!]_v &= [\![e_2]\!]_{v[x \mapsto [\![e_1]\!]_v]} \\
[\![\forall x.\ e]\!]_v &= \forall d,\ [\![e]\!]_{v[x \mapsto d]} \\
[\![\exists x.\ e]\!]_v &= \exists d,\ [\![e]\!]_{v[x \mapsto d]} \\
[\![\mathsf{ap}(e_1, e_2)]\!]_v &= [\![e_1]\!]_v\ ([\![e_2]\!]_v)
\end{aligned}
$$

We prove a series of lemmas (`interp_bool`, `interp_int`, `interp_nat`, `interp_array`, `interp_tuple`, etc.) that provide the witnesses of equalities needed by `cast` in the definition of `exp_rep`.

```
def interp_bool : typ_rep type_env dom_aux Typ._Bool = Bool := by
  simp[typ_rep]
```

We are able to prove the following theorem.

**Theorem 2.1**

If $\Gamma_\Delta \vdash e : t$, $v$ is a variable valuation for $\Gamma_\Delta$, then $[\![e]\!]_v \in [\![t]\!]$.

## 2.5 The Interpreter and Operational Semantics

While the denotational semantics gives a mathematical meaning to SST expressions, Verus also provides a concrete, executable interpreter that can evaluate SST expressions at compile time. This interpreter is used by proof strategies such as `assert(...) by (compute)` to discharge verification conditions by direct computation, thereby reducing the burden on the SMT solver.

In this section, we first describe the Verus interpreter and its motivation, then present an operational semantics that formalizes its behaviour as a big-step evaluation relation. Finally, we claim that we can establish the correctness of this interpreter by proving type preservation and semantic soundness with respect to the denotational semantics.

### 2.5.1 The Verus Interpreter

Verus includes an internal *interpreter* that can evaluate SST expressions at compile time [1]. This interpreter is used by the

```
assert(...) by (compute)
```

and

```
assert(...) by (compute_only)
```

proof strategies, which allow users to discharge proof obligations by direct computation rather than by SMT solving.

**Motivation.** Some proofs are "obvious" by simply computing on concrete values. For example, given a recursive function $\mathsf{pow}(base, exp)$ defining exponentiation, one would expect the assertion $\mathsf{pow}(2,8) = 256$ to be straightforward. However, Verus defaults to unrolling recursive definitions only once in the SMT encoding, to prevent the solver from diverging. Adjusting the unrolling depth is fragile: Z3 might not unroll the definitions enough,

or it might still refuse to simplify non-linear operations on statically known constants. The `assert(...) by (compute)` mechanism avoids this problem by running the interpreter over the asserted expression before encoding it to SMT. If the interpreter reduces the expression to `true`, the SMT solver receives a trivial obligation. `assert(...) by (compute_only)` requires the interpreter to reduce the expression completely to `true`; if it cannot, the assertion fails without invoking the solver at all, ensuring deterministic and stable proofs.

**Limitations.** An expression given to a proof by computation is interpreted in isolation from any surrounding context: free variables are treated symbolically and are not substituted with values from the enclosing scope. Furthermore, only spec-mode expressions can be interpreted. To prevent infinite interpretation loops which can arise even when the code is proven to terminate, Verus limits the time spent interpreting to the number of seconds.

### 2.5.2 Operational Semantics

The interpreter reduces a subset of SST expressions at compile time, providing a partial big-step operational semantics for SST expressions. We take the interpreter as a reference and list some of the evaluation rules.

We write $e \hookrightarrow_{F,w} e'$ to mean that expression $e$ evaluates to $e'$ under symbol table $w$ and global function context $F$. The symbol table $w$ maps free variables (de Bruijn indices) to expressions. The global context $F$ maps each function name to the SST expression representing its body. $e \hookrightarrow_{F,w} e'$ asserts that $e'$ is the fully reduced form of $e$, i.e., $e'$ is the expression obtained by applying as many evaluation steps as the interpreter can perform on $e$. We write $w[w']$ for extending $w$ with the bindings in $w'$.

We present representative evaluation rules. We omit $F$ and $w$ when they are clear from the context or when they are not used.

**Constants and Variables.** Constants are values and evaluate to themselves. A variable $x$ evaluates to its value in the symbol table if it's contained in the symbol table, and remains symbolic otherwise.

$$\frac{}{c \hookrightarrow c}\ \text{S-Const} \qquad\qquad \frac{x \in \mathsf{dom}(w)}{x \hookrightarrow_{F,w} w(x)}\ \text{S-Var}$$

**Boolean Connectives.** Boolean connectives use short-circuit evaluation. For instance, for conjunction, if the first operand evaluates to false, the result is false without evaluating the second operand. If the first operand evaluates to true, the result is the evaluation of the second operand. Disjunction and implication are analogous.

$$\frac{e_1 \hookrightarrow \mathsf{true} \qquad e_2 \hookrightarrow e_2'}{e_1 \wedge e_2 \hookrightarrow e_2'}\ \text{S-AND-T} \qquad \frac{e_1 \hookrightarrow \mathsf{false}}{e_1 \wedge e_2 \hookrightarrow \mathsf{false}}\ \text{S-AND-F}$$

**Equality.** Equality is decided syntactically on fully reduced expressions. The interpreter compares the results of reducing both expressions for syntactic equality. When the comparison result cannot be determined, the expression is returned in simplified form.

$$\frac{e_1 \hookrightarrow e_1' \qquad e_2 \hookrightarrow e_2' \qquad e_1' \equiv e_2'}{e_1 = e_2 \hookrightarrow \mathsf{true}}\ \text{S-EQ1} \qquad \frac{e_1 \hookrightarrow e_1' \qquad e_2 \hookrightarrow e_2' \qquad e_1' \not\equiv e_2'}{e_1 = e_2 \hookrightarrow \mathsf{false}}\ \text{S-EQ2}$$

$$\frac{e_1 \hookrightarrow e_1' \qquad e_2 \hookrightarrow e_2' \qquad e_1' \equiv e_2' \text{ undetermined}}{e_1 = e_2 \hookrightarrow e_1' = e_2'}\ \text{S-EQ3}$$

Here $e_1' \equiv e_2'$ denotes syntactic equality of fully reduced expressions, and "undetermined" means neither syntactic equality nor syntactic inequality can be established.

**Array Indexing.** Array indexing first evaluates the array expression and the index expression to values. If the index is within bounds, the whole expression evaluates to the element at the given position. Otherwise, the simplified array indexing expression is returned.

$$\frac{e_1 \hookrightarrow [v_1, \ldots, v_n] \qquad e_2 \hookrightarrow i \qquad 0 \le i < n}{\mathsf{array}\ e_1[e_2] \hookrightarrow v_i}\ \text{S-ARRIDX}_1$$

$$\frac{e_1 \hookrightarrow e_1' \qquad e_2 \hookrightarrow e_2'}{\mathsf{array}\ e_1[e_2] \hookrightarrow \mathsf{array}\ e_1'[e_2']}\ \text{S-ARRIDX}_2$$

**Tuples.** Both components are evaluated.

$$\frac{e_1 \hookrightarrow e_1' \qquad e_2 \hookrightarrow e_2'}{\langle e_1, e_2 \rangle \hookrightarrow \langle e_1', e_2' \rangle}\ \text{S-TUPLE}$$

**Let Binding.** The bound expression is evaluated and added to the valuation before evaluating the body.

$$\frac{e_1 \hookrightarrow_{F,w} e_1' \qquad e_2 \hookrightarrow_{F,w[x \mapsto e_1']} e_2'}{\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \hookrightarrow_{F,w} e_2'}\ \text{S-LET}$$

**Lambda.** A lambda expression evaluates to a closure that captures the current valuation. Note that the constructor closure only appears in the interpreter.

$$\frac{}{\lambda x.e \hookrightarrow_{F,w} \mathsf{closure}(\lambda x.e, w)}\ \text{S-LAM}$$

**Lambda Application.** Applying a closure substitutes the argument into the body under the captured valuation.

$$\frac{e_1 \hookrightarrow_{F,w} \mathsf{closure}(\lambda x.b, w') \qquad e_2 \hookrightarrow_{F,w} e_2' \qquad b \hookrightarrow_{F,w[w',x \mapsto e_2']} b'}{\mathsf{ap}(e_1, e_2) \hookrightarrow_{F,w} b'}\ \text{S-APP}$$

**Quantifiers.** Expressions with $\forall$ or $\exists$ at the top level are not reduced by the interpreter and are left unchanged.

Followng the approach in [18], we model $\hookrightarrow$ as a relation `eval` inductively.

```
inductive Eval: symbol_table → func_env → Exp → Exp → Prop where
  | const:
    ∀ (w: symbol_table) (f: func_env) (c: Const),
    Eval w f (.Const c) (.Const c)
  | var:
```

```
    ∀ (w: symbol_table) (f: func_env) (i: Nat) (e: Exp),
    Eval w f (.Var i) (w i)
  | unary_not_true :
    ∀ (w: symbol_table) (f : func_env) (arg : Exp),
    Eval w f arg (.Const (.Bool true)) →
    Eval w f (.Unary .Not arg) (.Const (.Bool false))
  ...
```

### 2.5.3 Substitution and Autosubst

The evaluation rules of the previous subsection rely on substitution. In the rule LET, evaluating the body $e_2$ under the extended valuation $w[x \mapsto e_1']$ amounts to substituting the value of the bound expression for the let-bound variable; in APP, the captured environment $w'$ of a closure is similarly threaded into the body. We use de Bruijn indices for variables to make it easier to implement capture-avoiding substitution.

We follow the approach of Autosubst [19, 20], which builds capture-avoiding substitution on top of $\sigma$-calculus [2]. In Autosubst, a *parallel substitution* is a function $\sigma : \mathbb{N} \to \mathtt{Exp}$ sending each de Bruijn index to an expression. Although the original Autosubst is implemented in Rocq, the underlying discipline is independent of the host proof assistant. We transcribe it into Lean 4 by hand.

**Renaming.** Following [19], we first define *renaming*, the special case in which the substitution $\xi$ has type $\mathbb{N} \to \mathbb{N}$. Renaming a variable simply applies $\xi$, i.e., substitutes each variable with another variable; for a non-binding constructor, $\xi$ is propagated to each subexpression; under a binder, $\xi$ is lifted by upr. upr $\xi$ $n$ is defined so that it leaves the indices $0, \ldots, n-1$ unchanged and maps every index $i \geq n$ to $\xi(i-n)+n$.

```
def upr (ξ : Nat → Nat) : Nat → (Nat → Nat)
  | 0     => ξ
  | n + 1 => snoc (fun i => (upr ξ n) i + 1) 0

def rename_exp (ξ : Nat → Nat) : Exp → Exp
```

```
  | .Const c            => .Const c
  | .Var x              => .Var (ξ x)
  | .Unary op arg       => .Unary op (rename_exp ξ arg)
  | .Let tys es body    => .Let tys (es.map (rename_exp ξ))
                             (rename_exp (upr ξ es.length) body)
  | .Lambda ty body     => .Lambda ty (rename_exp (upr ξ 1) body)
  | .Quant q ty body    => .Quant q ty (rename_exp (upr ξ 1) body)
  | .CallLambda lam arg => .CallLambda (rename_exp ξ lam)
                                       (rename_exp ξ arg)
  | ...
```

**Substitution.** A *substitution* is a function $\sigma : \mathbb{N} \to \mathsf{Exp}$. Its action on an expression, written subst_exp $\sigma$ $e$, replaces each variable index by $\sigma(i)$. At a binder we lift $\sigma$ by up $\sigma$ $n$, the substitution analogue of upr:

```
def up (σ : Nat → Exp) : Nat → (Nat → Exp)
  | 0     => σ
  | n + 1 => snoc (rename_exp Nat.succ comp (up σ n)) (.Var 0)

def subst_exp (σ : Nat → Exp) : Exp → Exp
  | .Const c            => .Const c
  | .Var x              => σ x
  | .Unary op arg       => .Unary op (subst_exp σ arg)
  | .Binary op a1 a2    => .Binary op (subst_exp σ a1) (subst_exp σ a2)
  | .If c b1 b2         => .If (subst_exp σ c)
                             (subst_exp σ b1) (subst_exp σ b2)
  | .Let tys es body    => .Let tys (es.map (subst_exp σ))
                             (subst_exp (up σ es.length) body)
  | .Lambda ty body     => .Lambda ty (subst_exp (up σ 1) body)
  | .Quant q ty body    => .Quant q ty (subst_exp (up σ 1) body)
  | .CallLambda lam arg => .CallLambda (subst_exp σ lam)
                                       (subst_exp σ arg)
  | ...
```

The single-variable substitution required by the operational rules LET and APP is recovered as $e[\,\mathsf{toSb}\,t\,]$ where the substitution $\mathsf{toSb}\,t = \texttt{snoc Exp.Var}\ t$ replaces index 0 by $t$ and shifts every other index down by one.

### 2.5.4 Interpreter Correctness

We establish two correctness properties for the interpreter: type preservation and semantic soundness.

The preservation theorem states that evaluation does not change the type of an expression. This property guarantees that the interpreter never produces ill-typed expressions from well-typed inputs, and that type information is invariant under evaluation.

> **Theorem 2.2 – Preservation**
> If $\Gamma \vdash e : t$, and $e \hookrightarrow e'$, then $\Gamma \vdash e' : t$.

*Proof.* The proof proceeds by induction on the derivation of $e \hookrightarrow e'$, with a case analysis on the evaluation rule used. For each rule, we rely on the inversion of the typing derivation for the source expression and the induction hypothesis when evaluating subexpressions. □

While preservation ensures that types are respected, soundness connects the operational semantics to the denotational semantics. It asserts that the interpreter's result is semantically equivalent to the original expression under the same variable valuation.

> **Theorem 2.3 – Soundness**
> If $e \hookrightarrow_{F,w} e'$, for each $x \in \mathsf{dom}(w)$, $[\![w(x)]\!]_v = v(x)$, then $[\![e]\!]_v = [\![e']\!]_v$.

The soundness theorem requires a compatibility condition between the symbol table $w$ (which maps de Bruijn indices to syntactic expressions) and the denotational valuation $v$ (which maps indices to semantic values). The condition states that for every variable $x$ in the symbol table, the denotation of $w(x)$ under $v$ equals $v(x)$ itself. Under this hypothesis, the theorem guarantees that evaluating an expression does not change its meaning.

*Proof.* The proof is by induction on the derivation of $e \hookrightarrow_{F,w} e'$. For rules that introduce new bindings (e.g., LET, LAM, APP), we extend the $w$ and $v$ appropriately and appeal to the induction hypothesis on the body. □

Together, preservation and soundness establish that the interpreter is both type-safe and semantically faithful: it transforms well-typed expressions into well-typed expressions of the same type without altering their denotation. This justifies the use of the interpreter in proof strategies such as `assert(...) by (compute)`, where reducing an expression to `true` suffices to conclude that its denotation is indeed true. The mechanization of the two theorems are left to future work.

# 3 AIR-AST: Assertion Intermediate Representation

The Assertion Intermediate Representation (AIR) is the last language-specific intermediate representation in the Verus pipeline before the final encoding into SMT-LIB. On the expression level, AIR closely mirrors SMT-LIB expressions. On the statement level, AIR is based on assert and assume statements, along with mutable updates to local variables. In the Verus codebase, AIR is defined in `verus/source/air/src/ast.rs`[1].

Rather than defining AIR's syntax and semantics from scratch, we formalize it as an instance of a generic *multi-sorted first-order logic* framework. Specifically, we build on the `Many-Sorted-Model-Theory` library [21], which provides, for any multi-sorted first-order signature, the notions of terms, formulas, structures (models), variable assignments, and satisfaction. We instantiate this framework with AIR's particular sorts and function symbols, obtaining a model-theoretic semantics "for free." This avoids the need to re-prove standard metatheoretic properties (e.g., coincidence lemmas for variable assignments) and ensures that our AIR semantics are consistent with the well-studied theory of multi-sorted first-order logic.

## 3.1 Syntax

This subsection presents the abstract syntax of AIR. We proceed in three steps. First, we define the sorts of AIR. Second, we describe the grammar of terms and formulas, following the standard pattern of multi-sorted first-order logic. Third, we enumerate the function symbols and relation symbols that form the signature of the language.

### 3.1.1 Sorts

The sorts in AIR are defined by the inductive type `AirSorts`:

```
inductive AirSorts where
  | Bool
```

[1] https://github.com/verus-lang/verus/blob/main/source/air/src/ast.rs

```
| Int
| Fun
| Named (name : String)
| Bitvec (width : UInt32)
```

The `Bool` and `Int` sorts correspond directly to the SMT-LIB sorts of the same name. The `Fun` sort contains function as elements. For example, in AIR, arrays are modeled as functions from indices to values. The `Named` sort constructor provides a family of uninterpreted sorts parameterized by a string; the most important instances are `Poly` and `TYPE`. `Poly` is used to encode polymorphism in the monomorphic SMT-LIB logic. Specifically, when an SST expression has a polymorphic type, the translation maps it to an AIR term of sort `Poly`. When a polymorphic function is applied at a concrete type, the translation inserts coercion functions to turn concrete values into terms of sort `Poly`. During the translation, each SST type is mapped to a constant of sort `TYPE` (e.g., `BOOL`, `INT`; or `T`, an uninterpreted sort). Finally, `Bitvec` $w$ represents $w$-bit bitvectors for bit-level reasoning.

### 3.1.2 Terms and Formulas

The syntax of AIR terms and formulas follows the standard pattern of multi-sorted first-order logic. Terms are built from variables, constants, and function applications; formulas are built from equality, logical connectives, and quantifiers.

**Definition 3.1 – Syntax of AIR Terms and Formulas**

The abstract syntax of AIR terms and formulas is given by the following grammar:

$$\begin{aligned} \text{Terms} \quad & e \quad ::= x \mid c \mid f(e_1, \ldots, e_n) \\ \text{Formulas} \quad & \varphi \quad ::= \bot \mid e_1 = e_2 \mid R(e_1, \ldots, e_n) \mid \varphi_1 \to \varphi_2 \mid \forall x : S.\, \varphi(x) \end{aligned}$$

Here $x$ ranges over variables, $c$ over constants (nullary function symbols), $f$ over function symbols, $R$ over relation symbols, and $S$ over sorts.

We are able to define other logical connectives with $\bot$ and $\to$.

```
abbrev bnot (p : TransFormula) : TransFormula := imp p falsum
abbrev band (p q : TransFormula) : TransFormula := bnot (imp p (bnot q))
abbrev biff (p q : TransFormula) : TransFormula := band (imp p q) (imp q p)
...
```

The `Many-Sorted-Model-Theory` library is designed such that users only need to provide constants (nullary functions), function symbols, and relations. The library then constructs the full language, including all well-sorted terms and formulas, and provides generic definitions of variable assignments, substitutions, and the satisfaction relation. The complete multi-sorted first-order language is assembled by packaging the function and relation symbols into an `MSLanguage` record.

```
def air_ast : MSLanguage AirSorts := {
  Functions := airFunc
  Relations := airReln
}
```

`airFunc` will be introduced in §3.1.3, and `airFunc` will be introduced in §3.1.4.

### 3.1.3 Function symbols

The function symbols of the AIR language are collected in the inductive type `airFunc`, indexed by an input sort list and an output sort. We organize them into several groups.

```
inductive airFunc : List AirSorts → AirSorts → Type
```

**Constants.** Nullary function symbols include integer literals `Nat` *i* (where *i* is a string representation of a non-negative integer), and bitvector literals.

```
| Nat : (i : String) → airFunc [] Int
```

```
| BitVector : (bits : List Nat) → (width: UInt32) → airFunc [] (Bitvec
    width)
```

**Type tags.** The nullary symbols `BOOL`, `INT`, `NAT`, `CHAR`, `USIZE`, `ISIZE` and the unary symbols `UINT`, `SINT`, `FLOAT` produce values of sort `TYPE`. Also, the type parameter constructor `TypParamConst` takes a type parameter (i.e., a string) and produces a value of sort `TYPE`. They are used in `has_type` axioms and will be further discussed in §4.3.2.

```
| BOOL : airFunc [] TYPE
| INT : airFunc [] TYPE
| TypParamConst : (name : String) → airFunc [] TYPE
...
```

***n*-ary operations.** AIR inherits from SMT-LIB the convention that addition, subtraction, and multiplication are *n*-ary operations. We model these as a family of function symbols parameterised by the arity *n*:

```
| Add : (n : Nat) → airFunc (List.replicate n Int)  Int
...
```

The use of `List.replicate` ensures that the sort list has exactly *n* copies of the appropriate sort.

**Binary arithmetic.** Binary arithmetic includes SMT built-in operations: `Add`, `Sub`, `Mul`, `EuclideanDiv` and `EuclideanMod`. In addition, we introduce a parallel set of *uninterpreted* arithmetic symbols `ADD`, `SUB`, `MUL`, `EucDIV`, and `EucMOD`. The interpretation of these uninterpreted symbols is constrained by axioms (see §4.2 below). This two-tier design mirrors the actual Verus pipeline, where the SMT encoding uses uninterpreted functions linked to built-in operations via axioms.

The reason for this duplication is pragmatic. When the Z3 option `smt.arith.nl=false` is enabled, the solver sometimes fails to prove obvious formulas that happen to contain

multiplication, division, or modulus operations (for example, failing to prove $P \Rightarrow P$). To improve stability, Verus wraps non-linear occurrences of these operators inside uninterpreted functions.

```
| Mul : airFunc [Int, Int] Int
| EuclideanDiv : airFunc [Int, Int] Int
| MUL : airFunc [Int, Int] Int
| EucDIV : airFunc [Int, Int] Int
...
```

**Polymorphic coercions.** Since SMT-LIB is monomorphic, Verus encodes polymorphic values by boxing them into the `Poly` sort. The function symbols `ofI`, `ofB`, `toI`, and `toB` provide boxing and unboxing between the concrete sorts (`Int`, `Bool`) and `Poly`.

$$\texttt{ofB} : \texttt{Bool} \to \texttt{Poly} \qquad \texttt{toB} : \texttt{Poly} \to \texttt{Bool}$$
$$\texttt{ofI} : \texttt{Int} \to \texttt{Poly} \qquad \texttt{toI} : \texttt{Poly} \to \texttt{Int}$$

```
| ofI : airFunc [Int] Poly
| toI : airFunc [Poly] Int
...
```

**Other functions.** The remaining function symbols are introduced to support specific constructs of the SST translation. Two notable examples are `ARRAY` and `ARRAY_INDEX`, used for encoding array types and array indexing respectively. `ARRAY` is a type constructor: given the type tags of the element type and the length, it produces the type tag for the corresponding array type, used in `has_type` axioms about arrays. `ARRAY_INDEX` takes a `Fun`-sorted array representation and a `Poly`-sorted index, and returns the `Poly`-sorted element at that position. They will be further introduced when we describe the translation in §4.1.2.

```
| ARRAY : airFunc [TYPE, TYPE] TYPE
| ARRAY_INDEX : airFunc [Fun, Poly] Poly
```

```
...
```

### 3.1.4 Relation Symbols

The relation symbols of the AIR language are collected in the inductive type `airReln`, indexed by a sort list. They are predicates that take a tuple of arguments and return a formula rather than a value. Relation symbols include integer comparisons $\leq$, $\geq$, $<$ and $>$. The relation `HAS_TYPE` takes arguments of sort `Poly` and `TYPE`, expressing that a polymorphic value inhabits the type denoted by a given type tag. It is the counterpart of the `has_type` predicate emitted by Verus, and is used in the axioms generated during the translation of polymorphic expressions (§4.3.2).

```
inductive airReln : List AirSorts → Type
  | Le : airReln [Int, Int]
  | Ge : airReln [Int, Int]
  | Lt : airReln [Int, Int]
  | Gt : airReln [Int, Int]
  | HAS_TYPE : airReln [Poly, TYPE]
```

## 3.2 Semantics

Having defined AIR's syntax, we now turn to its semantics. Our goal is to give a model-theoretic interpretation of AIR terms and formulas leveraging the constructions of the `Many-Sorted-Model-Theory` library, so that we can later state and prove the correctness of the SST-to-AIR translation as a relationship between SST denotations and AIR satisfaction. We proceed in two steps. §3.2.1 introduces the notion of a *model* for AIR: an assignment of a Lean type to each sort, together with concrete interpretations for every function and relation symbol of the language. §3.2.2 then defines, on top of any model, the realization of terms and formulas and the standard satisfaction relation $\models$, all of which are inherited from the library.

### 3.2.1 Models

A model of the AIR language is a multi-sorted structure that assigns a type to each sort and an interpretation to each function symbol and relation symbol. The `Many-Sorted-Model-Theory` library formalizes this notion as the type class `MSStructure`, which is parameterized by a language $L$ : `MSLanguage Sorts` and a carrier $C$ : `Sorts` $\to$ `Type`:

```
class MSStructure {Sorts} (L : MSLanguage Sorts) (C : Sorts → Type w) where
    funMap : ∀ {σ t}, L.Functions σ t → SortedTuple σ C → C t := by
        exact fun {σ} => fun {t} => isEmptyElim
    RelMap : ∀ {σ}, L.Relations σ → SortedTuple σ C → Prop := by
        exact fun {σ} => isEmptyElim
```

An `MSStructure` instance provides two pieces of data. The carrier $C$ maps each sort to a Lean type. The function `funMap` assigns to every function symbol $f$ : $L$.`Functions` $\overline{\sigma}$ $t$ an actual Lean function taking a sorted tuple of arguments (one value from $C\ \sigma_i$ for each input sort) and returning a value of type $C\ t$. Similarly, `RelMap` assigns to every relation symbol a Lean predicate over sorted tuples of arguments. Concretely, an `MSStructure` is a model in the model-theoretic sense: it tells us what each sort and each symbol of the language mean.

We now instantiate this framework with the AIR language. We've already defined the language in §3.1. Here, we define a parameterized carrier assignment:

```
abbrev AirCarrier (P T F : Type) : AirSorts → Type
  | Bool      => Bool
  | Int       => Int
  | Named "Poly"  => P
  | Named "TYPE"  => T
  | Fun       => F
  | Bitvec w  => BitVec w.toNat
```

where $P$, $T$, and $F$ are parameters representing the interpretation of the `Poly`, `TYPE`, and `Fun` sorts, respectively. The `Bool` and `Int` sorts are always interpreted as Lean's `Bool` and `Int`.

To constrain the interpretation of function and relation symbols, we define a type class `AirMod P T F` extending the generic `MSStructure` with requirements that pin down the meaning of some functions and relations.

```
class AirMod (P T F : Type) extends air_ast.MSStructure (AirCarrier P T F)
    where ...
```

Since `funMap` receives its arguments as a sorted tuple *xs*, we use helper projections $\texttt{eval}_{nk}$ defined in the library to denote the $k$-th element of a tuple of length $n$. For example, $xs.\texttt{eval}_{21}$ and $xs.\texttt{eval}_{22}$ are the first and second elements of a two-element tuple *xs*. Each projection returns a value at the appropriate sort determined by the function symbol's input sort list.

Some illustrative constraints are listed:

```
funMap_nat   : ∀ (i : String) xs, funMap (airFunc.Nat i) xs = i.toInt?.getD 0
```

In the AIR language, natural number literals are represented as strings. This constraint specifies that `Nat` $i$ is interpreted as the integer value obtained by parsing $i$, defaulting to 0 if parsing fails.

```
funMap_add   : ∀ (n : Nat) xs, funMap (airFunc.Add n) xs = foldInts (· + ·) 0
    xs
```

The $n$-ary addition symbol `Add` $n$ is interpreted as the sum of all $n$ arguments, computed by folding integer addition over the tuple with initial value 0.

```
funMap_euclideanDiv : ∀ xs, xs.eval22 ≠ 0 → funMap airFunc.EuclideanDiv xs =
     xs.eval21 / xs.eval22
```

Division and modulus are specified only for non-zero denominators; when the denominator is zero, the result is unconstrained, matching the SMT-LIB semantics of division by zero as an unspecified function.

Operations on `Poly`, `TYPE`, and `Fun` are left abstract. Their meaning is constrained only by the axioms.

The interpretation of relation symbols is similarly fixed by the `AirMod` class, but through the `RelMap` component inherited from `MSStructure` rather than `funMap`. For each comparison, an axiom equates the relation's interpretation with the corresponding Lean predicate on `Int`. For example, `Le` is interpreted as $\leq$, and `Ge` as $\geq$.

```
relMap_le : ∀ (xs : SortedTuple [AirSorts.Int, AirSorts.Int] (AirCarrier P T
    F)), RelMap airReln.Le xs ↔ xs.eval₂₁ ≤ xs.eval₂₂
relMap_ge : ∀ (xs : SortedTuple [AirSorts.Int, AirSorts.Int] (AirCarrier P T
    F)), RelMap airReln.Ge xs ↔ xs.eval₂₁ ≥ xs.eval₂₂
```

### 3.2.2 Satisfaction

With the carrier and interpretations of some function and relation symbols, we can now define the semantics of AIR terms and formulas. The `Many-Sorted-Model-Theory` library provides three layers of evaluation, which we summarize in turn.

**Variable assignment.** A variable assignment $u$ is a function mapping each variable to a value in the interpretation of its sort.

**Term realization.** A term $t$: `L.Term` $\alpha$ $s$ of sort $s$ has free variables indexed by $\alpha$. Given a variable assignment $u$, `realize` $u$ $t$ : $C$ $t$ produces the value of $t$ under $u$ by structural recursion:

```
def realize {t : Sorts} (u : α →ₛ C) : L.Term α t → C t
  | var t k       => u t k
  | func _ t f ts => funMap f (fun i => (ts i).realize u)
```

The interpretation of variables is directly given by $u$. The interpretation of a function application $f(e_1,\ldots,e_n)$ is obtained by first interpreting each argument and then applying the interpretation of $f$, i.e., `funMap f`, to them.

**Definition 3.2 – Term Interpretation**

Let $[e]^t_{M,u}$ be the interpretation of term $e : t$ in model $M$ under variable assignment $u$. $f_M$ is the interpretation of $f$ in the model $M$. $[e]^t_u$ is defined as:

$$\begin{aligned} [x]^t_{M,u} &= u(x) \\ [f(e_0,\ldots,e_n)]^t_{M,v} &= f_M([e_0]^t_{M,u},\ldots,[e_n]^t_{M,u}) \end{aligned}$$

**Formula realisation.** Analogously, a formula $\varphi$ is evaluated under a variable assignment to produce a proposition in Lean's `Prop`.

```
nonrec def Realize (φ : L.Formula α) (u : famMap α C) : Prop := ...
```

$M, u \vDash \varphi$ means $\varphi$ is true in the model $M$ under the assignment $u$.

**Definition 3.3 – Satisfaction Relation**

The satisfaction relation $\models$ is defined by structural recursion on $\varphi$:

$$\begin{aligned} M,u &\not\models \bot \\ M,u \models t_1 = t_2 &\iff [t_1]^t_{M,u} = [t_2]^t_{M,u} \\ M,u \models R(t_1,\ldots,t_n) &\iff R_M\left([t_1]^t_{M,u},\ldots,[t_n]^t_{M,u}\right) \\ M,u \models \varphi_1 \to \varphi_2 &\iff \text{if } \left(M,u \models \varphi_1\right) \text{ then } \left(M,u \models \varphi_2\right) \\ M,u \models \forall x.\,\varphi &\iff \forall d \in C(s),\ M,u[x \mapsto d] \models \varphi \end{aligned}$$

where $t_1, t_2$ are terms of sort $t$, and $R_M$ is the interpretation of the relation symbol $R$ in the model $M$.

For a set of formulas $T$,

$$M, w \vDash T \iff \forall \varphi \in T, M, w \vDash \varphi.$$

# 4 Translating SST Expressions to AIR-AST

The SST is the last Verus intermediate representation that still retains rich, Rust-specific type and expression information. AIR is essentially a multi-sorted first-order logic with a monomorphic type system, designed to be a low-level IR for SMT encoding. The translation from SST to AIR is therefore complex. It must map a polymorphic, high-level language into a monomorphic logic, while preserving the semantics of expressions.

In the Verus codebase, the translation of expressions consists of two parts: the translation of types and the translation of expressions. In Verus codebase, the function `typ_to_air`[1] takes an SST type and translates it to an AIR sort. `exp_to_expr`[2] takes an SST expression and translates it to an AIR expression.

## 4.1 Translation Overview

The translation from SST to AIR is the final compilation step in Verus before SMT encoding. Its job is twofold: it must produce, for each SST expression, a corresponding AIR term or formula, and it must accumulate axioms that capture the background theory the translated result sits in.

Before diving into the details, we provide an overview of what happens in the translation process by first investigating the translation result. Roughly, the result is composed of two parts: background axioms and the translated proof obligations (the AIR formulas corresponding to SST assertions). Background axioms consists of two layers. The first layer is a fixed set of *prelude axioms* (§4.2). The second layer consists of *translation-generated axioms* that are accumulated as `trans_exp` traverses the expression. We model this by threading an axiom set through the translation: each call to the translation function takes the current set as input and returns a possibly-augmented set together with the proof obligation. For instance, consider the translation of variables at polymorphic type. In SST, when a variable $v$ has type $\alpha$ for some type parameter $\alpha$, the translation adds the axiom $\texttt{has\_type}(v, \alpha)$ to `aenv` to constrain

[1] https://github.com/verus-lang/verus/blob/main/source/vir/src/sst_to_air.rs#L154
[2] https://github.com/verus-lang/verus/blob/main/source/vir/src/sst_to_air.rs#L882

the Poly-sorted value to inhabit the type denoted by $\alpha$. After the translation finishes, the SMT query sent to Z3 consists of all the axioms and the generated proof obligations.

### 4.1.1 Translation of Types

We model the translation function typ_to_air in Verus by the Lean function trans_typ. It maps each SST type to its corresponding AIR sort. Concrete types are mapped to their natural AIR counterparts: Bool to Bool, and all numeric types (Int, Nat, floats, etc.) to Int. AIR uses a single Int sort and relies on axioms to distinguish the various numeric types, such as Nat and Float. Array types and spec-mode function types are both mapped to the Fun sort. For arrays, they are modeled as functions from indices to elements. Type parameters are mapped to the Poly sort. This will be further discussed in §4.3.2. In the rest of the thesis, we will write $\bar{t}$ to be the translation of SST type $t$.

```
def trans_typ (t : sst.Typ): AirSorts :=
  match t with
  | ._Bool => Bool
  | .Int _ | .Float _ => Int
  | .Array _ => Fun
  | .TypParam (i : String)  => Poly
  | .SpecFn _ _ => Fun
  | .FnDef _ _ => FnDef
  ...
```

### 4.1.2 Translation of Expressions

The core of the SST-to-AIR expression translation is the function exp_to_expr, which translates a well-typed SST expression into its AIR representation. We model it by trans_exp. Its definition is as follows:

```
def trans_exp {Γ: context} (e : sst.Exp) (t : sst.Typ) (hty : Γ ⊢ e : t )(
    aenv : TransAxioms) : (TransTerm ⊕ TransFormula) × TransAxioms :=
  match e with
```

```
...
```

The function takes four inputs: (i) the SST expression $e$ to be translated; (ii) the type $t$ of $e$; (iii) a proof that $e$ has type $t$ in context $\Gamma$; and (iv) a set of axioms accumulated so far. It returns a pair consisting of: (i) a sum type $\texttt{TransTerm} \oplus \texttt{TransFormula}$, indicating that the translation yields either an AIR term or an AIR formula (this is because boolean-valued SST expressions are translated to formulas, while other expressions are translated to terms); and (ii) an *updated* axiom set, which may contain new axioms generated during the translation of $e$.

We now introduce some auxiliary definitions before presenting the translation itself.

We introduce a family of *type invariants* $\mathsf{inv}_t(\cdot)$ for each SST type $t$. These invariants capture the additional constraints that values of type $t$ must satisfy in AIR beyond the sort that is the translation result of $t$.

**Definition 4.1 –** $\mathsf{inv}_t(\cdot)$

Let $t$ be an SST type, $x$ be a free variable of type $t$. The type invariant $\mathsf{inv}_t(x)$ is defined as follows (only contains some cases):

$$\begin{aligned} \mathsf{inv}_{\texttt{Bool}}(x) &= \top \\ \mathsf{inv}_{\texttt{Int}}(x) &= \top \\ \mathsf{inv}_{\texttt{Nat}}(x) &= x \geq 0 \\ \mathsf{inv}_{\alpha}(x) &= \mathsf{has\_type}(x, \alpha) \end{aligned}$$

For base types `Bool` and `Int`, their translated sorts already faithfully represent their inhabitants, so the invariant is simply $\top$ (true). `Nat` is translated to `Int`, which doesn't capture the fact that the inhabitants of `Nat` are non-negative. Hence, the invariant enforces that the integer value is non-negative. For a type parameter $\alpha$, the invariant uses the `has_type` predicate to constrain the `Poly`-sorted value $x$ to be tagged with the type represented by $\alpha$.

**Definition 4.2 – Translation of Expressions**

The translation function $\overline{\cdot}$ maps an SST expression $e$ to an AIR term or formula. It is

defined by structural recursion on $e$ as follows:

$$
\begin{aligned}
\overline{b_{\texttt{Bool}}} &= b \\
\overline{i_{\texttt{Int}}} &= i \\
\overline{n_{\texttt{Nat}}} &= n \text{ as } \texttt{Int} \\
\overline{c_{\texttt{Char}}} &= \text{unicode}(c) \\
\overline{x} &= x \\
\overline{e_1 + e_2} &= \texttt{ADD}(\overline{e_1}, \overline{e_2}) \\
\overline{\neg e} &= \neg \overline{e} \\
\overline{e_1 \otimes e_2} &= \overline{e_1} \otimes \overline{e_2}, \qquad \otimes \in \{\wedge, \vee, \rightarrow\} \\
\overline{[e_1, \ldots, e_n]} &= \texttt{ARRAY\_NEW}(n, \overline{e_1}, \ldots, \overline{e_n}) \\
\overline{\texttt{array}\ a[i]} &= \texttt{ARRAY\_INDEX}(\overline{a}, \overline{i}) \\
\overline{\forall x : t.\, e} &= \forall x : \bar{t}.\ \left(\mathsf{inv}_t(x) \rightarrow \overline{e}\right) \\
\overline{\exists x : t.\, e} &= \exists x : \bar{t}.\ \left(\mathsf{inv}_t(x) \wedge \overline{e}\right)
\end{aligned}
$$

Boolean literals $b$ of type `Bool` and integer literals $i$ of type `Int` map to AIR terms of sorts `Bool` and `Int` respectively. Natural number literals $n$ are also represented as `Int`-sorted terms, since AIR has no dedicated natural-number sort. Character literals are encoded as integers via their Unicode code. An SST variable at de Bruijn index $i$ becomes an AIR variable at the same index. If the variable has a polymorphic type in SST, then some axioms will be added. This will be discussed in §4.3.2.

Arithmetic operations on integer-typed expressions are translated to calls to the uninterpreted arithmetic symbols `ADD`, `SUB`, `MUL`, `EucDIV`, `EucMOD`, rather than directly to the built-in SMT-LIB operators.

Array literals are wrapped in a call to the function `ARRAY_NEW`. This wrapping is necessary because AIR arrays are modelled as `Fun`-sorted total functions with no built-in notion of length. `ARRAY_NEW` takes the array length together with array elements as arguments. Array indexing is translated to the application of `ARRAY_INDEX` on the translation result of the array and the index. `ARRAY_NEW` and `ARRAY_INDEX` are characterized by some prelude axioms, ensuring that arrays are characterized correctly in AIR.

Universal and existential quantifiers are translated by combining an AIR quantifier with the type invariant of the bound variable. For $\forall x : t.\, e$, the translation produces $\forall x : \bar{t}.\ (\mathsf{inv}_t(x) \to \bar{e})$: the quantifier ranges over the potentially larger AIR sort (e.g., `Int` for `Nat`), and the invariant restricts the scope of $x$ further. For $\exists x : t.e$, the translation produces $\exists x : \bar{t}.(\mathsf{inv}_t(x) \wedge \bar{e})$, indicating that there is an $x$ satisfying both the invariant and the body.

**Theorem 4.1**

If $\Gamma \vdash e : t$ in SST, then:

1. if $t$ is `Bool`, then either $\bar{e}$ has sort $\bar{t}$, or $\bar{e}$ is a formula;
2. otherwise, $\bar{e}$ has sort $\bar{t}$.

*Proof.* We proceed by structural induction on the typing derivation $\Gamma \vdash e : t$.

For the base cases:

- If $e$ is a boolean constant $b_{\texttt{Bool}}$, then $\bar{e} = b$, which is a `Bool`-sorted term. If $e$ is an integer constant $i_{\texttt{Int}}$ or a natural constant $n_{\texttt{Nat}}$, then $\bar{e}$ is an `Int`-sorted term.
- If $e$ is a variable $x$, then $\bar{e} = x$, which inherits the sort of its type according to $\bar{t}$.

For the inductive cases:

- For $\neg e$ and $e_1 \otimes e_2$ ($\otimes \in \{\wedge, \vee, \to\}$): $t = \texttt{Bool}$ in these cases, and the translation produces a formula.
- For array literals and array indexing, the translations `ARRAY_NEW` and `ARRAY_INDEX` are defined to return the appropriate sort (`Fun` for arrays, the element sort for indexing).
- For quantifiers $\forall x : t.\, e$ and $\exists x : t.\, e$, the translation produces a formula, which is consistent with $t = \texttt{Bool}$.

□

We formalize the correctness of the translation in Lean. Instead of proving a single monolithic theorem, we decompose the statement into a collection of lemmas. Each lemma corresponds to one SST typing judgment.

For example, the translation of boolean constants always produces a formula: $\overline{\mathsf{true}}$ becomes the true proposition, and $\overline{\mathsf{false}}$ becomes false. This is captured by the following theorem:

```
theorem trans_exp_bool_const (b : Bool)
    (hty : Γ ⊢ .Const (.Bool b) : ._Bool) :
    (trans_exp (.Const (.Bool b)) ._Bool hty aenv).1 =
      .inr (if b then truth else falsum) := by ...
```

Similarly, the translation of a variable $x$ of type $t$ is an AIR term of sort $\bar{t}$. The theorem formalizing this case is:

```
theorem trans_exp_var (idx : Nat) (t : Typ)
    (hty : Γ ⊢ .Var idx : t) :
    (trans_exp (.Var idx) t hty aenv).1 =
      .inl ⟨ trans_typ t, Term.var (trans_typ t) idx ⟩ := by ...
```

## 4.2 Prelude Axioms

The `prelude.rs` file in Verus codebase[1] emits a set of axioms that every SMT query assumes. We formalize a subset of these axioms as AIR sentences.

**Arithmetic axioms.** For each uninterpreted arithmetic symbol, a universally quantified axiom equates it with the corresponding built-in operation:

$$\forall x\, y : \mathsf{Int},\ \mathsf{ADD}(x,y) = \mathsf{Add}(x,y)$$

and analogously for SUB/Sub, MUL/Mul, EucDIV/EuclideanDiv, and EucMOD/EuclideanMod. Additional axioms constrain the range of unsigned multiplication, division, and modulus to ensure that non-negative inputs produce non-negative outputs within the expected bounds.

[1] https://github.com/verus-lang/verus/blob/main/source/vir/src/prelude.rs

For example,

$$\forall x\, y : \mathsf{Int},\ (x \geq 0 \wedge y \geq 0) \rightarrow \mathtt{MUL}(x, y) \geq 0$$

**Poly round-trip axioms.** Two axioms assert that the `ofB` is the left-inverse of `toB`, and similarly for `ofI` and `toI`:

$$\forall x : \mathsf{Bool},\ x = \mathtt{toB}(\mathtt{ofB}(x)) \qquad \forall x : \mathsf{Int},\ x = \mathtt{toI}(\mathtt{ofI}(x))$$

Conversely, two additional axioms provide the right-inverse direction for `Poly`-sorted values: boxing after unboxing recovers the original polymorphic value, provided the value has the expected type:

$$\forall x : \mathtt{Poly}, \mathtt{has\_type}\ (x, \mathtt{BOOL}) \rightarrow x = \mathtt{ofB}(\mathtt{toB}(x))$$
$$\forall x : \mathtt{Poly}, \mathtt{has\_type}\ (x, \mathtt{INT}) \rightarrow x = \mathtt{ofI}(\mathtt{toI}(x))$$

There are also axioms guaranteeing that a value boxed from a concrete type is always tagged with the corresponding type constant, which is necessary for the `has_type` constraints on polymorphic values to be satisfied.

$$\forall x : \mathsf{Bool},\ x = \mathtt{has\_type}\ (\mathtt{ofB}(x), \mathtt{BOOL}) \qquad \forall x : \mathsf{Int},\ x = \mathtt{has\_type}\ (\mathtt{ofI}(x), \mathtt{INT})$$

The complete set of prelude axioms is collected in `preludeAxioms`.

## 4.3 Monomorphization

SST inherits Rust's parametric polymorphism: functions and data types can be parameterized by type variables, and callsites supply concrete types. AIR, however, targets the monomorphic logic of SMT-LIB, which has no notion of type parameters. The SST-to-AIR translation must therefore encode polymorphism into a monomorphic setting. Verus adopts the *type-argument* approach of [12], in which type parameters are explicitly passed as values to functions. We illustrate the encoding with a running example and then describe the translation of polymorphic functions, and polymorphic function applications.

### 4.3.1 Running example

Consider the following Verus program, which defines a polymorphic identity function and a monomorphic wrapper:

```
spec fn spec_identity<T>(v: T) -> T {
    v
}

spec fn concrete_identity(u: bool) -> (result: bool) {
    spec_identity(u)
}
```

`spec_identity` takes an argument of any type and returns itself. `concrete_identity` specializes `spec_identity` by calling it on an argument of type `bool`. At the SST level, `spec_identity` carries a type parameter list `typ_params = [T]`, and its argument has type `T`. In contrast, `concrete_identity` has an empty type parameter list, and its argument has type `bool`.

### 4.3.2 Translating Polymorphic Functions

The translation from SST to AIR maps each SST type to an AIR sort. A type parameter `T` is mapped to the `Poly` sort. Consequently, when `spec_identity` is translated to AIR, its argument `v` and its return value `result` are both declared with sort `Poly`:

```
(declare-const result! Poly)
(declare-const v! Poly)
```

The translation also introduces a `Type`-sorted variable for the type parameter `T` and emits a `has_type` axiom. In our example, the generated AIR declares:

```
(declare-const T& Type)
(axiom (has_type v! T&))
```

The axiom `has_type(v!, T&)` constrains the polymorphic value `v!` to inhabit the type denoted by `T&`.

A polymorphic SST function $f$ with $m$ type parameters and $n$ arguments is then declared in

AIR as an uninterpreted function taking $m$ additional arguments of sorts `TYPE` (one per type parameter), followed by $n$ arguments, among which the polymorphically typed ones have sort `Poly`. For `spec_identity` with one type parameter and one argument, the declaration is:

```
(declare-fun poly!spec_identity.? (Dcr Type Poly) Poly)
```

The `Dcr` is not within the scope of this thesis, so we omit it for now. The translated function `poly!spec_identity.?` takes an argument `T` of sort `Type`, and an argument `v` of sort `Poly`, which satisfies the axiom `has_type v T`. And it returns a result of sort `Poly`.

### 4.3.3 Translating Polymorphic Function Applications

The translated version of `concrete_identity` is declared as a function that takes a `Poly`-sorted argument and returns a `Bool`:

```
(declare-fun poly_spec!concrete_identity.? (Poly) Bool)
```

Its defining axiom equates `poly_spec!concrete_identity.?` to the application of `poly!spec_identity.?` with the concrete type argument `BOOL`, followed by unboxing the result back to `Bool`:

$\forall$`u:Poly`, `poly_spec!concrete_identity.? u = ofB (poly!spec_identity.? $ BOOL u)`

## 4.4 Soundness of the Translation

This subsection states and sketches a proof of the central correctness result of the thesis: the SST-to-AIR translation of expressions is sound, in the sense that whenever the SMT solver concludes that in every model of the generated axioms, the AIR translations of two SST expressions are semantically equivalent, the original SST expressions denote the same value.

Stating this theorem precisely requires three ingredients, which we introduce in turn before presenting the theorem. First, since the translation produces either an AIR term or an AIR formula depending on the input, we need a uniform notion of *equivalence between translation results* that subsumes both cases (§4.4.1). Second, the AIR carrier for the `Poly` sort

must be chosen carefully so that polymorphic SST values have appropriate AIR representations. We describe this in §4.4.2. Third, since the SST denotational semantics takes an SST variable valuation while AIR satisfaction takes an AIR variable assignment, we need a *coherence relation* that connects the two (§4.4.3). Once these pieces are in place, §4.4.4 states the soundness theorem and sketches its proof.

### 4.4.1 Equivalence between Translation Results

The goal of the translation is to faithfully encode SST expressions into AIR while preserving their meaning. To state this formally, we need a notion of equivalence between translation results.

Recall that the translation function $\overline{\cdot}$ may return either an AIR term or an AIR formula. To compare two translation results, we introduce a function $\approx$ that takes two AIR terms and return a formula. Intuitively, two terms are equivalent if they are equal; two formulas are equivalent if each one implies the other; a term $t$ and a formula $\varphi$ are considered equivalent when $t = \texttt{true}$ holds if and only if $\varphi$ holds.

> **Definition 4.3 – Equivalence between Translation Results**
>
> Let $e_1, e_2$ be two SST expressions. The definition of $\approx$ is as follows:
>
> 1. If $\overline{e_1}$ and $\overline{e_2}$ are both formulas, then $\overline{e_1} \approx \overline{e_2}$ is defined as $\overline{e_1} \leftrightarrow \overline{e_2}$;
> 2. If $\overline{e_1}$ and $\overline{e_2}$ are both terms, then $\overline{e_1} \approx \overline{e_2}$ is defined as $\overline{e_1} = \overline{e_2}$;
> 3. If $\overline{e_1}$ is a term of sort `BOOL`, and $\overline{e_2}$ is a formula, then $\overline{e_1} \approx \overline{e_2}$ is defined as $(\overline{e_1} = \texttt{true}) \leftrightarrow \overline{e_2}$;
> 4. If $\overline{e_1}$ is a formula, and $\overline{e_2}$ is a term of sort `BOOL`, then $\overline{e_1} \approx \overline{e_2}$ is defined as $\overline{e_1} \leftrightarrow (\overline{e_2} = \texttt{true})$;
> 5. Otherwise, $\overline{e_1} \approx \overline{e_2}$ is defined as $\bot$.

The definition of $\approx$ is formalized as `AirResultEquiv`, which pattern-matches on the two results and constructs the appropriate AIR formula.

```
def AirResultEquiv {Γ : context} {t : Typ}
    (e₁ e₂ : sst.Exp) (hty₁ : Γ ⊢ e₁ : t) (hty₂ : Γ ⊢ e₂ : t) : TransFormula
    :=
  match (trans_exp e₁ t hty₁ preludeAxioms).1, (trans_exp e₂ t hty₂
    preludeAxioms).1 with
  | .inr φ₁, .inr φ₂ =>
      -- Both are formulas: logical equivalence
      biff φ₁ φ₂
  | .inl ⟨ s₁, tm₁ ⟩, .inl ⟨ s₂, tm₂ ⟩ =>
      -- Both are terms: value equality
      have hty : s₁ = s₂ := by sorry
      Term.equal tm₁ (hty ▸ tm₂)
  | .inl ⟨ s, tm ⟩, .inr varφ | .inr varφ, .inl ⟨ s, tm⟩ =>
    -- a formula and a term
    if h : s = .Bool
    then
      biff varφ
        (Term.equal (h ▸ tm) (fun j => Fin.elim0 j)
        (Term.func [] .Bool
        (show air_ast.Functions [] .Bool from .True)))
    else falsum
```

### 4.4.2 Interpretation of `Poly`

For the soundness theorem to relate the SST and AIR side, we must fix a particular interpretation for the `Poly` sort. Recall (§3.1.1) that `Poly` is the container into which polymorphic SST values are boxed. Intuitively, `Poly` should contain all the values of the parametric types.

We capture this intuition by interpreting `Poly` as a dependent pair: a type-parameter name $\alpha$ together with a domain `type_env`($\alpha$). Concretely, given a type variable valuation `type_env` : `String` → `Type`, we define the domain of the `Poly` sort as

```
def Poly_domain type_env: Type := Σ i : String, type_env i
```

That is, an element of `Poly` is a pair $\langle \alpha, x \rangle$ where $\alpha$ names a type parameter and $x$ is a value of the corresponding type.

This choice plays two roles in what follows. First, it provides the target sort for the encoding function $\mathsf{encode}_\alpha$ (§4.4.3), which packages an SST value of a polymorphic sort into a `Poly`-sorted AIR value by pairing it with its type tag. Second, when AIR axioms speak about $\texttt{has\_type}(v, \alpha)$ for a `Poly`-sorted $v$ and a `TYPE`-sorted $\alpha$, the pair lets us read off the type tag from the first component of $v$ and check that it matches the asserted type.

In the rest of the thesis, we write $C(\texttt{type\_env})$ to mean the AIR carrier that interprets `Poly`as `Poly_domain type_env`.

### 4.4.3 Correspondence between Valuations

To relate the semantics of SST and AIR, we must connect SST variable valuations with AIR variable assignments. This is achieved by an encoding function and a coherence relation.

The encoding function encode bridges the gap between the semantic domain of SST types and the domain of AIR sorts. Given a semantic value $x$ in the domain of type $t$, $\mathsf{encode}_t(x)$ produces an AIR semantic term in the domain of sort $\bar{t}$ that represents the same value. This function is essential for stating the correctness of the translation: it allows us to compare an SST valuation with another AIR valuation. The definition proceeds by recursion on $t$ and is largely straightforward: booleans, integers, and characters map to their obvious AIR literals; natural numbers are promoted to integers; type parameters $\alpha$ are encoded as a pair $\langle \alpha, x \rangle$ of sort `Poly`, which pairs the type parameter with the encoded value.

> **Definition 4.4 –** encode
>
> For each SST type $t$, and AIR carrier $C$ that maps each AIR sort to its interpretation, we define an encoding function
>
> $$\mathsf{encode}_t : [\![t]\!] \longrightarrow C(\bar{t})$$

by recursion on $t$:

$$\begin{aligned} \mathsf{encode}_{\mathtt{Bool}}(b) &= b \\ \mathsf{encode}_{\mathtt{Int}}(i) &= i \\ \mathsf{encode}_{\mathtt{Nat}}(n) &= n \text{ as } \mathtt{Int} \\ \mathsf{encode}_{\mathtt{Char}}(c) &= \mathrm{unicode}(c) \\ \mathsf{encode}_{\alpha}(x) &= \langle \alpha,\, x \rangle \end{aligned}$$

The encoding function is formalized as follows:

```
def encode (type_env : String → Type)
    (dom_aux : (String → Type) → Typ → Type)
    (t : Typ) {T F : Type} :
    typ_rep type_env dom_aux t →
    MSLanguage.AirCarrier (Poly_domain type_env) T F (trans_typ t) :=
  match t with
  | ._Bool      => fun v => cast interp_bool v
  | .Int .Int   => fun v => cast interp_int v
  | .Int .Nat   => fun v => Int.ofNat (cast interp_nat v)
  | .TypParam p => fun v => ⟨ p, cast (interp_typparam p) v ⟩
  ...
```

With encode in hand, we can relate SST variable valuations to AIR variable valuations. The relation $v \sim u$ holds exactly when for every variable, the AIR valuation $u$ maps the translated variable to the encoding of the value that the SST valuation $v$ assigns to the original variable. This coherence condition is a key hypothesis in the soundness proof: it guarantees that the translation does not change the meaning of variables.

**Definition 4.5 – Valuation Coherence**

Let $v : (i : \mathrm{Var}) \to [\![\Gamma[i]]\!]$ be a SST variable valuation in the context $\Gamma$. Let $u : (i : \mathrm{Var}) \to C(\overline{\Gamma[i]})$ be a variable assignment of AIR. The correspondence relation $\sim$ is defined as:

$$v \sim u \;\Leftrightarrow\; \text{for all } i \text{ in } \Gamma,\; u(i) \;=\; \mathsf{encode}_{\Gamma[i]}\big(v(i)\big).$$

The relation $\sim$ is formalized as follows:

```
def CoherentAssignment {type_env : String → Type}
    {dom_aux : (String → Type) → Typ → Type}
    {Γ : context}
    (venv : @val_vars' type_env Γ dom_aux) {T F : Type}
    (v : TransVarFam →ₛ AirCarrier (Poly_rep type_env) T F) : Prop :=
  ∀ (i : Nat) (hi : i < Γ.length),
    v (trans_typ Γ[i]) i = encode type_env dom_aux Γ[i] (venv i hi)
```

### 4.4.4 Soundness Theorem

The soundness theorem relates the denotational semantics of SST with the model-theoretic semantics of AIR. It states that if the translations of two SST expressions are semantically equivalent in all AIR models that satisfy the relevant axioms, then the original SST expressions are denotationally equal.

More precisely, fix a type variable context $\Delta$, a typing context $\Gamma_\Delta$ whose types are well-formed under $\Delta$, a type variable valuation type_env, and an SST variable valuation $v$ for $\Gamma_\Delta$. Let $e_1, e_2$ be SST expressions such that $\Gamma_\Delta \vdash e_1 : t$ and $\Gamma_\Delta \vdash e_2 : t$. Their translations produce pairs $(\overline{e_1}, A_1)$ and $(\overline{e_2}, A_2)$, where $A_1, A_2$ are the accumulated axioms generated during translation.

Now consider any AIR carrier $C(\text{type\_env})$ derived from the type variable valuation, and any AIR variable assignment $u$ that is coherent with the SST valuation $v$ (written $v \sim u$), meaning that $u$ assigns to each AIR variable the encoding of the value that $v$ assigns to the corresponding SST variable.

If, under every such choice of model and coherent valuation, the truth of the prelude axioms together with the accumulated axioms $A_1$ and $A_2$ forces the translations to be equivalent (i.e., $M_{C(\text{type\_env})}, u \vDash \overline{e_1} \approx \overline{e_2}$), then the original SST expressions must be denotationally equal: $[\![e_1]\!]_v = [\![e_2]\!]_v$.

> **Theorem 4.2 – Soundness of the Translation**
>
> Let $\Delta$ be a type variable context, $\Gamma_\Delta$ a typing context, type_env a type variable valuation, and $v$ an SST variable valuation in context $\Gamma_\Delta$. Let $e_1, e_2$ be SST expressions with

> $\Gamma_\Delta \vdash e_1 : t$ and $\Gamma_\Delta \vdash e_2 : t$, and let $(\overline{e_1}, A_1)$ and $(\overline{e_2}, A_2)$ be their respective translation results, where $A_1, A_2$ are the accumulated axiom sets.
>
> For every choice of AIR carrier $C(\mathsf{type\_env})$ and every AIR variable assignment $u$ with $v \sim u$ such that $M_{C(\mathsf{type\_env})} \models A_1 \cup A_2$,
>
> if
>
> $$M_{C(\mathsf{type\_env})}, u \models \overline{e_1} \approx \overline{e_2}$$
>
> then
>
> $$[\![e_1]\!]_v \;=\; [\![e_2]\!]_v.$$

Part of the proof of Theorem 4.2 proceeds as follows:

*Proof.* We proceed by induction on $t$.

1. $t = \mathtt{Bool}$. Now we induct on the structure of $e_1$.
   (a) $e_1 = \mathtt{true}$. Then we induct on the structure of $e_2$.
      i. $e_2 = \mathtt{true}$. In this case, $[\![e_1]\!]_v \;=\; [\![e_2]\!]_v$ always holds.
      ii. $e_2 = \mathtt{false}$. In this case, the theorem holds vacuously.
      iii. $e_2 = e_3 \wedge e_4$ for some $e_3$, $e_4$. $M, u \models \overline{e_3} \wedge \overline{e_4}$ under the hypothesis, and thus $M, u \models \overline{e_3}$ and $M, u \models \overline{e_4}$ both hold. By the induction hypothesis, we obtain that both $e_3$ and $e_4$ denote $\mathsf{true}$, so $[\![e_2]\!]_v = [\![e_3]\!]_v \wedge [\![e_4]\!]_v = \mathtt{true}$.
      iv. Other cases can be proved similarly.
   (b) $e_1 = \mathtt{false}$. This can be proved similarly as the previous case.
   (c) $e_1 = e_3 \wedge e_4$ for some $e_3$, $e_4$. We induct on the structure of $e_2$.
      i. $e_2 = \mathtt{true}$ or $\mathtt{false}$. These can be dealt with in a similar way as previous cases.
      ii. $e_2 = e_5 \wedge e_6$. Then $M, u \models \overline{e_5} \wedge \overline{e_6}$ iff $M, u \models \overline{e_3} \wedge \overline{e_4}$. If $M, u \models \overline{e_5} \wedge \overline{e_6}$, then $M, u \models \overline{e_3} \wedge \overline{e_4}$ as well. By the induction hypothesis, we obtain $[\![e_3]\!]_v = [\![e_4]\!]_v = [\![e_5]\!]_v = [\![e_6]\!]_v = \mathtt{true}$. Therefore $[\![e_3 \wedge e_4]\!]_v = [\![e_5 \wedge e_6]\!]_v$. If $M, u \not\models$

$\overline{e_5} \wedge \overline{e_6}$, then either $M, u \not\models \overline{e_5}$ or $M, u \not\models \overline{e_6}$. Without loss of generality, assume $M, u \not\models \overline{e_5}$. By the induction hypothesis, $[\![e_5]\!] = \texttt{false}$. As a result, $[\![e_5 \wedge e_6]\!]_v = [\![e_5]\!]_v \wedge [\![e_6]\!]_v = \texttt{false}$. Similarly, $[\![e_5 \wedge e_6]\!]_v = \texttt{false}$.

iii. Other cases can be proved similarly.

2. $t = \texttt{Int}$. Now we induct on the structure of $e_1$.

   (a) $e_1 = i$, $i$ constant. Then we induct on the structure of $e_2$.

      i. $e_2 = j$, $j$ constant. Since $M_{C(\texttt{type\_env})}, u \vDash i = j$, $[\![i]\!]_v = i = j = [\![j]\!]_v$.

      ii. $e_2 = e_3 + e_4$. Then $\overline{e_2} = \texttt{ADD}(\overline{e_3}, \overline{e_4})$. By the prelude axiom $\forall x\, y.\ \texttt{ADD}(x, y) = x + y$, we have $M, u \vDash \overline{e_3} + \overline{e_4} = i$. By the induction hypothesis on $e_3$ and $e_4$, we obtain $[\![e_3]\!]_v + [\![e_4]\!]_v = i$, so $[\![e_2]\!]_v = i = [\![e_1]\!]_v$.

      iii. Other cases can be proved similarly.

   (b) Other cases are left to future work.

3. $t = \alpha$, where $\alpha$ is a type parameter. Now we induct on the structure of $e_1$.

   (a) $e_1$ is a variable. We induct on the structure of $e_2$.

      i. $e_2$ is a variable. $v \sim u$ gives us the desired result.

   (b) $e_1$ is function application. This case is left to future work.

□

The theorem is formalized as follows:

```
theorem trans_sound
  {type_env : String → Type} {dom_aux : (String → Type) → Typ → Type}
  {Γ : context} {t : Typ}
  (e₁ e₂ : sst.Exp)
  (hty₁ : Γ ⊢ e₁ : t) (hty₂ : Γ ⊢ e₂ : t)
  (aenv : TransAxioms)
  (venv : @val_vars type_env Γ dom_aux)
  (h_air : ∀ (T F : Type) [M : AirMod (Poly_rep type_env) T F]
```

```
    (v : TransVarFam →ₛ AirMod.toFam (Poly_rep type_env) T F),
    CoherentAssignment venv v →
    AirMod.toFam (Poly_rep type_env) T F ⊨
      (trans_exp e₁ t hty₁ aenv).2 ∪ (trans_exp e₂ t hty₂ aenv).2 →
    (AirResultEquiv e₁ e₂ hty₁ hty₂).Realize v) :
-- The typ_rep denotations agree:
@exp_rep type_env dom_aux Γ venv t e₁ hty₁ =
@exp_rep type_env dom_aux Γ venv t e₂ hty₂
:= by ...
```

The mechanized proof of the theorem is left to future work.

# 5 Conclusion and Future Work

This thesis has presented a formalisation, in the Lean 4 theorem prover, of the SST-to-AIR translation pass inside the Verus verification tool. The formalization can be found at https://github.com/SSSPigeon/SST2AIR-in-Lean. The formalisation comprises four interlocking components. First, we modelled a fragment of Verus's SST, giving it a syntax of types and expressions, a static type system, a denotational semantics, and an inductively defined operational semantics that mirrors the Verus interpreter. Second, we formalized the AIR-AST as an instance of a generic multi-sorted first-order logic framework drawn from the `Many-Sorted-Model-Theory` library, instantiating its language with AIR's sorts and function symbols and its semantics with a parameterized carrier and the `AirMod` type class. Third, we defined the translation as a Lean function `trans_exp` that consumes a typed SST expression and produces an AIR term or formula together with a set of accumulated axioms. This function is accompanied by a translation of types, a fixed prelude of background axioms, and a type-argument encoding of polymorphism. Fourth, we stated a soundness theorem (Theorem 4.2) connecting the denotational semantics of SST with the model-theoretic semantics of AIR: whenever the translations of two SST expressions are equivalent in every AIR model that satisfies the prelude and the translation-generated axioms, and whenever the SST and AIR variable valuations are coherent, the original expressions are denotationally equal. Along the way we stated, but have not proved, on the SST side, preservation and soundness of the interpreter relative to the denotational semantics, justifying its use as a semantic-preserving reduction strategy in proofs by computation.

## 5.1 Limitations

The formalization in its current state has several limitations, each of which suggests a direction for further work. On the source-language side, we restricted attention to a fragment of SST expressions, omitting statements such as assignments, assertions, assumptions, and loops. Within the expression fragment itself, several constructors are not yet handled: mutable-borrow related forms (`VarLoc`, `Loc`), snapshot references (`Old`), fuel constants, internal interpreter values, and the type decorations that mediate between Rust's reference and

ownership machinery and Verus's logical view of values. Struct and enum constructors and field projections are present in the syntax but their denotational semantics and translation are stubbed out; they are mapped through the auxiliary `dom_aux` parameter rather than given concrete interpretations. Also, the denotational semantics and translation of functions and function applications are not discussed at the current stage.

On the target-language side, we likewise formalize only a fragment of AIR. At the level of sorts, the carrier assignment leaves the `Fun` and `TYPE` sorts as abstract parameters $F$ and $T$. The `AirMod` type class leaves operations that act on `Fun` or `TYPE` are unconstrained beyond what the prelude axioms impose. At the level of function symbols, several operations are not yet present in our `airFunc` signature; for instance, the array primitives `ARRAY_INDEX` and `ARRAY_NEW`, which Verus uses to encode arrays as functions, are absent, and any SST expression that would translate to them is correspondingly outside the scope of our `trans_exp`. At the level of background theory, we formalized a subset of the prelude axioms emitted by `prelude.rs`. In particular, we covered the arithmetic axioms relating uninterpreted symbols to their built-in counterparts and the Poly round-trip axioms governing boxing and unboxing (§4.2), but axioms specific to arrays-as-functions, bitvector reasoning, and the various decoration-related predicates remain as future work.

The proof of the soundness theorem itself is not completed. Of the inductive cases over the target type $t$ and the structure of $e_1$ and $e_2$, we have presented representative cases for $t = \texttt{Bool}$, $t = \texttt{Int}$ (integer constants, addition), and $t = \alpha$ (variables at polymorphic type), and have indicated how the remaining cases follow. A complete mechanisation will require a complete case analysis.

## 5.2 Future Work

We see four natural lines along which the formalisation can be extended.

**Completing the soundness proof.** The most immediate task is to complete and mechanize the proof of Theorem 4.2. The cases of arithmetic comparisons, conditionals, let-bindings, and quantifiers should follow the patterns already established for boolean connectives and arithmetic. Lambda abstractions and applications are more delicate, since the translation

interacts with the type-argument encoding of polymorphism: the soundness of an application $\mathsf{ap}(e_1, e_2)$ relies on the translation of $e_1$ producing an AIR function whose defining axioms agree with the meaning of $e_1$ at every concrete instantiation.

**Extending the source fragment.** An important extension is to define the semantics of functions, function applications, struct and enum. For these extension, we can follow the approaches in [6] and [5]. Beyond the expression fragment, the natural next target is the statement fragment of SST: assignments, assertions, assumptions, and control flow. These move the formalization from a question about denotational equality to a question about Hoare-style judgments relating preconditions and postconditions.

**Extending the target fragment.** The AIR formalisation itself can be filled out in several directions. The signature `airFunc` should be extended with primitives such as `ARRAY_INDEX` and `ARRAY_NEW`. This in turn requires committing to a concrete interpretation of the `Fun` sort, replacing the abstract parameter $F$ with a Lean function space. A similar move is needed for the `TYPE` sort: where the current development carries an abstract $T$, a concrete formalization would interpret `TYPE` with a concrete Lean type. With these in place, the remaining prelude axioms for arrays, bitvectors and primitive functions can be added.

**Connecting to the broader pipeline.** A long-term goal is to verify the full Verus pipeline end-to-end. This would require formalizations of HIR, VIR-AST, and the SMT-LIB encoding step that follows AIR, together with lemmas ensuring that a chain of sound passes is itself sound. CompCert [13] and CakeML [10] provide exemplars of how such a pipeline verification can be structured.

In summary, this thesis has taken a first step toward a fully mechanized correctness proof for the SST-to-AIR translation in Verus, delivering a Lean 4 formalisation of the source and target languages, the translation function, and a soundness theorem with proofs of representative cases. The development demonstrates that the end-to-end, proof-assistant-based approach to compiler verification scales to a real, in-use SMT-based verification tool, and lays the groundwork for the more ambitious goal of verifying the Verus pipeline as a whole.

# Acknowledgments

I wish to express my gratitude to all the individuals who contributed to the completion of this thesis. I would especially like to thank my supervisor, Professor Jeremy Avigad, for his guidance, insightful advice, and patience throughout this project. I am also sincerely grateful to my committee member Professor Bryan Parno, for all his help during this process. Beyond my committee, I would like to thank Dr Ruben Martins, Dr Reid Barton, and Professor Steve Awodey for all their support in this academic journey. I am also deeply thankful to my fellow students Amar Shah, Wojciech Nawrocki, and Joseph Hua, who generously shared their time and insights with me.

# References


[1] Verus tutorial and reference. https://verus-lang.github.io/verus/guide/, 2025. Accessed: 2026-03-27.

[2] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 31–46, 1989.

[3] J. Avigad, L. de Moura, and S. Kong. Theorem proving in lean. 2024.

[4] C. Barrett, A. Stump, C. Tinelli, et al. The smt-lib standard: Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)*, volume 13, page 14, 2010.

[5] A. Chlipala. *Certified programming with dependent types: a pragmatic introduction to the Coq proof assistant*. MIT Press, 2013.

[6] J. M. Cohen and P. Johnson-Freyd. A formalization of core why3 in coq. *Proc. ACM Program. Lang.*, 8(POPL), Jan. 2024.

[7] L. de Moura and N. Bjørner. Z3: an efficient smt solver. In *2008 Tools and Algorithms for Construction and Analysis of Systems*, pages 337–340. Springer, Berlin, Heidelberg, March 2008.

[8] L. de Moura and S. Ullrich. The Lean 4 theorem prover and programming language. In *Automated Deduction – CADE 28*, volume 12699 of *LNCS*, pages 625–635. Springer, 2021.

[9] DeepWiki. Verus — DeepWiki. https://deepwiki.com/verus-lang/verus, 2025. Accessed: 2026-04-06.

[10] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. Cakeml: a verified implementation of ml. *ACM SIGPLAN Notices*, 49(1):179–191, 2014.

[11] A. Lattuada, T. Hance, C. Cho, M. Brun, I. Subasinghe, Y. Zhou, J. Howell, B. Parno, and C. Hawblitzel. Verus: Verifying rust programs using linear ghost types. *Proc. ACM Program. Lang.*, 7(OOPSLA1), Apr. 2023.

[12] K. R. M. Leino and P. Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2010)*, volume 6015 of *LNCS*, pages 312–327. Springer, 2010.

[13] X. Leroy, S. Blazy, D. Kästner, B. Schommer, M. Pister, and C. Ferdinand. Compcert-a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, 2016.

[14] L. Li and E. L. Gunter. K-llvm: a relatively complete semantics of llvm ir. In *34th European Conference on Object-Oriented Programming (ECOOP 2020)*, pages 7–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020.

[15] N. P. Lopes, J. Lee, C.-K. Hur, Z. Liu, and J. Regehr. Alive2: bounded translation validation for llvm. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 65–79, 2021.

[16] N. P. Lopes, D. Menendez, S. Nagarakatte, and J. Regehr. Provably correct peephole optimizations with alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 22–32, 2015.

[17] Mathlib Community. Mathlib4: The Lean 4 mathematical library. https://github.com/leanprover-community/mathlib4, 2025. Accessed: 2026-03-27.

[18] B. C. Pierce, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, V. Sjöberg, and B. Yorgey. Software foundations. *Webpage: http://www. cis. upenn. edu/bcpierce/sf/current/index. html*, 16, 2010.

[19] S. Schäfer, T. Tebbi, and G. Smolka. Autosubst: Reasoning with de bruijn terms and parallel substitutions. In *International Conference on Interactive Theorem Proving*, pages 359–374. Springer, 2015.

[20] K. Stark, S. Schäfer, and J. Kaiser. Autosubst 2: reasoning with multi-sorted de bruijn terms and vector substitutions. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 166–180, 2019.

[21] M. Stout and J. Nicholson. Many sorted model theory. https://github.com/Mathias-Stout/Many-sorted-model-theory, 2026. Accessed: 2026-03-27.

[22] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic. Formalizing the llvm intermediate representation for verified program transformations. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 427–440, 2012.