Università degli Studi di Napoli Federico II

Scuola Politecnica e delle Scienze di Base

Dipartimento di Ingegneria Elettrica
e delle Tecnologie dell'Informazione

Informatica

Classe L-31

Tesi di Laurea Triennale

Aristarchus: Design and Implementation of a Verifier for the Sunya Dialect of Hypatia

Relatore
Fabio Mogavero

Candidato
Roberto Fiorino
Matricola N86004967

Anno Accademico 2025–2026

HYPATIA

Bachelor Theses

Bachelor Thesis

Aristarchus: Design and Implementation of a Verifier for the Sunya Dialect of Hypatia

Roberto Fiorino

Component: Aristarchus

Developed within the Hypatia project

hypatiafsa.github.io

github.com/hypatiafsa

Abstract

Many Interactive Theorem Provers entrust the correctness of proofs to a verification kernel, which nonetheless remains a program subject to errors, as also shown by a recent bug found in the Lean 4 kernel. Hypatia, a formal language and suite of tools still under development, proposes to make explicit the path that leads from a formal text written by the user to the object that the system actually verifies, by means of a hierarchy of dialects, Sunya, Sifr and Zero, linked by translations towards the underlying level, designed to be certifiable in the future.

This thesis concerns Aristarchus, the component responsible for verification, and describes its design and implementation for the Sunya dialect alone, in the Unica mode in which every construct admits a single interpretation. The verifier, written in Haskell, operates in two phases. In the first, starting from the source document, whose syntax is context-dependent, an Abstract Syntax Tree is built. In the second, this AST is traversed to check the vocabulary, phrase and schema declarations, proof steps, dependencies between them, any cycles and conjectural results; the process ends with a verdict accompanied by diagnostics.

Contents

Abstract **i**

1 Introduction **1**

2 Background **4**
2.1 Formal Verification and Theorem Proving 4
2.1.1 Automated and Interactive Theorem Provers 4
2.2 State of the Art in Interactive Theorem Proving 5
2.2.1 Foundations: from AUTOMATH to LCF 5
2.2.2 A Comparative Study of Four Formal Systems: Metamath, Mizar, Rocq, and Lean 6
2.3 The Trust Problem and System Vulnerabilities 11
2.3.1 Case Study: A soundness bug in the Lean 4 kernel 11
2.4 Positioning Hypatia 12

3 Problem **13**
3.1 Problem Definition 13
3.1.1 Input, Output, and the Verdict 13
3.1.2 What Correctness Means for a Hypatia Document 13
3.1.3 What the Problem Is Not 13
3.1.4 Solution Approach 14
3.2 Domain Specifications 15
3.2.1 Tower of Dialects 15
3.2.2 Bootstrapping and the Self-Formalisation Hierarchy 15
3.2.3 Document and Language Structure 16
3.2.4 Host Language, Meta Language, and Object Language 17
3.2.5 Representations 17
3.2.6 A note on the generality of representations 19
3.2.7 Active Terminals and Escaping 19
3.2.8 Syntax and Semantics 19

4 Design **51**
4.1 Requirements 51
4.1.1 Functional Requirements 51
4.1.2 Non-Functional Requirements 52
4.2 System Architecture 53
4.2.1 The Command Line Interface 53
4.2.2 Overview: From Document to Verdict 53
4.3 The Trusted Code Base 55
4.4 Abstract Syntax Tree Modelling 55
4.4.1 What the Representation Is For 55

4.4.2 What It Discards . . . 56
4.4.3 Object Text Delimiters . . . 56
4.4.4 The Shape of a Document . . . 56
4.4.5 Statements: Phrases, Object Text, and Schemas . . . 57
4.4.6 Proof Scripts . . . 58
4.4.7 Source Positions . . . 58
4.4.8 Single-Interpretation Representation . . . 58
4.5 The Verification Model . . . 59
4.5.1 Three Channels . . . 59
4.5.2 Order of Traversal . . . 59

5 Implementation 60
5.1 The Haskell Programming Language . . . 60
5.2 System Architecture and Module Structure . . . 60
5.2.1 Imperative Shell: `App` . . . 60
5.2.2 Shared Data Structures: `Hypatia.AST` . . . 60
5.2.3 Construction: `Hypatia.Parser` . . . 61
5.2.4 Logical Kernel: `Hypatia.Toolkits.Validator` . . . 61
5.3 The Command-Line Interface and the I/O Boundary . . . 62
5.3.1 File Interfacing and Exception Handling . . . 62
5.3.2 Orchestrating Pure State within the IO Monad . . . 62
5.3.3 Exit Codes . . . 64
5.4 Parser Architecture and Fundamental Types . . . 64
5.4.1 The Dynamic State: The `Vocabulary` . . . 64
5.4.2 The Choice of Parsing Engine: Megaparsec . . . 65
5.4.3 The Monad Stack and the Input Flow . . . 65
5.4.4 Blob Handling . . . 66
5.4.5 Byte Extraction and Escape Handling . . . 66
5.4.6 Reopening fragments: *merge*, not restoration . . . 68
5.4.7 `ActiveTerminals` as a context parameter . . . 69
5.4.8 Parser errors . . . 70
5.4.9 Positions: byte offsets and UTF-8 rendering . . . 72
5.5 Verifier Architecture and Fundamental Types . . . 77
5.5.1 Declarative traversal and updating of the environment . . . 79
5.5.2 Representation of declared items and error tolerance . . . 80
5.5.3 Qualified references and export between fragments . . . 81
5.5.4 Verification of proof scripts and local state . . . 82
5.5.5 Re-interpretation of instances after substitution . . . 84
5.5.6 Prevention of proof circularity . . . 86

6 Validation and Testing 88
6.1 A case study: elementary ring theory . . . 88
6.1.1 Notation: prefix, fixed arity . . . 89
6.1.2 Principles and rules of the system . . . 89
6.1.3 Formation of terms . . . 89
6.1.4 Equality . . . 90
6.1.5 The ring axioms . . . 90

6.1.6 A guided proof: `ZeroPlusZero` 91
6.1.7 A proof with a premise: `TrmMulZero` 92
6.1.8 The remaining derivations, in summary 93
6.2 Testing with Hspec 102
6.3 End-to-End Validation 104
6.3.1 Cases generated by the Metis Metamath translator 104
6.3.2 Corpus-Based Testing 105
6.3.3 Testing the Layered Formalisations 105

7 Conclusions **108**
7.1 Future developments 108

Bibliography **109**

Chapter 1

Introduction

Formal verification changes the role of a mathematical proof. A proof is no longer just an argument that a person has to examine. It becomes a formal object, whose correctness is checked automatically. Computer-assisted reasoning has been working on this idea for decades, from the first attempts at formalisation in systems such as AUTOMATH up to the interactive theorem provers and formal libraries of today. The aim was to make mathematical reasoning precise to the point that a machine could establish its correctness, without relying solely on informal inspection. The QED manifesto made this aim explicit. It proposed a computerised archive in which every piece of mathematical knowledge had a formal proof, verifiable by an algorithm. That vision was not realised in its original form. It has, however, left a problem that remains central still today, namely how to verify a proof so that its correctness is guaranteed.

From this aim two main approaches arise. Automated theorem provers build proofs with little or no human intervention, exploring a logical search space. Interactive theorem provers, instead, leave the guidance of the proof to the user, and use the machine chiefly as a verifier. The two approaches also change the relationship between the mathematical text, the representation of the proof and the mechanisms that establish its correctness. Interactive theorem provers are further distinguished by how they express proofs. In procedural systems the user writes commands, called tactics, which transform the proof goals. In declarative systems, instead, the proof is a sequence of logical statements, while other systems mix interaction and automation.

Two architectures address the reliability of the final verification step. The first originates with de Bruijn's AUTOMATH. Proofs become formal objects, and verifying them reduces to checking the corresponding formal terms. This idea is linked to the Curry-Howard isomorphism, which interprets propositions as types and proofs as terms that inhabit those types. A second architecture is LCF, which instead encapsulates the primitive inference rules in a small trusted kernel, and the abstract type that represents theorems is built only through these kernel operations, so that no program external to the kernel can create that type.

These ideas continue to influence contemporary formal systems. Metamath adopts a minimal approach. It represents mathematical statements as sequences of symbols, and verifies proofs by means of substitutions and a handful of primitive mechanisms. Mizar follows a different path, using a declarative language, closer to mathematical notation, and a broader verification pipeline. Rocq is based on dependent type theory and on the Curry-Howard correspondence. A kernel verifies the resulting proof terms. Lean combines a small trusted kernel with an extensible environment, in which parsers, tactics and other components can be programmed.

These systems are therefore very different from one another, yet they still share the same problem of trust. There is a distinction between the Trusted Theory Base, that is the logical and mathematical foundations of the system, and the Trusted Code Base, that is the

implementation on which the outcome of the verification depends. These two sources of trust are independent.

The Trusted Code Base matters above all when the verification mechanism has a flaw. Today's proof assistants are complex programs, and for this reason even their kernels can have implementation errors. This thesis discusses a recent example, a soundness issue in the Lean 4 kernel; a vulnerability in the handling of nested inductive types made it possible to construct a term that the kernel accepted as a proof of *False*.

The de Bruijn criterion and the LCF approach described at the beginning both reduce the risk associated with the Trusted Code Base. Another approach is to verify the verifier itself, as the MetaCoq project does. But to do so, the boundary of the verification must first be identified.

Artificial intelligence in formalised mathematics also brings trust problems. A system trained to obtain the prover's acceptance can reach it through a defect in parsing, elaboration or kernel code, independently of the mathematical content. A generated artefact might therefore be accepted even if the process that produced it is neither understandable nor reliable. A clear verification boundary, independently checkable, does not require trusting whoever or whatever produced the artefact; the check concerns the artefact itself, according to criteria that anyone can inspect.

Hypatia is a language for defining formal systems and writing derivations within them. It is organised as a public tower of formal dialects. The aim is to reduce higher-level descriptions to a minimal core, through explicit translations which, in the architecture planned for the future, will also be certifiable. This addresses two related problems. The semantic gap is the distance between a high-level formal description and the lower-level representation that a system ultimately verifies. The trust gap is instead the amount of implementation that has to be trusted between these two levels. Hypatia places the different translation steps in explicit linguistic levels, and the relationships between these levels can in turn be formalised.

Hypatia is organised as a tower of dialects. Sunya is the minimal core. It is explicit and deliberately verbose. Its semantics is based on the declarations in the document, not on a fixed logic or mathematical theory. Sifr is a syntactic level above Sunya, and adds syntactic sugar. Zero is instead an experimental dialect, intended for more complex and more structured reasoning. At the moment only Sunya exists.

Hypatia can formalise itself, within Sunya. This self-formalisation is a cumulative hierarchy. It starts from the symbolic and syntactic infrastructure of Elementa, continues with the metatheory of Fundamenta and with the textual infrastructure of Scriptum, moves on to the external grammar of Hypatia, and arrives at the formalisation of the individual dialects.

The aim of this thesis is to establish whether a Hypatia document is written correctly, and whether its declarations and its proof scripts comply with the constraints of the language.

This thesis addresses the problem with Aristarchus, Hypatia's verification component. The implementation in this work is limited to the Sunya dialect, in Unica mode, in which every construct must have exactly one valid interpretation. The language envisages the implementation of other modes in the future, but they are not the subject of this thesis.

The problem splits into two phases. First, the source document becomes a structured representation, which preserves the information needed for verification. This phase has to reckon with a language that is context-dependent. Symbols, lexical elements, mutabilia,

delimiters and active terminals can depend on the declarations read earlier. Nested fragments then add local contexts, which must remain distinct from the surrounding ones.

The second phase is validation. Once the structured representation is ready, the verifier traverses it following the structure of the document and the context accumulated up to that point. It checks the vocabulary, phrase and schema declarations, verifies the proof steps and their dependencies, resolves references to elements, and ensures that the proofs establish exactly the statement declared by the relevant productum or derivatio. In proofs of schemas, the premises introduced during the proof also have to be discharged, in order to reconstruct the schema that the derivation establishes. The verifier must also reject circular dependencies and distinguish elements genuinely established from those whose proof depends on a conjectural result.

The final result is a verdict, together with structured diagnostics and information on the purity of the formalisation. The system accumulates errors instead of stopping at the first one, so it can report several independent defects within the same run.

The implementation is designed as a small, explicit verification core. In the future its behaviour may be formalised and certified.

The rest of the thesis looks at this aim from three points of view. The chapter on design identifies Aristarchus's requirements, defines its architecture, and establishes the boundary of its Trusted Code Base. The chapter on implementation describes how the parser and the validator put this design into practice. It addresses context-dependent syntax, structured representations, proof state and logical verification. Finally, the chapter on validation evaluates the implementation with unit tests and end-to-end cases.

Chapter 2

Background

2.1 Formal Verification and Theorem Proving

Formal verification, in the context of computer-aided reasoning, marks a shift in the role of mathematical proofs: a proof is a formal object whose correctness a program can check.

The motivation for this shift was explicitly articulated in the QED Manifesto [5]. It proposed a computerised repository in which every theorem is proved and machine-checked. The objective was to reduce human error in mathematical literature and establish a formal foundation for mathematical deduction. The original vision of the QED project was never fully realised, but it established the fundamental requirements for modern formalisation. Computer-aided reasoning has developed along two lines of research: automated theorem provers (ATPs) and interactive theorem provers (ITPs).

2.1.1 Automated and Interactive Theorem Provers

Automated Theorem Provers are computational systems designed to automate logical deduction and proof search [16]. Given a formal specification of axioms and a conjecture, an ATP autonomously explores the logical search space to construct a valid derivation or find a contradiction. Robinson introduced the resolution principle, on which many automated provers are based [15]. Prior to this work, automated provers struggled with the problem of determining how variables should be instantiated, which could lead to a combinatorial explosion in the search space. Robinson's work addressed this problem through unification [1], which determines substitutions that make two logical terms identical. Combined with resolution, this provided a general inference mechanism for automated deduction.

In contrast, **Interactive Theorem Provers** rely on human guidance to build a proof, while the machine acts as a verifier. We adopt the terminology of Wiedijk [18]:

- **Formalising** is the process of translating a mathematical text into a formal language, which is a language with a well-defined syntax and semantics.
- **Formalisation** is the resulting set of input files.

There are two distinct levels of formalisation, the user level and the proof object level. The first one is the text written by the user, while the second one is the corresponding proof represented within the underlying formal system.

This type of theorem provers can use different proof styles [10, 18]:

- **Procedural style:** the proof is a sequence of commands, known as tactics, that transform proof obligations called goals. Each tactic reduces a goal to zero or more sub-goals. Coq, the HOL family, PVS, Matita and Metamath belong to this class.
- **Declarative style:** the user writes the proof in a stylised natural deduction language

[1] Unification is the process of finding a specific substitution that makes a set of logical expressions structurally identical

and the systems reports where the text still fails to check. Two subclasses are distinguished:

  - *natural language declarative*, where the proof language looks similar to formal mathematical writing and the system uses lightweight automation to infer trivial logical steps between the explicit statements. Some examples are Mizar and Isabelle/Isar.
  - *proof object declarative*, where the input language is a syntactic representation of the underlying proof object itself, carrying the structure of a natural deduction proof, as in Twelf and Agda. In some of these systems the user may generate part of the proof text through interface commands, which are not themselves part of the final formalisation.

- **Guided automated style:** the input is a sequence of lemma statements that the system attempts to prove on its own. User guides it by choosing the statements so that the gaps between them remain small, which is what keeps these systems interactive. Suppose we want to prove $A \implies D$ and the system knows how to get $A \implies B$, $B \implies C$ and $C \implies D$, we can provide a sequence as $A \implies B$, $A \implies C$ and $A \implies D$ so that the gaps remain small.

2.2 State of the Art in Interactive Theorem Proving

2.2.1 Foundations: from AUTOMATH to LCF

Two architectures underlie most interactive theorem provers. Both aim to ensure the correctness of mechanically verified mathematics, one through the generation of independent proof objects, the other through the encapsulation of logical rules within a small verification kernel.

The first approach originated with the Automath project, initiated by de Bruijn in 1967. Automath was the first system to treat proofs as first class objects in a formal language, placing them on the same footing as other terms. This approach is grounded in the Curry-Howard isomorphism, which establishes a structural correspondence between logic and computation. Under this isomorphism, a logical formula corresponds to a computational type, and a proof of that formula corresponds to a term of the corresponding type. Formally, a logical formula ϕ is derivable if and only if there exists a term M of type $T(\phi)$:

$$\Gamma \vdash_{\text{logic}} \phi \iff \Gamma' \vdash_{\text{type theory}} M : T(\phi) \tag{2.1}$$

In this equivalence, the left side represents traditional logical deduction, where Γ contains the assumptions. On the right side, representing type theory, the context Γ' is expanded to include the declarations $y : A$ of all the free variables occurring in the formulas. The term M is a direct encoding of the deduction of ϕ from Γ. It explicitly translates the logical deduction of ϕ from the assumptions in Γ into a computational object, such as a λ-term. In this system, assumptions in Γ' serve as hypothetical proofs, while proven lemmas become named proof objects. An assumption takes the form $y : T(\psi)$, treating y as a hypothetical proof of ψ; in contrast, a proven lemma is defined as $y := p : T(\psi)$, where p is the actual proof term that y names. Under this correspondence, proof checking reduces to type checking.

The second paradigm emerged from the LCF (Logic for Computable Functions) system.

The name was coined by Milner for a formal theory originally developed by Scott in 1969; the first implementation followed at Stanford in 1972, with later systems developed at Edinburgh and Cambridge. His early system was goal-directed, using tactics to break a goal down into simpler subgoals, and a built-in simplifier to solve easy cases. What Milner wanted was a way to let users write their own tactics, without risking unsoundness, and without having to keep every proof fully spelled out in memory, since rechecking a large proof was expensive. He represented theorems as an abstract data type, `thm`, whose only constants are the axioms and whose only functions are the primitive inference rules. The only terms of this type are derivable sequents $\Gamma \vdash \phi$, since there is no way to construct a value of type `thm` except by going through those functions. The operator `assume : form -> thm`, for instance, takes a formula ϕ and returns the derivable sequent $\phi \vdash \phi$. A term of type `thm` can therefore only ever be constructed by going through the kernel. Milner built the language ML specifically to write this kernel in. In LCF, every value of type `thm` is produced by a primitive inference rule, so every theorem is derivable provided that the kernel implementing these rules is correct. This is a different way of trusting the system. While Automath's checker looks at a finished proof term and confirms that it is well-typed, LCF restricts the construction of values of type `thm` to the kernel. The trade-off is that, by default, LCF does not retain an explicit proof object, only the resulting theorem. A proof object can still be generated separately if required, but it is not needed for the kernel to establish the theorem [8].

2.2.2 A Comparative Study of Four Formal Systems: Metamath, Mizar, Rocq, and Lean

Metamath Metamath was developed by Norman Megill in 1992 and distinguishes itself through its approach based on logical minimalism. Metamath operates as a pure metalanguage instead of integrating inference rules or first order logic into its own kernel, as other provers do [6]. That means it does not recognise the syntactic structure of well-formed formulas. Instead, every mathematical statement is treated as a finite sequence of symbols manipulated by substitutions. Axioms, definitions and inference rules are introduced externally via databases [11].

It checks proofs written by the user and performs no proof search. The language has 11 keywords, each one identified by $. The table at 2.1 summarises the fundamental keywords in Metamath.

Table 2.1: Fundamental keywords in Metamath.

Keyword	Description
$c	Define a constant of the language
$v	Define a variable of the language
$f	Specify the type of a variable
$e	Define a temporary logical assumption
$a	Introduce an axiom, a definition or a syntactic rule
$p	Introduce a theorem, followed by its formal proof delimited by $= and $.
${ and $ }	Define a block with its own local scope
$d	Specify that distinct variables cannot share terms

Metamath contains various mathematical databases, such as set.mm, which is the

principal Metamath database based on ZFC and contains developments in several areas of classical mathematics, including topology and number theory.

The verification process is based on a stack architecture to hold intermediate results, this architecture use the Reverse Polish Notation (RPN)[2]. A formal proof in metamath is just a sequence of labels; the verifier reads these one by one and applies the set of rules defined below [11] and summarised in Figure 2.1:

- If the label corresponds to a hypothesis or a basic variable, its symbol sequence is pushed onto the stack;
- When Metamath encounters an assertion label, it associates the most recent stack entries with the mandatory hypotheses of the assertion, starting in reverse order with the last mandatory hypothesis.
- It then applies the unification mechanism to determine what substitutions must be made to the variables of the assertion's mandatory hypotheses to make them identical to the associated stack entries. It then applies the substitution to the assertion itself.
- Finally, Metamath pops the matched hypotheses from the stack and pushes the substituted assertion.

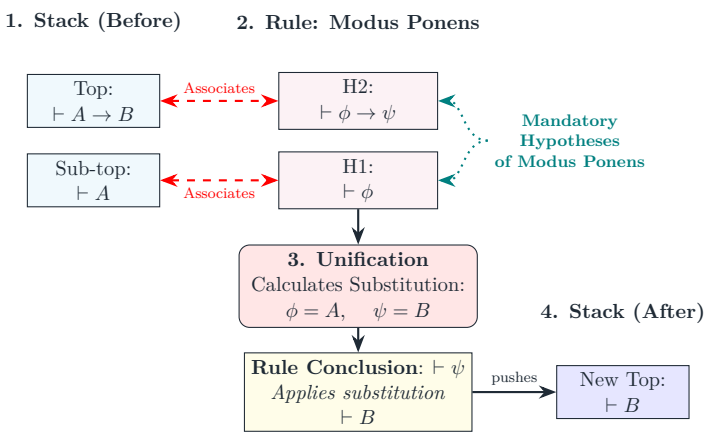

Figure 2.1: Abstract representation of the pattern matching and unification process in Metamath.

The simplicity of the Metamath verifier has facilitated certified conversions from other proof formats. Carneiro, for example, translated proofs from the OpenTheory exchange format into Metamath. Firstly, he translated OpenTheory's higher-order logic into Hol.mm; secondly, he translated the HOL system, which is based on type theory, into set.mm, which is instead based on Zermelo-Fraenkel set theory (ZFC). In short, he translated types into classical sets and functions into sets of ordered pairs. This serves a dual purpose: on the one hand, it enables Metamath to retrieve already-proven theorems; on the other, it validates proofs generated by a complex system using its minimal kernel, reducing the risk of errors [6].

Mizar Mizar was designed by Andrzej Trybulec in 1973 at the University of Białystok in Poland. It is an interactive theorem prover which, unlike Metamath, follows a declarative style. Indeed, the aim was to define a language as close as possible to mathematical notation,

[2]For example $(2 + 3) * 4$ becomes $2\ 3 + 4\ *$

the so-called *mathematical vernacular* [3, 9].
The file – also known as an 'article' – read by Mizar is divided into two main sections. The first is called the "environment" (**environ**) section and contains symbols and logical constructs defined in the Mizar Mathematical Library (MML). This is achieved using specific directives such as '**vocabularies**' for the lexicon, '**constructors**' for the semantics, and '**requirements**' to activate built-in procedures. The other section, namely the development section, is introduced with '**begin**' and contains the actual mathematical text [9].

A proof follows the structure of the formula being provided. For example, to prove $\phi_1 \implies \phi_2$, one introduces the antecedent by **assume** ϕ_1, develops the intermediate steps, and concludes with the thesis **thus** ϕ_2.

Mizar is not based on classical set theory, but on the Tarski–Grothendieck theory [8]. This is similar to ZFC but allows for the consideration of universes of sets. On top of this set-theoretic foundation, Mizar provides a soft type system, where each type can have an attribute (**attr**) that refines its semantics. Defining a new type requires proving an existence condition, to ensure that the set associated with that type is non-empty. Defining an operator requires a proof to ensure that the result belongs to the declared type. It is also possible to use registrations to define relationships between attributes, which the system can exploits during inference[9].

The verification pipeline goes through the following stages:

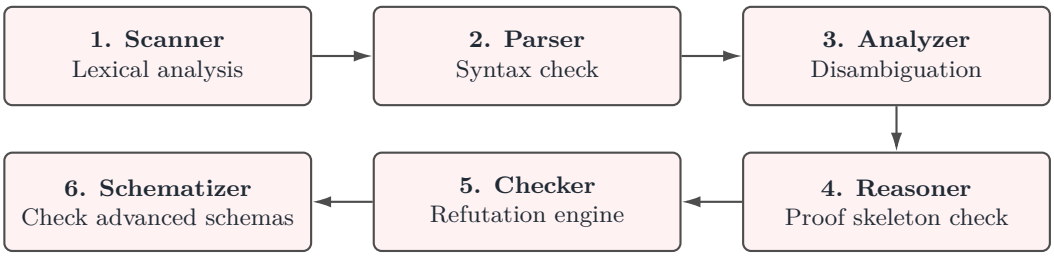

Figure 2.2: The Mizar verification pipeline architecture. The flow proceeds sequentially from lexical analysis to schema validation.

As can be inferred from the pipeline in Figure 2.2, Mizar does not have a minimal verification kernel and, consequently, does not satisfy De Bruijn's criterion, unlike Metamath, which does satisfy it.

Rocq Rocq was developed from 1984 onwards by Thierry Coquand. While Metamath is based on a minimalist approach and Mizar on a language close to mathematical prose, Rocq (or Coq) is based on the Calculus of Inductive Constructions and the Curry–Howard correspondence.

In Rocq, theorems are type declarations, and proving them means writing a programme that inhabits that type. Dependent types are types that accept values as parameters; this formalises universal quantification as a function that associates a proof of $P(x)$ with every element x[4].

Rocq's kernel implements the Calculus of Inductive Constructions (CIC), a constructive type theory with built-in reduction rules, including function application (β-reduction), pattern matching (ι-reduction), and the expansion of local (ζ) and global (δ) definitions. A consequence of this approach is the principle of **convertibility**: if two expressions reduce to the same result, Rocq regards them as identical and equivalent. This saves the user from

having to provide proofs for purely algorithmic equalities [17, 2].

Rocq classifies terms into different sorts [17]:

- **Prop:** This sort is reserved for logical propositions. Since the proofs inhabiting these propositions do not contribute to the computational content of a program, they are erased during program extraction. This property should not be confused with *Proof Irrelevance*, which concerns the logical significance of the particular proof used for a proposition. Rocq provides the separate `SProp` sort for propositions whose proofs are definitionally proof-irrelevant.

- **Set** and **Type:** While `Set` describes concrete data types and executable algorithms, classifying `Set` itself requires the `Type` sort. To prevent logical inconsistencies, `Type` is organised into an infinite cumulative hierarchy ($\texttt{Type}_0 : \texttt{Type}_1 : \ldots$). Allowing a type to belong to itself (e.g., `Type : Type`) would introduce cyclical, self-referential structures a vulnerability known in type theory as Girard's paradox. The hierarchy avoids it by placing each Type_i in Type_{i+1}.

Rocq uses an interactive procedural style. Proofs are constructed by interacting with an interpreter (REPL) through tactics, breaking down the initial objective into simpler sub-objectives. These tactics generate the proof object, which is then passed to a small kernel that performs the type-checking [18, 4]. As the kernel is minimal, Rocq satisfies De Bruijn's criterion.

A distinctive feature of Rocq is that, thanks to its division into sorts, it is able to extract functional programs from proofs. Extraction is implemented in OCaml, and is not verified; hence the MetaCoq project was launched, which formalises Rocq within Rocq itself [4].

Lean Lean was developed starting in 2013 by Leonardo de Moura at Microsoft Research to combine interactive proving with automation.

After the initial release, several versions followed, such as Lean 2, which supported two interchangeable logical frameworks for its kernel: CIC and Homotopy Type Theory (HoTT), which used proof-relevant logic. The subsequent version, Lean 3, removed HoTT to focus on CIC. This marked the beginning of the Mathlib project, which completed the formalisation of Peter Scholze's Liquid Tensor Experiment. Despite this, Lean 3 had performance issues [13, 14].

In the following years, Lean 4 was released, with the goal of making Lean *self-hosting*, that is, implementing Lean 4 in Lean itself. This new version introduced several changes, including support for defining new tactics and extending the language. In fact, by importing the Lean package, it is possible to modify components of the system, such as the parser, the pretty-printer, or even the compiler while using the system itself. Typically, the creation of new syntax is subject to "accidental name capture" within tactical scripts, causing bugs. One solution was to implement a system of hygienic macros, which handled the expansion of the macro [13].

As previously mentioned, earlier versions of Lean were prone to a decline in performance due to the "diamond problem", which was caused by strong dependencies between structures [13]. Let us consider the following situation:

- A ring is both an abelian group and a semiring;

- both the abelian group and the semiring derive from the monoid.

If Lean 3 has to prove that a certain structure T is a monoid, the search algorithm explores the ring branch, then the abelian group branch until it reaches the monoid, where it either fails or succeeds. It then backtracks, goes back, and explores the semiring branch, recalculating everything from scratch to reach the monoid again. As the depth increases, the time complexity becomes $O(2^n)$. Lean 4 addresses this with two techniques: the first involves using Discrimination Trees to index the data and thereby reduce search times; the second involves tabling, i.e. storing intermediate calculations in a cache so that they can be reused.

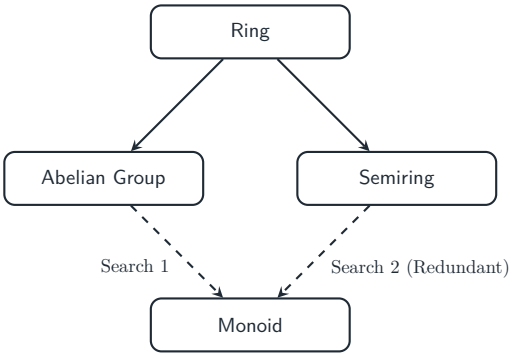

Figure 2.3: Representation of the *Diamond Problem*.

Lean 4 also has its own code generator. It translates its logical expressions into an *intermediate representation* (IR), which is then compiled into C code. This allows users to compile custom tactics or decision procedures and load them directly into the system as plugins compiled at runtime, avoiding the interpretation overhead of the virtual machine. This procedure relies on the FBIP (*Functional But In-Place*) paradigm. Lean 4 manages memory with reference counting instead of a tracing garbage collector. The key concept is the *Reuse Hypothesis*. When an object's count is one, the runtime updates it in place*in-place* [13].

2.3 The Trust Problem and System Vulnerabilities

The use of these verification systems gives rise to a recursive problem, known as the "who checks the checker" problem. To analyse the security of these systems, a distinction is made between two components: the Trusted Theory Base (TTB) and the Trusted Code Base (TCB) [17].

The *Trusted Theory Base* comprises the logical and mathematical foundations on which the system is based, such as, for example, higher-order logic. An error at this level can compromise the validity of the verification, even if the implementation is correct. The problem of *wrong mathematics* [18] also falls into this category: if the definitions formalised by the user do not correctly represent the problem they intend to describe, the assistant may still correctly verify a proof of a statement that does not correspond to the original problem. The introduction of new axioms could also lead to conflicts. A historical example is the *Univalence Axiom*, which temporarily broke the assumptions of the guard-checking algorithm for recursion in Coq [17].

The *Trusted Code Base*, by contrast, represents the actual implementation of the assistant. As these are highly complex programs, the risk of bugs is high. To reduce the TCB, two strategies are employed (both discussed in section 2.2.1) [17, 8]:

- *De Bruijn's criterion*, which involves designing an ITP so that it generates a proof object that is then checked by a minimal kernel.
- *LCF architecture*, where theorems are treated as an abstract data type, and the only way to create one is through the kernel, which implements the primitive inference rules.

Another approach is self-verification. The MetaCoq project has formalised the syntax and semantics of the Coq kernel within Coq itself, thereby enabling its verification. A compilation pipeline could then be created to validate the code down to machine code [17].

2.3.1 Case Study: A soundness bug in the Lean 4 kernel

A soundness bug in the Lean 4 kernel, fixed in release 4.32.2 (July 2026), shows the role of the TCB.

The problem emerged when an alleged formal proof of the falsity of the Collatz conjecture was published. The proof was free of unsafe axioms (*sorry-free*), but used a kernel defect to obtain acceptance of a proof of `False`.

The vulnerability was in the C++ implementation of the kernel, specifically in the handling of the reduction of nested inductive types. In the definition of these types, it is possible to declare *phantom parameters*, that is, parameters that appear in the type signature but are not used in the constructor fields. When the type is instantiated, an argument must still be provided for that parameter. Due to an incorrect optimisation in the code, the kernel did not check the argument provided for the phantom parameter.

In short, it was possible to provide a logically invalid term as an argument for the phantom parameter. Since the kernel ignored that argument, the term was accepted as valid.

Under normal circumstances, Lean's *frontend* checks types and would have rejected the error before it reached the kernel. However, metaprogramming was used to manipulate and dynamically generate the Abstract Syntax Tree in order to pass the term directly to the kernel.

The same proof term could also be accepted by Nanoda, the main external and independent verifier for Lean, written in Rust. This was not due to the same vulnerability: Nanoda contained a different and completely independent bug in the verification of names within specific projection nodes [12].

2.4 Positioning Hypatia

Hypatia is designed as a metalanguage for representing formal systems and as a toolkit for checking documents and translations between its dialects.

The growing use of artificial intelligence in formalised mathematics raises significant reliability concerns. An automated system might not be concerned with finding a mathematically meaningful proof, but rather with constructing a term that the prover's kernel accepts as valid.

In traditional ITPs, such as Lean and Rocq, high-level inputs undergo a complex elaboration process. The resulting proof term is ultimately checked by a small trusted kernel, but the elaboration process itself is not formally certified, and neither is the kernel.

Faced with these reliability concerns, also expressed by the Leiden Declaration on Artificial Intelligence and Mathematics [1], Hypatia follows a different design. In place of a single elaboration step, the project plans a public tower of dialects. This intended architecture aims to allow the translation from a high-level description of mathematics down to the computational core through a series of certified translations and compilations. Certification of each translation step is planned for future work.

The present thesis focuses on Aristarchus, the verifier for the Sunya dialect, within this broader architecture.

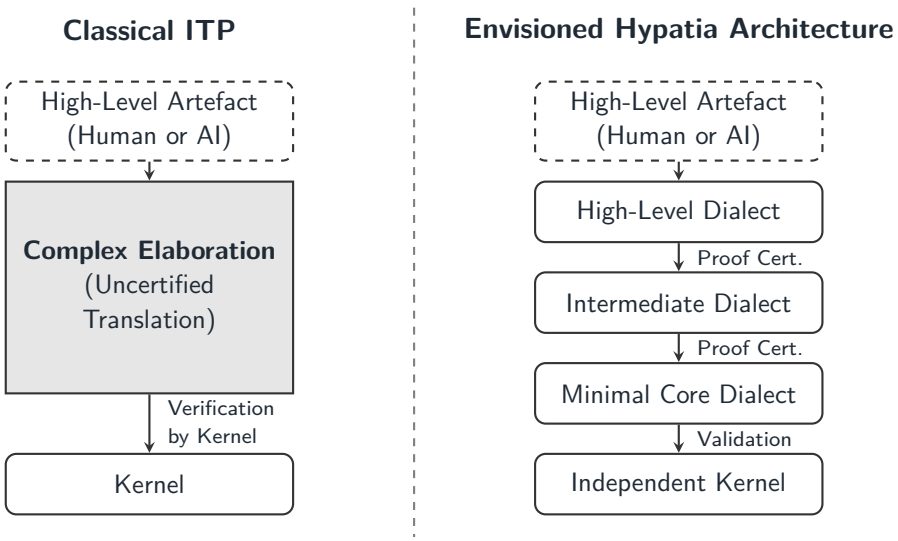

Figure 2.4: Conceptual comparison between the uncertified elaboration pipeline of classical ITPs and the intended architecture of Hypatia's public tower of formal dialects.

Chapter 3

Problem

3.1 Problem Definition

The previous chapter provided some basic information on theorem-proving systems and outlined the advantages of a small kernel, as well as explaining Hypatia's role in this context. Building upon that foundation, this chapter defines the specific problem addressed by this thesis. The objective of this work is to design and implement Aristarchus, the component responsible for verifying documents written in Hypatia . This thesis focuses on an initial version of Aristarchus, which supports the Sunya dialect in Unica mode. In future, this version will be extended to support other dialects and modes as well.

3.1.1 Input, Output, and the Verdict

The system receives the file containing the Hypatia document. This document may be split into several files, so it is necessary to be able to handle multiple inputs, concatenating the context each time.

One of the objectives is to make Hypatia as simple and clear as possible for the user; therefore, the final output that is the verdict, will not simply be accepted or rejected, but will also contain diagnostic messages. We will distinguish between two levels of diagnostics: syntax and semantics. For an accepted document, we will also determine whether it is 'pure', that is, whether no product or derivation remains *tainted*. This means that every product or derivation is genuinely established by at least one valid proof, even if the document happens to contain alternative pseudo-proofs.

3.1.2 What Correctness Means for a Hypatia Document

A Hypatia document is considered correct when it is well-formed according to the active dialect and its declarations and proofs satisfy the rules of the language. The document must therefore be syntactically and lexically valid, use only declared symbols and constructs, and contain admissible proof steps. Each proof must also establish exactly the phrase or schema required by its declaration.

Hypatia does not assume a fixed underlying logic. The language provides the mechanisms for describing and checking a formal system, while its vocabulary, axioms, rules, and derivations are defined by the document itself.

3.1.3 What the Problem Is Not

Hypatia does not search for proofs; it only verifies them against the system defined by the user. This means that inconsistent axioms or rules can still be introduced, and the verifier will accept a proof as long as it is correctly carried out within that system. Here, validity means derivability in the declared system. As an example, consider a system containing the following rule:

$\texttt{\$Regula}$ Bad $\texttt{\$:}$ $\texttt{\${}$ x $\texttt{\$-}$ z $\texttt{\$}\}$ $\texttt{\$-}$ x $\texttt{\$.}$

The rule is not logically sound in general, but it is nevertheless a valid declaration in the system defined by the document. The verifier does not assess the validity of the rule itself; it only checks whether subsequent proof steps use it according to the rules of the Hypatia language.

3.1.4 Solution Approach

The verification process described in this chapter consists of two stages. The document is first parsed into a structured representation of its constructs and their nesting. This representation is then checked against the language rules and the declarations contained in the document. Errors are accumulated during verification, so the process does not stop at the first failure. The final result is the verdict described together with a structured report of the errors.

The two stages address different aspects of correctness. Parsing checks whether the document can be represented according to the language syntax. Verification checks whether the resulting representation satisfies the constraints imposed by the language and by the document.

3.2 Domain Specifications

This chapter describes the specification of the Sunya dialect, based on version `0.0.0.0` of the Hypatia documentation. The version is stated explicitly so that the content of this chapter remains well defined, regardless of any later changes to the specification.

3.2.1 Tower of Dialects

Hypatia defines a public tower of formal dialects. Each dialect builds on the one below it, while higher-level dialects provide syntactic conveniences that are systematically and deterministically reduced (or *desugared*) to the stable, minimal core. Currently, three dialects are defined:

- *Sunya* is the stable minimal core of Hypatia. It is explicit and verbose, making it easy to parse and verify. Sunya is a *grammar-first* dialect, inspired by Metamath, meaning it does not assume any ambient foundational logic or set theory. Instead, its semantics relies on the explicit declaration of symbols and rules, while proofs are validated through strict syntactic identity, using a structural analogue of *modus ponens*.
- *Sifr* acts as a purely syntactic layer over Sunya. It introduces *syntactic sugar* to reduce verbosity without adding any expressive or logical power. Every Sifr construct must be translatable into an equivalent combination of Sunya constructs.
- *Zero* is the top layer of the tower and is designed for complex reasoning. It builds on Sifr, providing a more ergonomic interface and a more *declarative vernacular*.

3.2.2 Bootstrapping and the Self-Formalisation Hierarchy

A characteristic of the Hypatia language domain is its capacity for self-formalisation. The vocabulary, syntax, inference rules, and derivation principles of the language are formalised within Sunya. These documents are the intended base for a future self-certification of the language.

The domain defines these foundational documents as an ordered and cumulative hierarchy of formalisation layers:

- *Elementa:* It provides the foundational symbolic, lexical, and syntactic infrastructure shared by all Hypatia formalisation documents. It specifies a broad and extensible alphabet of atomic symbols, the whitespace and control characters acting as interval markers within the meta language, and the delimiters enclosing object-language strings.
- *Fundamenta:* This layer defines the foundational meta-theory of the language. It introduces the core structural lexicon, mutable variables, and the derivation rules required to manipulate object-language sequences.
- *Scriptum:* This layer formalises the textual infrastructure, defining how whitespace, characters, and textual structures are recognised and composed.
- *Hypatia:* This layer defines the outermost grammar shared by all dialects, formalising the structure of documents, naming conventions, contexts, and proof certificates.
- *Dialect Formalisations:* Finally, the specific constructs of Sunya, Sifr, and Zero are formally encoded on top of this common foundation.

Each layer is nested within the preceding layers and is interpreted using the document context accumulated along the enclosing hierarchy, as illustrated in Figure 3.1. When

alternative formalisations exist, for example when a layer is formalised in both Sunya and Sifr, they occupy the same position in the hierarchy and are not evaluated cumulatively with one another.

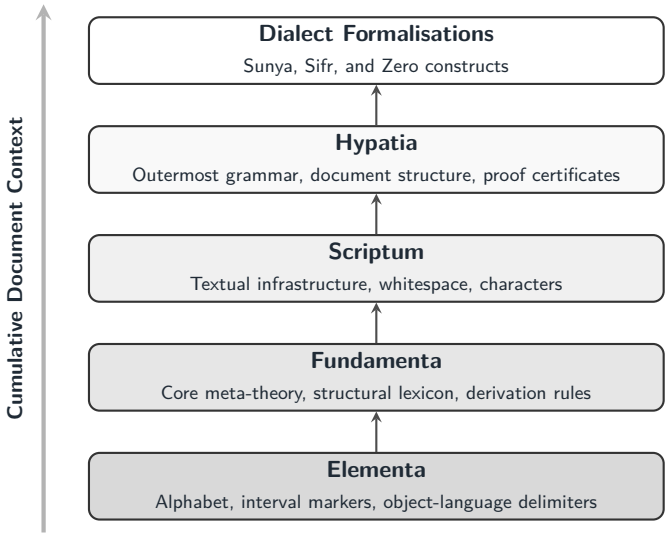

Figure 3.1: The Self-Formalisation Hierarchy of the Hypatia language. Each layer is nested within its predecessors, progressively accumulating the semantic and syntactic context required to bootstrap the system.

3.2.3 Document and Language Structure

Hypatia documents are organised as sequences of well-nested *constructs*, typically introduced by a dollar-prefixed keyword and closed by the delimiter `$.`. The document is interpreted top-down.

At the top level, a document generally consists of a possibly empty sequence of *Hypatia constructs* `HYPATIA(MODE)`, which select the active dialect and may set a new interpretation mode for their nested documents, and *Lingua constructs* `LINGUA(DIALECT, MODE)`, which define named *language fragments*, i.e., formal systems and their subsystems.

Every construct in Hypatia has both a descriptive full name and a standardised canonical form, consisting of the dollar sign followed by exactly two letters.

The domain specification uses a lightweight *Extended Backus-Naur Form* (EBNF) notation to describe the language. Informal text occurring between constructs is treated as a blob or, when explicitly delimited, as a comment, so it is omitted from the production rules.

The grammar uses a few conventions that depend on the context:

- *Terminals:* Literal keywords of the concrete syntax are represented by tokens prefixed with `$`, followed by a Latin word. Words written literally in a production are also terminals, such as `Sunya` and `Unica`.
- *Non-terminals:* Capitalised English words, such as `NAME` and `SYMBOL`, denote syntactic categories. Non-terminals may also be parameterised by contextual values, as in `DOCUMENT(DIALECT, MODE)`.
- *Operators:* Square brackets `[ ... ]` group expressions, while the pipe `|` separates alternatives. The markers `?`, `*`, and `+` denote, respectively, zero or one, zero or more, and one or more occurrences.

Whitespace and structural delimiters are explicitly defined as part of the concrete syntax. Whitespace is formalised in the *Elementa* and *Scriptum* layers and consists of seven symbols, each of which can be referred to by a short alias: `\nh` (nothing), `\sp` (space), `\th` (horizontal tab), `\tv` (vertical tab), `\cn` (carriage return), `\ln` (new line), and `\pn` (new page). Their descriptive aliases are, respectively, `\nihil`, `\spatium`, `\tabulatiohorizontalis`, `\tabulatioverticalis`, `\caputnovum`, `\lineanova`, and `\paginanova`.

Several terminals also have a recurring structural role across the grammar:

- `$.` closes the current construct.
- `$:` opens the body of a named construct.
- `$-` separates a premise from a conclusion within a schema.
- `${` and `$}` delimit a nested schema, allowing compound assertions.
- `$=` introduces the value assigned to a mutable lexeme, while `$,` separates consecutive assignments in a list.
- `$@` qualifies a named reference through its enclosing language fragments.
- `$ex` introduces an evidence or proof section.
- `$per` and `$pr` link a proof step to a previously declared named item.
- `$cum` and `$cm` introduce assignments used to instantiate a proof step.
- `$modo` and `$md` explicitly introduce an interpretation mode.

3.2.4 Host Language, Meta Language, and Object Language

Hypatia is used to define formal-system fragments and to write proof scripts for them. In this sense, Hypatia is the *host language* of the system. Its constructs use a deliberately Latin-based vocabulary, which helps distinguish them from the terminology introduced by the fragment.

A fragment is described by its *phrases* and *schemas*. Phrases contain the fragment text, while schemas define the grammatical rules for that text. Within a phrase, two linguistic levels can be present: the *meta language* and the *object language*.

The meta language consists of the fragment text that occurs outside object-language delimiters. It defines the formal system and provides the language in which it is described and reasoned about. Object-language text, instead, occurs inside the delimiters declared for the object language and corresponds to the language being described. The two kinds of text can occur as different parts of the same phrase.

Object-language delimiters are declared using the `$Inclusores` construct. A matching pair of delimiters identifies the enclosed text as object-language text; text outside the delimiters is treated as meta-language text. Object-language text is then interpreted according to the alphabet declared for the object language. When mutable object-language text is allowed, declared mutable lexemes can also occur as placeholders within that text.

3.2.5 Representations

Hypatia distinguishes two representation levels for its syntactic objects, the *surface* and *internal* representations.

The *surface representation* (s) is the exact finite sequence of source characters forming an instance of a syntactic category. It corresponds literally to what the user writes, whereas

the *internal representation* (r) abstracts away from the surface spelling, retaining only the syntactically relevant structure. Conceptually, it can be viewed as the *Abstract Syntax Tree*. The mapping between surface and internal forms depends on the context Γ, which encapsulates all visible contextual information and internal encodings active at a given point.

To handle potential grammatical ambiguities, the domain formalises this relationship through set-theoretic functions.

Definition 3.1 (Representation functions)**.** Let C be a syntactic category and let Γ be a context. The function

$$IRep_C^\Gamma(s)$$

denotes the set of internal representations that the surface string s can have as an object of C within Γ.

Conversely,

$$SRep_C^\Gamma(r)$$

denotes the set of surface representations that represent r as an object of C in Γ.

For a surface occurrence s of a syntactic category C, let

$$VRep_C^\Gamma(s) \subseteq IRep_C^\Gamma(s)$$

denote the set of internal representations of s that are preliminarily valid as objects of C in the current context Γ, according to the preliminary validity conditions specified for C. When no preliminary validity condition applies to C,

$$VRep_C^\Gamma(s) = IRep_C^\Gamma(s).$$

Moreover, let

$$[[s]]_C^\Gamma$$

denote the semantic value of s as an object of C in the context Γ, whenever the requirements of the active interpretation mode are satisfied.

By definition,

$$i \in IRep_C^\Gamma(s) \quad \Longleftrightarrow \quad s \in SRep_C^\Gamma(i),$$

ensuring structural symmetry between parsing and potential formatting operations. Under the *Unica* interpretation mode,

$$|IRep_C^\Gamma(s)| = 1 \text{ and } VRep_C^\Gamma(s) = IRep_C^\Gamma(s) = \{r\}.$$

The distinction between surface and internal representations applies to syntactic categories of both the Hypatia host language and the fragment language.

Canonical Embeddings and Context Representations Internal representations of compound syntactic objects are constructed compositionally from those of their components. When a syntactic category C_1 is included in a more general category C_2, written $C_1 \subseteq C_2$, their internal representations are not necessarily related by set inclusion. Instead, it is incorporated into the latter through a *canonical embedding*. For example, if an identifier is also considered an expression, its internal representation is wrapped by the constructor

associated with the expression category. The embedding of a category into itself is the identity, and embeddings compose along chains of syntactic inclusions.

The interpretation of syntactic objects depends on a context Γ, which contains the information required to determine their representations and the internal encodings fixed for that context. The context itself may admit multiple internal representations. We denote one such representation by γ. Thus, while Γ describes the abstract contextual environment, γ represents that environment internally as a coherent whole, using the encodings fixed by Γ and preserving the dependencies among its components. Different procedures may produce the same γ but this does not result in different interpretations.

3.2.6 A note on the generality of representations

Internal representations are defined here in general terms, for each surface string, we consider the set of all admissible representations. The implementation focuses on Unica mode, which requires a unique representation, but the parser needs to compute the full set of possible interpretations anyway in order to check for uniqueness, as will become clear in the implementation chapters.

The same infrastructure needed to compute and manage multiple internal representations would also support the Omnes, Valida and Aliqua modes. The parser is already structured so that this work can be reused should those extensions ever be implemented, which is why the discussion keeps a general formalisation even though it concentrates on the Unica regime.

3.2.7 Active Terminals and Escaping

In the language, terminals are context-dependent. A terminal is considered *active* only where the current grammatical production admits it, alongside any enclosing terminals required to delimit the syntactic context. The recognition of active terminals is part of the host syntax and precedes the interpretation of the fragment text. Therefore, whenever an active terminal matches the source text, its recognition takes precedence over fragment content containing the same sequence of characters. This does not change according to the interpretation mode.

To represent the character sequence of an active terminal as fragment content, an additional dollar sign (`$`) must be placed before the dollar sign that would otherwise begin that terminal. It remains part of the surface representation.

This escaping mechanism differs from symbol aliases and escaping mechanisms, which instead are based on the backslash and are used within the fragment's meta-language text, object-language text, blobs, and multiline comments.

3.2.8 Syntax and Semantics

Now, instead, we shall turn our attention to the language itself, defining its syntax and semantics.

Language Parameters

At this level, there are two parameters that govern the context: the first is the active dialect, which determines the syntax and semantics, whilst the second is the interpretation mode, which determines how to handle ambiguities in an internal representation.

```
DIALECT := Sunya | Sifr | Zero
```

The currently supported dialects are: Sunya, Sifr and Zero. The initial dialect is Sunya.

```
MODE := Unica | Omnes | Valida | Aliqua
```

The interpretation mode concerns the internal representations of the context Γ. The context must admit at least one internal representation. Under *Unica*, Γ must have exactly one internal representation, and its validity conditions must hold for that representation.

The grammar also defines other modes, but these have not yet been implemented and will be introduced in future versions of Aristarchus.

Lexical Infrastructure: Names

A NAME is the primitive non-terminal used to name fragments and their declared items. It is an identifier that belongs to the host language and is treated as a single syntactic unit. A NAME consists of a non-empty sequence of admissible Unicode characters, excluding whitespace and the dollar sign.

Document Structure and Dialect Selection

While the general structure of a document relies on nested constructs, its formal syntax is governed by the active interpretation parameters. A `DOCUMENT(DIALECT, MODE)` is defined as a possibly empty sequence of `HYPATIA(MODE)` and `LINGUA(DIALECT, MODE)` constructs:

```
DOCUMENT(DIALECT, MODE) :=
[ HYPATIA(MODE)
| LINGUA(DIALECT, MODE)
]*
```

The evaluation of an outermost `DOCUMENT(DIALECT, MODE)` is performed relative to a document context that is initially empty. It records the top-level language fragments currently available, together with their accumulated fragment-local contexts,including their nested language fragments. The source text of a document may be partitioned across one or more source files, and their order is part of the external input, determining the order of the corresponding portions of source text within that same document. File boundaries have no syntactic or semantic effect, and do not reset or otherwise alter the document context.

Within this structure, the `$Hypatia` construct is responsible for dialect and interpretation selection. It is introduced by `$Hypatia` or `$Hp`, followed by the dialect name DIALECT and, optionally, by `$modo` or `$md` and a new interpretation mode. When the mode is omitted, the currently active mode is inherited.

```
HYPATIA(MODE) :=
[ $Hypatia | $Hp ] DIALECT
[ [ $modo | $md ] MODE' ]? $:
  DOCUMENT(DIALECT, MODE')
$.
```

Semantically, this construct introduces its nested `DOCUMENT(DIALECT, MODE')` without introducing a new document context or a fragment context. The nested document is interpreted within the document and fragment contexts that are currently active. Any persistent changes made there, remain available after the construct has been closed.

Language Structure and Fragment Definition

A `LINGUA(DIALECT, MODE)` construct opens a language fragment identified by a name. It is introduced by `$Lingua` or `$Ln`, followed by the name and its body, given by `LANGUAGE(DIALECT , MODE)`. The body may be empty or may contain a sequence of Hypatia constructs, Lingua constructs, and dialect-specific constructs.

```
LINGUA(DIALECT, MODE) :=
[ $Lingua | $Ln ] NAME $:
  LANGUAGE(DIALECT, MODE)
$.
```

The *fragment context* at a given point consists of the information inherited from the enclosing context together with the *fragment-local context* accumulated within the fragment. The latter contains declarations and other dialect-specific information introduced by the fragment that remain available to subsequent constructs and later occurrences of the same fragment. In particular, it includes the namespace of named phrases and schemas and the fragment's nested language fragments.

Within the context that contains a fragment, a `NAME` identifies a single language fragment. Multiple `LINGUA(DIALECT, MODE)` constructs with the same `NAME` refer to the same fragment. A later occurrence reopens and extends that fragment rather than creating a new one. The identity of a language fragment is independent of the dialect and interpretation mode under which it occurs. A fragment starts with an empty fragment-local context; when it is reopened, its previously accumulated local context is preserved. Its fragment context is determined from the information currently visible in the enclosing context, so a later occurrence can use information that became available there after an earlier occurrence.

Changes to the inherited linguistic environment made within a nested fragment remain local to that fragment and its nested fragments and do not modify the corresponding environment of the enclosing fragment. Named items declared in a nested fragment can nevertheless be accessed from enclosing contexts through qualified references; they remain associated with the fragment in which they were declared.

For example, consider a fragment B nested within a fragment A. If A introduces an item x and B introduces an item y, their local contexts can be represented as

$$\Gamma_A^{\mathrm{loc}} = \{x\}, \qquad \Gamma_B^{\mathrm{loc}} = \{y\}.$$

The context of B includes the information inherited from A, so that

$$\Gamma_B = \Gamma_A \cup \{y\}.$$

However, the declaration of y does not modify Γ_A; it remains local to B and its nested fragments. The item y therefore remains associated with B, while still being accessible from an enclosing context through a qualified reference to B.

The `LANGUAGE(DIALECT, MODE)` body specifies the dialect and the operating mode of the language fragment. It is a possibly empty sequence of constructs: `HYPATIA(MODE)`, `LINGUA(DIALECT, MODE)`, and dialect-specific constructs `CONSTRUCT(DIALECT, MODE)`.

```
LANGUAGE(DIALECT, MODE) :=
[ HYPATIA(MODE)
| LINGUA(DIALECT, MODE)
| CONSTRUCT(DIALECT, MODE)
]*
```

The locally active terminals are the opening terminals of `HYPATIA(MODE)` and `LINGUA(DIALECT, MODE)`, together with the dialect-specific constructs admitted by `CONSTRUCT(DIALECT, MODE)`. The `$.` terminal closing the enclosing `LINGUA(DIALECT, MODE)` remains active.

Informal Text and Comments

Hypatia documents allow arbitrary informal text to be interleaved between formal constructs, it is named *blob*. While a blob does not contribute to the formal logical structure of the document, it remains subject to the active-terminal rules: any Hypatia terminal admitted by the current grammatical context is still recognised and evaluated.

Comments behave differently and disable Hypatia syntax. Terminals appearing inside a comment are treated as ordinary text, so the host-level dollar escaping mechanism no longer applies. The language supports two comment formats:

- `-` introduces a line comment that extends to the end of the current line.
- `/-` and `-/` delimit a multiline comment, which supports nesting.

Comment-opening delimiters are recognised only in host text, where blobs are allowed. When the same character sequences occur inside fragment meta-language or object-language text, they are treated as ordinary fragment content instead of comment delimiters.

A specific backslash escaping mechanism handles literal occurrences of these delimiters. Within a blob, placing a backslash immediately before an opening delimiter (such as `\-` or `\/-`) suppresses its role. The backslash is then removed, leaving the delimiter as part of the blob content. Inside an active multiline comment, the sequences `/-` and `-/` retain their structural role and determine the nesting depth. Prefixing either delimiter with a backslash prevents it from affecting the comment structure, so that `/-` and `-/` are treated as literal text rather than as delimiters.

Alphabetic Infrastructure

The primitive non-terminal `SYMBOL` denotes the set of atomic symbols available to the fragment alphabet. Although a surface representation of a `SYMBOL` may comprise multiple Unicode characters, it is treated as an indivisible atomic unit at the fragment syntax level.

Symbol Aliases and Escaping Certain symbols can be represented by predefined aliases introduced by a backslash. When recognised, an alias yields the same internal representation as its direct surface counterpart. This mechanism is just defined for whitespace characters.

To use a literal source sequence that would otherwise trigger an alias, the initial backslash must be escaped with an additional backslash. Unless it forms a recognised alias or escapes another backslash, the backslash character has no special structural role at this level and represents itself.

The `$Alphabetum` Construct The alphabet of the current fragment is declared or extended using the `$Alphabetum` construct.

```
ALPHABETUM :=
[ $Alphabetum | $Ab ]
  [ SYMBOL ]*
$.
```

Successive declarations are cumulative. Since the fragment alphabet is a set, declaring the exact same `SYMBOL` multiple times has no additional effect.

Within the construct, adjacent symbols are separated by whitespace. This whitespace is consumed by the host syntax and does not become part of the declared symbols. As a direct consequence, any whitespace that actually belongs to the surface representation of a `SYMBOL` must be written using its recognised alias. Its direct surface representation may contribute to represented fragment content, either by itself or as part of a longer surface representation of a `SYMBOL`, only within object-language text.

There, the closing terminal `$.` is locally active. If a declared symbol explicitly contains the sequence `$.`, it must be disambiguated using the dollar escaping mechanism.

The declared alphabet is not required to determine a unique segmentation of subsequent fragment text. The resulting ambiguity is not resolved within the declaration itself, but is deferred to the interpretation of the fragment text.

Internal Representation - Albe For a fixed context Γ, each symbol declared in the fragment alphabet is assigned a unique natural-number identifier. Let

$$enc_\Gamma : SYMBOL_\Gamma \to \mathbb{N}$$

denote this encoding, where $SYMBOL_\Gamma$ is the set of symbols belonging to the alphabet declared in Γ. The corresponding internal alphabet is defined as the image of this encoding:

$$Albe^\Gamma = \mathrm{img}(enc_\Gamma) = \{enc_\Gamma(s) \mid s \in SYMBOL_\Gamma\}.$$

The encoding is injective, so distinct symbols are assigned distinct identifiers. Instead, different surface representations denoting the same symbol are mapped to the same identifier. Hence, the identifier depends on the represented symbol.

The encoding is local to Γ and is not part of the surface syntax of Hypatia. Different encodings may be used for different contexts or implementations without changing the identity of the represented symbols. Repeated declarations of the same symbol likewise refer to the same element of $Albe^\Gamma$.

They provide the internal representations of alphabet symbols relative to Γ and will be used when defining the identity and equality of larger syntactic objects.

The `$Intervalla` Construct The `$Intervalla` construct declares or extends the set of atomic symbols acting as interval markers in the current fragment. Each declared symbol is drawn from the universe denoted by `SYMBOL` and need not belong to the fragment alphabet. As for `ALPHABETUM`, successive interval declarations are cumulative and determine a set, so repeated declarations of the same `SYMBOL` have no additional effect.

```
INTERVALLA :=
[ $Intervalla | $It ]
  [ SYMBOL ]*
$.
```

From a lexical standpoint, the symbols specified within an intervalla declaration are governed by the exact same host-level separation, active-terminal precedence, whitespace, and escaping rules established for the `$Alphabetum` construct.

Let us adopt the following notation: $INTERVAL \subseteq SYMBOL$ denotes a `SYMBOL` declared in the current `INTERVALLA`.

Internal Representation - Intr For the purposes of the internal representation, interval markers are encoded analogously to alphabet symbols. At a fixed context Γ, $Intr^{\Gamma}$ denotes the set of internal identifiers assigned to symbols declared in the current `INTERVALLA`. The encodings of $Albe^{\Gamma}$ and $Intr^{\Gamma}$ are independent: the same `SYMBOL` may therefore identify unrelated elements in the two categories.

The `$Inclusores` Construct The `$Inclusores` construct declares or extends the set of ordered pairs of atomic symbols acting as opening and closing delimiters in the current fragment. Each declared symbol is drawn from the universe denoted by `SYMBOL` and need not belong to the fragment alphabet. As for `ALPHABETUM`, successive `INCLUSORES` declarations are cumulative and determine a set of ordered pairs, so repeated declarations of the same pair have no additional effect.

```
INCLUSORES :=
[ $Inclusores | $Ic ]
  [ SYMBOL SYMBOL ]*
$.
```

The symbols supplied to the construct are read from left to right in consecutive pairs. The first symbol of each pair is the opening delimiter while the second is its corresponding closing delimiter. The opening and closing symbols of a pair may coincide.

A valid `INCLUSORES` declaration must contain an even number of symbols. The symbols within an `INCLUSORES` declaration are subject to the same host-level separation, active-terminal, whitespace, and escaping rules established for `ALPHABETUM` declarations. Let us adopt the following notation: $OPENING - DELIMITER \subseteq SYMBOL$ and $CLOSING - DELIMITER \subseteq SYMBOL$ denote, respectively, the opening and closing `SYMBOL` of a given pair declared in `INCLUSORES`.

Internal Representation - Incl For a fixed context Γ, each ordered delimiter pair is assigned a unique natural-number identifier. The encoding follows the same principles

described for $Albe^{\Gamma}$ and is independent of the encodings used for $Albe^{\Gamma}$ and $Intr^{\Gamma}$. Repeated declarations of the same delimiter pair therefore refer to the same identifier.

The internal representation distinguishes the opening and closing roles of a delimiter pair. Both delimiters of a pair share the same identifier. Their opening or closing role is determined by the corresponding subcategory of $Incl^{\Gamma}$. Consequently, a **SYMBOL** may have different identifiers when it occurs in different delimiter pairs.

Meta Text The **META-TEXT** non-terminal denotes a non-empty, finite sequence of **SYMBOL**s from the alphabet declared for the current fragment. It represents the formal text of the meta-language.

A surface representation is a valid **META-TEXT** if, after resolving symbol aliases and the applicable escape sequences, it can be completely decomposed into declared alphabet symbols.

The decomposition need not be unique. A surface string may correspond to multiple sequences of declared symbols, all of which are considered valid interpretations of that string.

The **META-TEXT** construct does not define any additional escaping rules. It uses the symbol-alias rules already defined for the alphabet, while escaping related to text boundaries is handled by the enclosing **TEXT** construct.

Internal Representation - Meta-Text For a fixed context Γ, each valid segmentation of a `META-TEXT` surface representation gives a non-empty finite sequence of elements of $Albe^{\Gamma}$. This sequence is the corresponding internal representation. For a surface representation s, we denote the set of all such sequences by

$$IRep^{\Gamma}_{\texttt{META-TEXT}}(s).$$

Different segmentations are considered different internal representations only when they result in different sequences. The internal representation does not preserve the original surface spelling, the use of symbol aliases or escape sequences, or any information about how the input was parsed.

For example, suppose that, in a fixed context Γ, the symbols a, b, and ab are represented by 0, 1, and 2, respectively. The surface representation ab can then be segmented either as the single symbol ab or as the two symbols a and b. Its internal representations are therefore

$$[2] \qquad \text{and} \qquad [0, 1].$$

Both sequences represent the same `META-TEXT` surface representation.

Object Text **OBJECT-TEXT** is the context-dependent non-terminal denoting a portion of object-language text enclosed by a matching pair of delimiters declared in **INCLUSORES**.

```
OBJECT-TEXT :=
OPENING-DELIMITER OBJECT-CONTENT(CLOSING-DELIMITER) CLOSING-DELIMITER
```

The **OPENING-DELIMITER** and **CLOSING-DELIMITER** in an **OBJECT-TEXT** belong to the same declared delimiter pair, delimiting the object-language text, but do not belong to its represented object-language content.

The context-dependent non-terminal `OBJECT-CONTENT(CLOSING)` represents a possibly empty finite sequence of `SYMBOL`s from the alphabet declared for the current fragment. Here, `CLOSING` identifies the `CLOSING-DELIMITER` associated with the surrounding `OBJECT-TEXT`. Its surface representation may be empty and, after resolving aliases or escape sequences, must be segmentable into symbols from the declared alphabet. More than one segmentation may be possible.

Whitespace occurring directly in `OBJECT-CONTENT(CLOSING)` may belong to its represented content when the corresponding `SYMBOL` belongs to the declared alphabet. An unescaped occurrence of `CLOSING` terminates the `OBJECT-CONTENT(CLOSING)` and closes the surrounding `OBJECT-TEXT`. To use the same sequence as object-language content, it must be preceded by a backslash. The backslash suppresses the closing role and it is not itself part of the represented content. The escaped occurrence contributes to the content when the corresponding `SYMBOL` belongs to the declared alphabet.

More generally, within `OBJECT-CONTENT(CLOSING)`, a backslash that is not consumed by the alias or escape mechanism may act as an escape marker for the immediately following content.

The marker itself is not part of the represented content and does not need to belong to the declared alphabet. Its position is preserved in the internal representation, allowing an enclosing syntactic category to use it to suppress context-dependent recognition.

When a backslash-prefixed sequence admits more than one valid interpretation, each compatible reading is retained.

If the same `OPENING-DELIMITER` belongs to more than one declared delimiter pair, each pair compatible with the surface representation determines an interpretation of the `OBJECT-TEXT`.

Internal Representation - Object-content For a fixed context Γ, an interpretation of `OBJECT-CONTENT(CLOSING)` is represented by a pair consisting of a finite sequence of elements of $Albe^{\Gamma}$ and a set E of positions indicating where the represented content begins immediately after an escape marker.

For a surface representation s, we define

$$IRep^{\Gamma}_{\text{OBJECT-CONTENT(CLOSING)}}(s) \subseteq (Albe^{\Gamma})^* \times \mathcal{P}(\mathbb{N}),$$

where the first component is the sequence of encoded symbols and the second records the marked positions. For a sequence

$$[a_1, \ldots, a_n],$$

the marked positions satisfy

$$E \subseteq \{1, \ldots, n\}.$$

The empty surface representation corresponds to

$$([], \varnothing).$$

Different segmentations or escaping readings are distinct only when they produce different symbol sequences or different sets of marked positions. Escape markers are not included in the encoded symbol sequence; only the positions they mark are retained.

Internal Representation - Object Text For a fixed context Γ, an interpretation of `OBJECT-TEXT` consists of the identifier assigned by $Incl^{\Gamma}$ to the selected delimiter pair together with the internal representation of its `OBJECT-CONTENT(CLOSING)`.

For a surface representation s, we define the set of valid internal representations as:

$$IRep^{\Gamma}_{\text{OBJECT-TEXT}}(s) \subseteq Incl^{\Gamma} \times \left((Albe^{\Gamma})^{*} \times \mathcal{P}(\mathbb{N}) \right).$$

Alternative surface representations of the same delimiter pair do not by themselves produce distinct internal representations. Different delimiter-pair selections remain distinct when $Incl^{\Gamma}$ assigns them different values.

Fragment Text `TEXT` is the context-dependent non-terminal denoting fragment text. Its empty form is represented explicitly by the distinguished host-language terminal `$VACUUM` / `$Vc`, denoted by `VACUUM`.

```
VACUUM :=
[ $Vacuum | $Vc ]
```

The non-empty form of `TEXT` is instead composed of `META-TEXT`s constituting its meta-language portions and `OBJECT-TEXT`s constituting its object-language portions, with declared intervals separating consecutive text components where required.

```
TEXT :=
[ META-TEXT TEXT-AFTER-META
| OBJECT-TEXT TEXT-AFTER-OBJECT
]
| VACUUM

TEXT-AFTER-META :=
[
  INTERVAL*
  [ INTERVAL META-TEXT TEXT-AFTER-META
  | OBJECT-TEXT TEXT-AFTER-OBJECT ]
]?

TEXT-AFTER-OBJECT :=
[
  INTERVAL*
  [ META-TEXT TEXT-AFTER-META
  | OBJECT-TEXT TEXT-AFTER-OBJECT ]
]?
```

In a non-empty `TEXT`, intervals can occur only between consecutive text components. They are not part of the meta-language or object-language content of the components they separate, but remain structural components of the `TEXT` itself. At least one interval is required between two consecutive `META-TEXT`s, otherwise, intervals are optional. Multiple intervals may occur consecutively; they have the same separating effect and do not create empty components. Each interval is nevertheless retained in the structural decomposition of

the TEXT.

Hypatia whitespace between a TEXT and the surrounding host syntax keeps its host-level role, even when the corresponding SYMBOL has been declared as an INTERVAL or delimiter. Active Hypatia terminals likewise keep the precedence defined by the rules for *active terminals and escaping*, and therefore delimit the surrounding fragment text.

Within a TEXT and outside an OBJECT-TEXT, a direct occurrence recognised as an INTERVAL or OPENING-DELIMITER is not part of a META-TEXT. A recognised symbol alias does not acquire the same structural role merely because it denotes a SYMBOL that has been declared as an INTERVAL or OPENING-DELIMITER. To represent such an occurrence as meta-language content instead, a backslash is placed immediately before its direct surface representation. The backslash suppresses the structural role and does not itself belong to the represented META-TEXT. Clearly, the escaped occurrence can contribute to the META-TEXT only if the corresponding SYMBOL belongs to the declared alphabet.

A backslash-prefixed sequence may have more than one valid reading. For example, it may be interpreted as a symbol alias or as an escape of an INTERVAL or OPENING-DELIMITER. In such cases, all compatible readings are retained. The same applies when a source occurrence matches more than one structural element, for example because overlapping surface representations of declared intervals or opening delimiters are possible, or because the occurrence can be recognised as both an INTERVAL and an OPENING-DELIMITER.

Internal Representation - Text For a fixed context Γ, a TEXT is represented internally as a finite sequence of tagged components. A META-TEXT component contains one of its internal representations, an INTERVAL component contains the identifier assigned to its SYMBOL by $Intr^{\Gamma}$, and an OBJECT-TEXT component contains one of its internal representations. The tag records the syntactic role of the component within the TEXT. The empty TEXT, represented by VACUUM, corresponds to the empty sequence.

For a surface representation s, we define

$$IRep^{\Gamma}_{\mathrm{TEXT}}(s)$$

as the set of all internal sequences obtained from the possible structural readings of s and from the corresponding internal representations of its components. Different readings give distinct elements of this set when their tagged sequences differ. In particular, every occurrence of an INTERVAL is kept in the sequence, so consecutive intervals remain distinct structural occurrences even when $Intr^{\Gamma}$ assigns them the same value. The surface representation of an interval is not retained; only the corresponding identifier is stored. VACUUM likewise does not occur as a component of the internal representation; it represents the absence of structural components.

For example, suppose that | is both a declared INTERVAL and the opening and closing SYMBOL of a declared delimiter pair. If a, b, and c are valid META-TEXTs, the surface representation $a|b|c$ can be read in at least two ways:

$$\texttt{META-TEXT}(a),\ \texttt{INTERVAL}(|),\ \texttt{META-TEXT}(b),\ \texttt{INTERVAL}(|),\ \texttt{META-TEXT}(c),$$
$$\texttt{META-TEXT}(a),\ \texttt{OBJECT-TEXT}(|b|),\ \texttt{META-TEXT}(c).$$

These give different internal representations because the first contains two INTERVAL components, whereas the second contains an OBJECT-TEXT component instead.

Vocabulary Infrastructure

Now we introduce the vocabulary infrastructure that will be used to build the fragment phrases and schemas. Immutable lexemes form the fixed vocabulary of the meta language, whereas mutable lexemes act as placeholders that may be instantiated during proofs.

The `$Lexicon` Construct `$Lexicon` declares or extends the set of immutable lexemes of the current fragment with a possibly empty sequence of `META-TEXT`s. Each declared lexeme becomes a fixed, non-substitutable element of the fragment meta-language vocabulary.

```
LEXICON :=
[ $Lexicon | $Lx ]
  [ META-TEXT ]*
$.
```

The `META-TEXT`s in a `LEXICON` declaration are separated by Hypatia whitespace. This whitespace belongs to the host syntax and does not contribute to the declared lexemes.

Internal Representation - Lexicon For a fixed context Γ, a `LEXICON` declaration adds to $Lexc^{\Gamma}$ all internal representations admitted by the declared surface representation as a `META-TEXT` in the current context. Since $Lexc^{\Gamma}$ is a set, repeated declarations or additions of representations that are already present have no effect.

The declaration is evaluated only in the context in which it occurs. If the context is later extended and the same surface representation admits additional `META-TEXT` representations, these are not added to $Lexc^{\Gamma}$ automatically (*non-retroactivity*). They must be introduced by a new declaration.

Lexemes are identified by their internal representation rather than by their surface form. Therefore, different surface representations that produce the same internal representation refer to the same lexeme. Conversely, if a single surface representation has several valid internal representations, each of them is added to the lexicon as a distinct lexeme.

The surface form and the way a lexeme was parsed are not retained in $Lexc^{\Gamma}$. A declared lexeme can later be used as a component when decomposing a larger `META-TEXT`; it does not have to match the whole text. Lexeme recognition therefore depends only on the internal representation and on whether it belongs to $Lexc^{\Gamma}$.

The `$Mutabilia` Construct `$Mutabilia` declares or extends the set of mutable lexemes of the current fragment with a possibly empty sequence of `META-TEXT`s. Each declared lexeme becomes a placeholder that may be instantiated by an assignment value during proofs.

```
MUTABILIA :=
[ $Mutabilia | $Mt ]
  [ META-TEXT ]*
$.
```

The `META-TEXT`s in a `MUTABILIA` declaration are separated by Hypatia whitespace. This whitespace belongs to the host syntax and does not contribute to the declared lexemes.

Internal Representation - Mutabilia For a fixed context Γ, and for each declared surface representation s, all elements of $IRep^{\Gamma}_{META-TEXT}(s)$ determined in the context active at that point are added to $Mutb^{\Gamma}$. Mutable lexemes follow the same accumulation, identity, non-retroactivity, and surface-independence rules specified above for immutable lexemes and $Lexc^{\Gamma}$. A declared mutable lexeme may subsequently occur as one component of a lexical decomposition of a `META-TEXT` internal representation and may also be recognised as a mutable placeholder within an `OBJECT-TEXT`. Mutable-lexeme identity depends only on internal representation and membership in $Mutb^{\Gamma}$.

We define `MUTABLE-LEXEME` as the context-dependent non-terminal denoting interpretations of a `META-TEXT` whose internal representation, taken as a whole, belongs to $Mutb^{\Gamma}$ and is read with the mutable lexical role, for a surface representation s:

$$IRep^{\Gamma}_{\text{Mutable-Lexeme}}(s) = IRep^{\Gamma}_{\text{Meta-Text}}(s) \cap Mutb^{\Gamma}.$$

The Phrases Fragment phrases are specific interpretations of `TEXT`. A `PHRASE` is the context-dependent syntactic category for refinements of `TEXT` in which meta-language components are decomposed into immutable or mutable lexemes, while object-language components may contain recognised mutable placeholders.

For an internal representation $[a_1, ..., a_n]$ of a `META-TEXT`, a *lexical decomposition* consists of pairwise non-overlapping, consecutive, non-empty subsequences of that sequence, ordered as they occur in it. Each subsequence is assigned either an immutable or a mutable lexical role. The immutable role applies exactly when the subsequence belongs to $Lexc^{\Gamma}$; the mutable role applies exactly when it belongs to $Mutb^{\Gamma}$. A lexical decomposition is *complete* when it covers the entire sequence. If the same internal representation belongs to both $Lexc^{\Gamma}$ and $Mutb^{\Gamma}$, the two roles give distinct lexical readings. Decomposition takes place independently within each `META-TEXT`, so it cannot cross the boundaries between `META-TEXT` components established by `TEXT`. No longest-match rule or other preferred decomposition is imposed. Every complete decomposition is therefore admissible. Each `META-TEXT` component of a `PHRASE` must admit at least one complete lexical decomposition.

Within an `OBJECT-TEXT`, immutable lexemes are not recognised. Mutable lexemes may instead be recognised as placeholders. Let $[a_1, ..., a_n]$ together with $E \subseteq \{1, ..., n\}$ be an internal representation of the `OBJECT-CONTENT` of an `OBJECT-TEXT`. At an unconsumed position i, a consecutive non-empty subsequence beginning at a_i may be recognised as a mutable placeholder exactly when it belongs to $Mutb^{\Gamma}$ and none of its positions belongs to E. Such a placeholder must be recognised whenever one is possible. If several mutable lexemes can be recognised at the same position, each gives a possible reading. The selected subsequence is then consumed as a single mutable placeholder. For `OBJECT-TEXT`, no longest-match rule or other preference is applied, so multiple possible selections give distinct readings. If no mutable placeholder can be recognised at an unconsumed position, that position remains ordinary object-language content. Every position in E must belong to at least one consecutive non-empty subsequence in $Mutb^{\Gamma}$ that could otherwise be recognised as a mutable placeholder. In this way, every retained escape marker suppresses at least one possible mutable-placeholder recognition. An escape-marker reading that has no such effect does not contribute a `PHRASE` internal representation.

A surface representation of `TEXT` is also a surface representation of `PHRASE` exactly when at least one of its `TEXT` internal representations admits such a refinement.

$$s \text{ is a surface representation of } PHRASE \iff \exists t \in IRep^{\Gamma}_{TEXT}(s) \text{ such that } t \text{ admits a refinement to } PHRASE.$$

A `TEXT` internal representation is therefore discarded at the `PHRASE` level if at least one of its `META-TEXT` components has no complete lexical decomposition or one of its `OBJECT-TEXT` components has no valid mutable-placeholder reading.

`VACUUM` represents the empty `PHRASE` and contains no lexeme or object-language content.

The internal logical structure of a `PHRASE` is left to the formal system encoded by the fragment.

Internal Representation - Phrase At a fixed context Γ, an internal representation of a `PHRASE` is obtained by refining one internal representation of the underlying `TEXT`. The structure of the `TEXT` is preserved during this refinement. Each `META-TEXT` component is replaced by one of its complete lexical decompositions, together with the immutable or mutable role assigned to each lexical component. An `INTERVAL` component remains unchanged. Each `OBJECT-TEXT` component is replaced by one of its valid mutable-placeholder readings, while its delimiter-pair identifier and represented object-language content are preserved. The resulting representation also records each occurrence recognised as a mutable placeholder and the corresponding element of $Mutb^{\Gamma}$. Once an escape marker has been taken into account when selecting the mutable-placeholder reading, its marked position is no longer retained. The empty internal representation of `TEXT` gives the empty internal representation of `PHRASE`.

For a surface representation s, $IRep^{\Gamma}_{\mathrm{PHRASE}}(s)$ is the set of all refined representations obtained from the elements of $IRep^{\Gamma}_{\mathrm{TEXT}}(s)$, together with their valid lexical decompositions and mutable-placeholder readings. Different choices give different elements of this set when they result in different refined representations. Different derivation or parsing routes that produce the same refined representation are represented by a single element of $IRep^{\Gamma}_{\mathrm{PHRASE}}(s)$. Since marked positions are not retained, two `TEXT` internal representations that differ only in their use of escaping may result in the same `PHRASE` internal representation when they produce the same lexical decomposition and mutable-placeholder structure.

Mutabilia For a surface representation s and an internal representation $r \in IRep^{\Gamma}_{\mathrm{PHRASE}}(s)$, $mut(r)$ denotes the set of mutable lexemes occurring in r. A mutable lexeme occurs when an element of $Mutb^{\Gamma}$ is selected either with the mutable lexical role in a `META-TEXT` or as a mutable placeholder in an `OBJECT-TEXT`. The same element of $Mutb^{\Gamma}$ identifies the same mutable lexeme in both cases, regardless of its surface representation. Since $mut(r)$ is a set, repeated occurrences of the same mutable lexeme are counted only once. In particular, $mut(r) = \varnothing$ for the empty `PHRASE` represented by `VACUUM`.

Phrase Safety An internal representation $r \in IRep^{\Gamma}_{\mathrm{PHRASE}}(s)$ is *safe* exactly when

$$mut(r) = \varnothing.$$

Validity of a `PHRASE` internal representation is exactly phrase safety. Therefore,

$$VRep^{\Gamma}_{\mathrm{PHRASE}}(s) = \{r \in IRep^{\Gamma}_{\mathrm{PHRASE}}(s) \mid mut(r) = \varnothing\}.$$

Safety is a property of an internal representation of **PHRASE**; it does not restrict the internal representations admitted by the category itself. Both safe and unsafe representations may therefore belong to $IRep^{\Gamma}_{\text{PHRASE}}(s)$ whenever the surface representation admits them. A construct that requires a phrase to be independently reusable may impose phrase safety as a preliminary validity condition, according to the active interpretation mode.

Schemas A **SCHEMA** captures Hypatia's core inferential pattern *("because … therefore …")*: it relates a premise part to a conclusion part using the terminal **$-**. An **ASSERTION** is either a **PHRASE** or an explicitly nested schema delimited by **${** and **$}**.

```
SCHEMA :=
ASSERTION $- ASSERTION

ASSERTION :=
PHRASE
| ${ SCHEMA $}
```

Internal Representation - Assertion and Schema At a fixed context Γ, the internal representations of **ASSERTION** and **SCHEMA** are defined recursively from the internal representations of their components.

An internal representation of an **ASSERTION** is a tagged value that distinguishes its two syntactic forms. When the assertion is a **PHRASE**, the value contains an internal representation of that **PHRASE**. This provides the canonical embedding of **PHRASE** representations into **ASSERTION** representations. When the assertion has the form **${** S **$}**, the value contains an internal representation of the **SCHEMA** S and a tag identifying it as a nested-schema assertion. This tagged representation is determined by the internal representation of S through the nested-schema form of **ASSERTION**. The **${** and **$}** terminals are not retained. The tag distinguishes this **ASSERTION** representation from the internal representation of S as a **SCHEMA**.

An internal representation of a **SCHEMA** of the form **P $- C** is an ordered pair containing the internal representations of its premise assertion **ASSERTION** P and conclusion assertion **ASSERTION** C. Their order preserves the premise and conclusion roles. The **$-** terminal is not retained.

For a surface representation s, $IRep^{\Gamma}_{\text{ASSERTION}}(s)$ and $IRep^{\Gamma}_{\text{SCHEMA}}(s)$ contain all representations obtained from the internal representations of their components in this way. Different readings of an **ASSERTION** give distinct elements when they produce different tagged values. For a **SCHEMA**, distinct readings give different elements when they produce different ordered pairs. Different derivation or parsing routes that produce the same internal representation contribute only one element to the corresponding set.

For example, suppose that the premise and conclusion of a **SCHEMA** admit the following internal representations:

$$IRep^{\Gamma}_{\text{ASSERTION}}(P) = \{p_1, p_2\}, \qquad IRep^{\Gamma}_{\text{ASSERTION}}(C) = \{c_1, c_2\}.$$

The possible internal representations of the **SCHEMA** are then

$$(p_1, c_1), \quad (p_1, c_2), \quad (p_2, c_1), \quad (p_2, c_2).$$

Each pair represents one possible reading of the **SCHEMA**.

Free and Active Mutabilia The function mut introduced for **PHRASE** internal representations also applies to **ASSERTION** and **SCHEMA** internal representations. The corresponding schema interfaces are described by two additional sets of mutable lexemes, *free mutabilia* and *active mutabilia*, through the functions free and act.

For an internal representation a of an **ASSERTION** carrying a **PHRASE** representation r, the three sets coincide:

- $\mathrm{mut}(a) = \mathrm{mut}(r)$;
- $\mathrm{free}(a) = \mathrm{mut}(r)$;
- $\mathrm{act}(a) = \mathrm{mut}(r)$.

For an internal representation a of a nested-schema assertion containing a **SCHEMA** internal representation r, the definitions are inherited from r:

- $\mathrm{mut}(a) = \mathrm{mut}(r)$;
- $\mathrm{free}(a) = \mathrm{free}(r)$;
- $\mathrm{act}(a) = \mathrm{act}(r)$.

For a **SCHEMA** representation $r = (a_P, a_C)$, with a_P and a_C representing its premise and conclusion assertions, we have:

- $\mathrm{mut}(r) = \mathrm{mut}(a_P) \cup \mathrm{mut}(a_C)$;
- $\mathrm{free}(r) = \mathrm{free}(a_C) \setminus \mathrm{mut}(a_P)$;
- $\mathrm{act}(r) = \mathrm{free}(a_P) \cup \mathrm{act}(a_C)$.

Here, $\mathrm{mut}(r)$ contains all mutable lexemes occurring in the schema, whereas $\mathrm{free}(r)$ contains the mutable lexemes that occur freely in the conclusion but do not occur in the premise, and hence are exposed at its outer boundary. Instead, the active set records the mutable interface available for instantiation: the free interface of a schema premise contributes immediately, while activity is followed recursively through its conclusion. As a result, a nested schema can have an empty free set while still having non-empty act.

For example, consider the schema

```
${ xIz $- Uy $} $- ${ xU $- xy $}
```

and an internal representation r with the displayed mutable-lexeme reading. Let a_P and a_C denote its premise and conclusion assertion representations, respectively. Then:

- $\mathrm{mut}(a_P) = \{x, y, z\}$, $\mathrm{free}(a_P) = \{y\}$, and $\mathrm{act}(a_P) = \{x, y, z\}$;
- $\mathrm{mut}(a_C) = \{x, y\}$, $\mathrm{free}(a_C) = \{y\}$, and $\mathrm{act}(a_C) = \{x, y\}$.

It follows that:

- $\mathrm{mut}(r) = \{x, y, z\}$;
- $\mathrm{free}(r) = \varnothing$;

- $\mathrm{act}(r) = \{x, y\}$.

Thus, $\mathrm{free}(r) \subsetneq \mathrm{act}(r) \subsetneq \mathrm{mut}(r)$. The mutable lexeme y occurs freely in the conclusion but is already present in the outer premise. The lexeme x remains active through the conclusion, but it is not free there because the inner premise xU supplies it. The lexeme z occurs in the schema but does not remain active at its outer boundary.

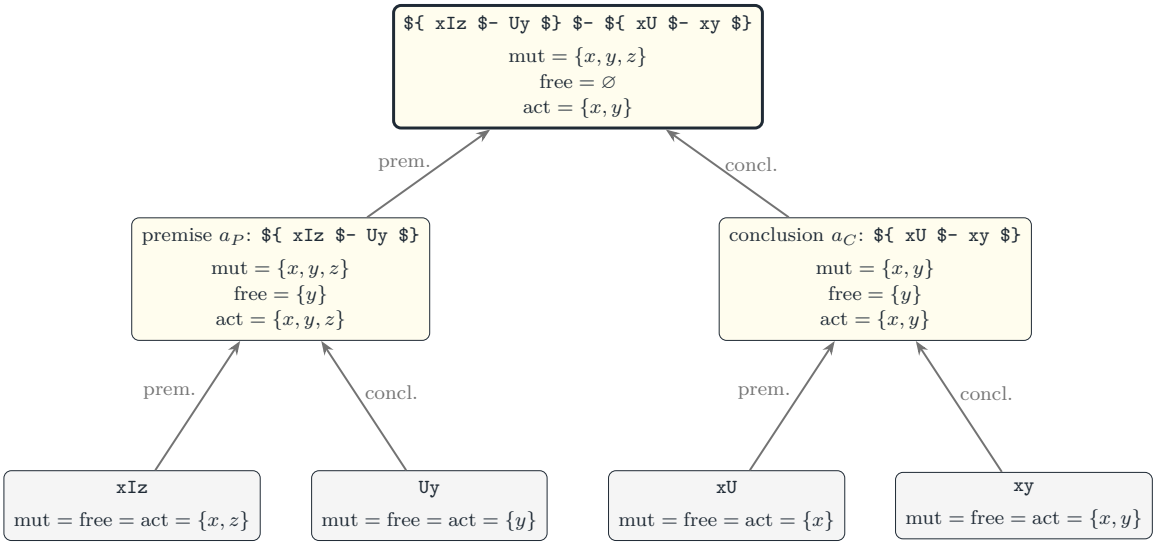

Figure 3.2: Bottom-up computation on the example. On the `PHRASE` the three sets coincide; the arrows go up to the node that combines them.

Schema Safety An internal representation $r \in IRep^{\Gamma}_{SCHEMA}(s)$ with $r = (a_P, a_C)$ of a surface representation s of a schema is safe exactly when

$$free(r) = \varnothing \iff free(a_C) \subseteq mut(a_P).$$

In other words, every mutable lexeme that occurs freely in the conclusion must also occur in the premise.

Schema safety applies to internal representations of `SCHEMA`; it does not restrict the representations admitted by the category itself. Both safe and unsafe representations can therefore belong to $IRep^{\Gamma}_{SCHEMA}(s)$ when they are admitted by the surface representation. A construct that requires a schema to be independently reusable may then impose schema safety as a preliminary validity condition, according to the active interpretation mode.

Validity of a `SCHEMA` internal representation is exactly schema safety. Therefore,

$$VRep^{\Gamma}_{\mathrm{SCHEMA}}(s) = \{r \in IRep^{\Gamma}_{\mathrm{SCHEMA}}(s) \mid free(r) = \varnothing\}.$$

Assignments An `ASSIGNMENTS` expression is a non-empty list of assignments separated by "`$,`", each of which maps one or more mutable lexemes to the same assignment value using the terminal `$=`.

```
ASSIGNMENTS :=
ASSIGNMENT [ $, ASSIGNMENT ]*

ASSIGNMENT :=
[ MUTABLE-LEXEME ]+ $= TEXT
```

In an **ASSIGNMENT** construct, the **MUTABLE-LEXEME** occurrences on the left-hand side are separated by Hypatia whitespace. This whitespace belongs to the host syntax and does not contribute to the mutable lexemes.

The right-hand side of an assignment specifies its value through a **TEXT** construct. This value is the fragment, which may also be empty, to be substituted for the mutable lexemes on the left-hand side. For a non-empty assignment value, only the structural requirements of the **TEXT** non-terminal apply. Its **META-TEXT**s do not need to admit lexical decompositions into declared lexemes, and mutable placeholders are neither evaluated nor recognised within its **OBJECT-TEXT**s. Evaluation is therefore deferred until the assignment is applied.

An empty assignment value can be written using the **VACUUM** construct. In this case, the substitution contributes no fragment text.

When several assignments occur in the same **ASSIGNMENTS** expression, they are applied simultaneously. This has two consequences:

1. *Explicit Modification:* Only mutable lexemes that occur on the left-hand side are modified. Mutable lexemes not listed there remain unchanged; no additional assignments are inferred from the structure of the text.

2. *Value Immunity:* The value specified on the right-hand side of an assignment is not affected by any other assignment in the same **ASSIGNMENTS** expression. Substitution does not proceed from one assignment to another, so cascading substitutions do not occur.

Internal Representation - Assignment Consider an **ASSIGNMENT** with surface form $m_1 \dots m_k\$ = v$, where $m_1, \dots, m_k$ are its **MUTABLE-LEXEME** occurrences and v is the surface representation of its assignment value. Its internal representation consists of a non-empty set $U = \{u_1, \dots, u_k\}$, with $u_i \in IRep^{\Gamma}_{\text{MUTABLE-LEXEME}}(m_i)$ for each i, together with a replacement surface fragment w determined by v. When v represents **VACUUM**, w is the empty sequence of source characters; otherwise, $w = v$.

For a surface representation s of an **ASSIGNMENT**, $IRep^{\Gamma}_{\text{ASSIGNMENT}}(s)$ contains all pairs (U, w) obtained in this way. Different choices of internal representations for the **MUTABLE-LEXEME** occurrences give different elements only when they result in different pairs (U, w). The replacement surface fragment is kept because assignment application operates by surface substitution. Apart from the conversion of **VACUUM** to the empty fragment, its surface spelling is preserved.

The internal representations of the assignment value as a **TEXT** are not retained and do not produce different internal representations of the **ASSIGNMENT**. At this stage, **TEXT** is used only to determine whether v is an admissible assignment value. Its fragment structure is determined again after substitution.

Repeated occurrences of the same mutable lexeme within an **ASSIGNMENT** do not add any new information. The same applies to the order of the mutable lexemes on the left-hand side, which is not preserved.

Internal Representation - `Assignments` For a surface representation s of an **ASSIGNMENTS** expression consisting of **ASSIGNMENT** surface representations $s_1, \ldots, s_n$, an internal representation of s is any set

$$q = \{q_1, \ldots, q_n\}$$

such that

$$q_i \in \mathit{IRep}^{\Gamma}_{\text{ASSIGNMENT}}(s_i) \qquad \text{for every } 1 \leq i \leq n.$$

The set $\mathit{IRep}^{\Gamma}_{\text{ASSIGNMENTS}}(s)$ contains all sets obtained in this way. Since the internal representation is a set, the order of the assignments is not retained.

An internal representation q of **ASSIGNMENTS** is *compatible* exactly when

$$w_1 = w_2,$$

whenever (U_1, w_1) and (U_2, w_2) belong to q and

$$U_1 \cap U_2 \neq \varnothing.$$

Validity of an **ASSIGNMENTS** internal representation is exactly this compatibility condition. Hence, for a surface representation s of **ASSIGNMENTS**,

$$\mathit{VRep}^{\Gamma}_{\text{ASSIGNMENTS}}(s) = \left\{ q \in \mathit{IRep}^{\Gamma}_{\text{ASSIGNMENTS}}(s) \mid q \text{ is compatible} \right\}.$$

The **ASSIGNMENT**, instead, has no additional validity condition, so

$$\mathit{VRep}^{\Gamma}_{\text{ASSIGNMENT}}(s) = \mathit{IRep}^{\Gamma}_{\text{ASSIGNMENT}}(s).$$

An internal representation q is valid exactly when its represented assignments are compatible. That is, for any

$$(U_1, w_1), (U_2, w_2) \in q,$$

if

$$U_1 \cap U_2 \neq \varnothing,$$

then

$$w_1 = w_2.$$

Each valid internal representation q determines a finite assignment function $\mathit{asg}(q)$. Its domain is

$$\operatorname{dom}(\mathit{asg}(q)) = \bigcup_{(U,w) \in q} U,$$

and, for every

$$u \in \operatorname{dom}(\mathit{asg}(q)),$$

the value $\mathit{asg}(q)(u)$ is the unique w such that

$$(U, w) \in q \qquad \text{and} \qquad u \in U.$$

The compatibility condition guarantees that this value is unique.

Assignment validity is a condition on an internal representation of **ASSIGNMENTS**, rather than a restriction on the internal representations admitted by the category. Hence, if a surface

ASSIGNMENTS representation admits both valid and invalid internal representations, both remain elements of $IRep^{\Gamma}_{\text{ASSIGNMENTS}}(s)$ whenever admitted by the surface representation, while only the valid representations belong to $VRep^{\Gamma}_{\text{ASSIGNMENTS}}(s)$.

Assignment Application Assignment application is defined relative to both a surface representation and a selected internal representation of that surface. Let s be a surface representation of a **PHRASE** and let $r \in IRep^{\Gamma}_{\text{PHRASE}}(s)$ be one of its internal representations. A *mutable surface factorisation* of r over s writes s as

$$v_0 m_1 v_1 \dots m_k v_k = s$$

together with mutable lexemes $u_1, \dots, u_k$, where each pair (m_i, u_i) corresponds to one of the mutable occurrences represented in r. Here, m_i is the exact portion of the surface representation consumed for that occurrence, while $u_i \in Mutb^{\Gamma}$ is the corresponding mutable lexeme. The portions $v_0, \dots, v_k$ contain all remaining source characters, so the m_i and v_i form a partition of s without overlap. A portion m_i includes every source character used to represent the occurrence, including symbol aliases and escaping characters, even when they are not part of the represented fragment content.

If r contains no mutable occurrence, then $k = 0$ and the factorisation consists only of $v_0 = s$. For fixed s and r, the factorisation is unique. Different readings that produce the same r therefore induce the same partition of s and the same sequence of mutable lexemes. The factorisation is used only during assignment application and is not part of the internal representation.

Let q be a valid internal representation of an **ASSIGNMENTS** expression. For each mutable surface factorisation, define

$$m'_i = \begin{cases} asg(q)(u_i), & \text{if } u_i \in \text{dom}(asg(q)), \\ m_i, & \text{otherwise.} \end{cases}$$

Applying q to the factorisation then produces the surface sequence

$$v_0 m'_1 v_1 \dots m'_k v_k.$$

Thus, every occurrence associated with a mutable lexeme in the domain of $asg(q)$ is replaced by the corresponding replacement surface fragment, while all other source material remains unchanged. The replacements are applied simultaneously, so a replacement fragment is not modified again by another assignment in the same application.

If the resulting surface sequence is empty, it is represented by the fixed **VACUUM** surface form **$Vc** for the purpose of subsequent syntactic interpretation. This does not restrict the surface forms accepted by a construct applying the assignments, since **$Vc** and **$Vacuum** determine the same empty **PHRASE** internal representation. Otherwise, the resulting surface sequence is kept unchanged. This sequence is the result of applying q to r with respect to s.

The same operation extends recursively from **PHRASE** to **ASSERTION** and **SCHEMA**. For an **ASSERTION** represented by a **PHRASE**, assignment application is performed on the corresponding **PHRASE** surface representation. For a nested-schema assertion, it is applied recursively to the enclosed **SCHEMA**, while the surface material that delimits the schema as an assertion remains unchanged.

For a **SCHEMA** surface representation s with internal representation

$$r = (a_P, a_C) \in IRep^{\Gamma}_{\text{SCHEMA}}(s),$$

the surface uniquely determines the premise and conclusion assertion representations. The same q is applied recursively to both. The resulting assertion surfaces are then placed back into the original schema surface, preserving the **$-** separator and all source material outside the two assertions. Hence, for fixed s, r, and q, the operation produces a unique resulting surface representation.

After assignment application, the resulting surface representation is interpreted again as the syntactic category required by the construct that invoked the assignments. The lexical and mutable-placeholder structure of the original representation is not carried over to this new interpretation, nor is the component structure used during assignment application. Only the replacement surface fragments supplied by the assignments are inserted.

Assignment application itself does not restrict the mutable lexemes that may belong to $\text{dom}(asg(q))$. Any such restriction, including conditions based on activity or proof-item rigidity, is imposed by the construct that invokes the operation.

Theory Infrastructure

The theory of a fragment consists of the named safe phrases and schemas declared within it. Each declaration introduces or extends a named item, identified within its declaring fragment by its **NAME** and declaration kind. A named item carries semantic content consisting of one or more internal representations of a **PHRASE** or **SCHEMA**.

```
REFERENCE :=
NAME [ $@ NAME ]*
```

Identity and Overloading Within a fragment, the combination of declaration kind and **NAME** identifies a single named item. Items declared in different fragments remain distinct even when they have the same kind and **NAME**.

Visibility in Nested Fragments A named item becomes available after its declaration is closed. When a nested fragment closes, items declared within it become accessible from the enclosing fragment.

Linguistic Environment Isolation An item becoming accessible from an enclosing fragment does not make the linguistic environment of its declaring fragment available there. In particular, declarations such as **$Alphabetum** or **$Inclusores** remain associated with the fragment in which they were made. Access to a named item therefore does not imply access to the environment in which that item was declared.

References References are resolved within the declaration kind required by the construct in which the **REFERENCE** occurs.

An *unqualified reference* contains only a **NAME**. Resolution starts from the current fragment and then moves through its enclosing fragments. The first item declared directly with that **NAME** is selected, so an item declared in a more deeply nested fragment takes precedence

over homonymous item in an enclosing fragment. Items declared in other fragments are not considered, even if they can be reached through qualified references. If no matching item is found, the reference is invalid.

A *qualified reference* contains at least one `$@`. To resolve such a reference, the *full qualification chain* of each accessible item of the required declaration kind is considered. The chain starts with the item's `NAME`, followed by the `NAME` of its declaring fragment and then those of the enclosing fragments up to the outermost one. A qualified reference matches an item when its sequence is a prefix of this chain. Each `NAME` following `$@` therefore refers to the fragment immediately enclosing the entity named before it. The reference is valid only if this prefix identifies exactly one accessible item. An insufficient qualification does not introduce interpretation ambiguity and is not governed by the active interpretation mode. The qualification can be partial, provided that it is sufficient to identify a single item.

For example, if the current fragment B is nested within A and both directly declare an item P of the same declaration kind, the unqualified reference P within B resolves to the item declared in B, while P `$@` A resolves to the item declared in A. If two accessible items of the same declaration kind instead have the distinct full qualification chains P `$@` C `$@` B `$@` A and P `$@` C `$@` A, then P `$@` C is insufficiently qualified and therefore invalid, while P `$@` C `$@` B and P `$@` C `$@` A resolve uniquely to the first and second item, respectively.

For a declaration kind K, let ***K-REFERENCE*** denote a `REFERENCE` interpreted as a reference to a named item of kind K. For a surface representation n of such a reference, if n resolves in Γ to an available named item of kind K with semantic content E, then

$$[[n]]^{\Gamma}_{\textit{K-REFERENCE}} = E;$$

otherwise, $[[n]]^{\Gamma}_{\textit{K-REFERENCE}}$ is undefined.

Semantic Content of Named Items The body of a named declaration with surface representation s is interpreted as a `PHRASE` for a `PRINCIPIUM` or `PRODUCTUM`, and as a `SCHEMA` for a `REGULA` or `DERIVATIO`. The declaration kind does not introduce a distinct representation of the body. Thus,

$$\mathit{IRep}^{\Gamma}_{\textsc{Principium}}(s) = \mathit{IRep}^{\Gamma}_{\textsc{Productum}}(s) = \mathit{IRep}^{\Gamma}_{\textsc{Phrase}}(s),$$

$$\mathit{VRep}^{\Gamma}_{\textsc{Principium}}(s) = \mathit{VRep}^{\Gamma}_{\textsc{Productum}}(s) = \mathit{VRep}^{\Gamma}_{\textsc{Phrase}}(s),$$

and

$$\mathit{IRep}^{\Gamma}_{\textsc{Regula}}(s) = \mathit{IRep}^{\Gamma}_{\textsc{Derivatio}}(s) = \mathit{IRep}^{\Gamma}_{\textsc{Schema}}(s),$$

$$\mathit{VRep}^{\Gamma}_{\textsc{Regula}}(s) = \mathit{VRep}^{\Gamma}_{\textsc{Derivatio}}(s) = \mathit{VRep}^{\Gamma}_{\textsc{Schema}}(s).$$

A `PRINCIPIUM` or `REGULA` imposes no further semantic obligation. Consequently,

$$[[s]]^{\Gamma}_{\textsc{Principium}} = [[s]]^{\Gamma}_{\textsc{Phrase}}$$

and

$$[[s]]^{\Gamma}_{\textsc{Regula}} = [[s]]^{\Gamma}_{\textsc{Schema}}.$$

A `PRODUCTUM` or `DERIVATIO` additionally requires every proof supplied by the declaration to be successful, according to the rules defined in *Proof Infrastructure*. Under the *Unica*

interpretation mode, when this proof obligation is satisfied, the semantic value of the declaration is the unique valid internal representation. Thus,

$$[[s]]^{\Gamma}_{\textsc{Productum}} = [[s]]^{\Gamma}_{\textsc{Phrase}}$$

and

$$[[s]]^{\Gamma}_{\textsc{Derivatio}} = [[s]]^{\Gamma}_{\textsc{Schema}}.$$

For a new named item, a successful declaration introduces the item and associates its semantic value with it. A failed declaration does not introduce or reserve the corresponding **NAME**. A later successful declaration may therefore introduce the same named item. For a subsequent declaration of the same kind and **NAME**, the semantic value already associated with the item must be preserved. A repeated **PRINCIPIUM** or **REGULA** is therefore semantically redundant, while repeated **PRODUCTUM** or **DERIVATIO** may supply additional proof scripts, provided that the resulting semantic value remains unchanged.

Under the *Unica* interpretation mode, successive declarations may use different surface representations or contexts, provided that they produce the same semantic value. The proofs supplied by a declaration are interpreted in the context of that declaration and according to the active interpretation mode. They are used to determine whether the declaration succeeds and are not part of the named item, and need not be retained after the declaration has been checked. If a subsequent declaration fails, the semantic value previously associated with the named item remains unchanged.

At a high level, two axes distinguish the declarations introduced here:

- *what is declared*, i.e., either a named safe phrase (**PRINCIPIUM** and **PRODUCTUM**) or a named safe schema (**REGULA** and **DERIVATIO**);
- *how it is justified*: principles and rules are accepted as given, whereas products and derivations must be justified by explicit proof scripts introduced by the **$ex** terminal.

The $Principium construct **$Principium** declares a named principle, whose body is a **PHRASE** that is accepted as given within the fragment and therefore requires no proof.

```
PRINCIPIUM :=
[ $Principium | $Pn ] NAME $:
  PHRASE
$.
```

The $Regula construct **$Regula** declares a named rule, whose body is a **SCHEMA** that is accepted as given within the fragment and therefore requires no proof.

```
REGULA :=
[ $Regula | $Rg ] NAME $:
  SCHEMA
$.
```

The $Productum construct The **$Productum** construct declares a *named product*, whose body is a **PHRASE** justified by one or more explicit phrase proofs. Each **PHRASE-PROOF** provides an alternative proof.

```
PRODUCTUM :=
[ $Productum | $Pd ] NAME $:
  PHRASE
$ex
  [ PHRASE-PROOF ]+
$.

PHRASE-PROOF :=
[ $Probatione | $Pb ]
  PHRASE-PROOF-SCRIPT
$.
```

The proof must not depend, directly or transitively, on the product it justifies.

The `$Derivatio` construct The `$Derivatio` construct introduces a named derivation by associating a `SCHEMA` with one or more explicit schema proofs. Each `SCHEMA-PROOF` provides an alternative proof of the same declared `SCHEMA`. A proof establishes the schema as a whole by reducing it to a residual target after zero or more outermost premises have been exposed through `$Praemitto`. The resulting interpretations must satisfy the safety and interpretation-mode requirements defined for `SCHEMA`.

```
DERIVATIO :=
[ $Derivatio | $Dr ] NAME $:
  SCHEMA
$ex
  [ SCHEMA-PROOF ]+
$.

SCHEMA-PROOF :=
[ $Probatione | $Pb ]
  SCHEMA-PROOF-SCRIPT
$.
```

Here too, the proof must not depend, directly or transitively, on the derivation it justifies.

Proof Infrastructure The proof semantics presented in this section is restricted to *Unica*, where each proof step has at most one successor proof state.

Sunya distinguishes two kinds of proof scripts:

- `PHRASE-PROOF-SCRIPT`s, which justify the `PHRASE` declared by a `PRODUCTUM`;
- `SCHEMA-PROOF-SCRIPT`s, which justify the `SCHEMA` declared by a `DERIVATIO`.

Both kinds consist of ordered sequences of proof steps, each terminated by the `$.` terminal. Within a `PRODUCTUM` or `DERIVATIO` declaration, the `$ex` terminal introduces the evidence section, and each proof is enclosed by `$Probatione`.

Proof States A proof is interpreted relative to the context Γ and the body of the declaration it justifies. Under *Unica*, a proof script

$$\Pi = s_1, \ldots, s_n$$

is interpreted as a linear sequence of proof states

$$p_0 \xrightarrow{s_1} p_1 \xrightarrow{s_2} \cdots \xrightarrow{s_n} p_n.$$

The initial proof state p_0 has an empty proof context; for a `SCHEMA-PROOF-SCRIPT`, the premise stack starts out empty too.

Each proof state records what has been accumulated along the corresponding prefix of the proof script. This includes an ordered proof context made up of the proof items established by the preceding steps, where each item records the surface representation of the expression it establishes, one selected internal representation of that expression, and any proof-context metadata tied to that establishment. A schema proof state carries, in addition, an ordered premise stack recording the premises introduced by `$Praemitto`; a phrase proof state carries no such stack.

Given a proof state p_i reached before step s_{i+1}, the successor state is determined by the partial function

$$Succ^{\Gamma}(s_{i+1}, p_i).$$

Under *Unica*, this function, wherever defined, returns exactly one successor proof state. If it is undefined at any state reached along the way, the proof fails and no terminal result is obtained; otherwise the interpretation proceeds linearly until the final step, yielding a single terminal proof state p_n and hence a single terminal result.

For a phrase proof, the terminal result comes from the expression established by the final proof item. For a schema proof, the premise stack is used to reconstruct the terminal `SCHEMA`. Either way, the terminal result is then compared against the internal representation of the body declared by the enclosing `PRODUCTUM` or `DERIVATIO`.

Terminal Result Under the *Unica* interpretation mode, a proof script that reaches the final proof state p_n produces a single terminal result. The terminal result is determined from the last proof item and, for a `SCHEMA-PROOF-SCRIPT`, from the premises recorded in the premise stack.

For a `PHRASE-PROOF-SCRIPT`, the terminal result is the internal representation carried by the last proof item, provided that this representation is a `PHRASE`. Thus, if the proof

completes successfully, the terminal result is the internal representation established by the last proof step.

For a `SCHEMA-PROOF-SCRIPT`, if the premise stack is empty, the terminal result is the internal representation carried by the last proof item, provided that it is a `SCHEMA`. If the premise stack is non-empty, the last proof item may establish either a `PHRASE` or a `SCHEMA`. The former is first embedded as a phrase assertion, while the latter is embedded as a nested-schema assertion. The stored premises are then restored in reverse order, each becoming the premise of a new schema. After the premise stack has been exhausted, the resulting `SCHEMA` is the terminal result of the proof.

Let $Out(\Lambda)$ denote the set of terminal results produced by the proof object Λ. Under *Unica*, the proof object has only one branch and therefore, whenever the proof completes successfully,

$$Out(\Lambda) = \{r\}$$

for some internal representation r.

Let s be the surface representation of the declaration body and let C denote its category: `PHRASE` for a `PRODUCTUM` and `SCHEMA` for a `DERIVATIO`. Under *Unica*, the semantic value $[\![s]\!]^{\Gamma}_{C}$ is determined independently of the proof. If

$$[\![s]\!]^{\Gamma}_{C} = r,$$

then the proof is successful exactly when its unique terminal result is r, that is,

$$Out(\Lambda) = \{r\}.$$

Consequently, a `PRODUCTUM` or `DERIVATIO` is successfully justified only when every proof supplied by the declaration produces the same terminal result as the semantic value of its body.

Proof Dependencies and Non-Circularity A proof script depends directly on every principle, product, rule, or derivation referenced by one of its proof steps. This dependency is read off from the resolved references occurring in the script, and does not depend on how the underlying proof object branches. For non-circularity, only dependencies on products and derivations actually require recursive justification, since principles and rules are simply accepted as given and so terminate the dependency chain there. A proof script justifying a product or derivation is non-circular precisely when each of its direct dependencies on a product or derivation can itself be justified recursively, by choosing a proof of the referenced item that is already available at the point where that reference occurs; the resulting justification must then be finite and must never depend on the very item being proved. In particular, a proof may not reference, directly, the product or derivation it is justifying. Non-circularity is required for a proof to succeed.

Proof Status and Purity A proof is either *genuine* or a *pseudo-proof*. Its status is determined when the proof is completed and does not change afterwards. A proof is genuine if it contains no `$Divino` step and every reference to a product or derivation resolves to an item that is genuinely established at that point. Otherwise, the proof is a pseudo-proof.

Principles and rules are always genuinely established because their declarations are accepted as given. For products and derivations, the status depends on the proofs available

for the corresponding item. The item is genuinely established if at least one available proof is genuine. If all available proofs are pseudo-proofs, the item is *tainted*.

An item's status may change as the document is interpreted. A product or derivation that is initially tainted becomes genuinely established if a later declaration provides a genuine proof. Once an item is genuinely established, subsequent pseudo-proofs do not make it tainted again.

A reference uses the status of the referenced item at the point where the reference occurs, so if a proof refers to a tainted item, that proof remains a pseudo-proof even if a later declaration establishes the referenced item genuinely. The later declaration affects only subsequent references.

A document is *pure* if no product or derivation is tainted. It may therefore contain pseudo-proofs and still be pure, provided that every product or derivation with a pseudo-proof also has at least one genuine proof.

Phrase-Proof Scripts

A phrase-proof script is a non-empty sequence of proof steps.

```
PHRASE-PROOF-SCRIPT :=
[ PHRASE-PROOF-STEP ]+
```

Every proof item established in a *phrase proof* must carry a valid internal representation of its category. Under the *Unica* interpretation mode, each expression written by a proof step must admit exactly one internal representation satisfying the conditions of the current proof context.

For a phrase proof, these conditions are the standard validity conditions of the expression's category, namely *Phrase Safety* or *Schema Safety*. When the same proof steps are reused in a schema proof, additional conditions imposed by the proof context may apply.

The Phrase-Proof-Step `$Assumo` The `$Assumo` step establishes in the local proof context the `PHRASE` or `SCHEMA` associated with an available named item. It can refer to a principle or product to assume a `PHRASE`, or to a rule or derivation to assume a `SCHEMA`.

```
[ $Assumo | $Am ]
  PHRASE
[ $per | $pr ]
  [ $Principium | $Pn | $Productum | $Pd ]
  REFERENCE
$.

[ $Assumo | $Am ]
  SCHEMA
[ $per | $pr ]
  [ $Regulam | $Rg | $Derivationem | $Dr ]
  REFERENCE
$.
```

Let n be the surface representation of the `REFERENCE` and K the declaration kind indicated

by the step. If

$$[\![n]\!]^{\Gamma}_{\text{K-REFERENCE}}$$

is undefined, the step cannot be performed. Otherwise, let

$$E = [\![n]\!]^{\Gamma}_{\text{K-REFERENCE}}.$$

Let s be the surface representation of the expression written by the step and C its category. Under the *Unica* interpretation mode, the step is defined only when

$$VRep^{\Gamma}_{C}(s) = \{r\}$$

for a unique internal representation r. The step then establishes s with representation r in the local proof context, provided that

$$\uparrow E = \{r\}.$$

The Phrase-Proof-Step `$Obtineo` `$Obtineo` establishes an instance of the `SCHEMA` carried by the immediately preceding proof item, using explicit `ASSIGNMENTS` introduced by `$cum`.

```
[ $Obtineo | $Ot ]
  SCHEMA
[ $cum | $cm ]
  ASSIGNMENTS
$.
```

Let A be the `SCHEMA` carried by the preceding proof item and let r be its selected internal representation. The assignments introduced by the step can only be applied to active mutabilia of A: an assignment is applicable when every mutable lexeme on its left-hand side belongs to $act_r(A)$, and it then replaces all occurrences of that lexeme in the interpretation of A (activity being a property of mutable lexemes relative to a fixed internal representation). The assignments are applied simultaneously, following the rules of `ASSIGNMENTS`, and this produces a fragment of text from the interpretation of A, which is then interpreted afresh as a `SCHEMA`. Under *Unica*, exactly one assignment route must be applicable, and the resulting fragment must admit exactly one valid internal representation r'. Let B be the `SCHEMA` written by `$Obtineo`: the step is only defined when r' is also a valid internal representation of B, in which case r' becomes the internal representation of the new proof item.

The Phrase-Proof-Step `$Consequor` The `$Consequor` step adds to the proof context a `PHRASE` or `SCHEMA` obtained as a direct consequence of the two immediately preceding proof items. It corresponds to *modus ponens*. The most recent proof item must carry a `SCHEMA` of the form `P $- C`, while the preceding item must establish the expression represented by the premise P.

```
[ $Consequor | $Cn ]
  [ PHRASE | SCHEMA ]
$.
```

Let r be the internal representation carried by the penultimate proof item, and let (a_P, a_C) be the internal representation of the **SCHEMA** carried by the most recent item, where a_P and a_C are its premise and conclusion assertions respectively. The penultimate item matches a_P when it carries the representation that assertion requires: a **PHRASE** representation r if a_P is a phrase assertion, or a **SCHEMA** representation r if it is a nested-schema assertion. If the premise is not matched, the step cannot be performed; otherwise, let r_C be the internal representation carried by the conclusion assertion, which comes from the preceding **SCHEMA** rather than being chosen independently by **$Consequor**. Let s be the surface representation written by **$Consequor**. Under *Unica*, the step is only defined if s admits the unique valid internal representation r_C, in which case it establishes s with representation r_C and adds it to the proof context.

The Phrase-Proof-Step `$Reitero` The **$Reitero** step reintroduces a **PHRASE** or **SCHEMA** already established in the current proof context, typically to make it available among the most recent proof items for a subsequent step.

```
[ $Reitero | $Rt ]
  [ PHRASE | SCHEMA ]
$.
```

Let s be the surface representation written by **$Reitero** and C its category. Under *Unica*, s must have a unique valid internal representation in $VRep_C^\Gamma(s)$. A proof item matches the step if its selected internal representation has category C and is one of the valid representations of s. The item is then re-established with the same representation and with its proof-context metadata preserved.

$Reitero succeeds only if the matching items give rise to a single successor state. This state is then added to the proof context.

The Phrase-Proof-Step `$Divino` **$Divino** establishes a conjectured phrase or schema without requiring it to follow from previously established proof items.

```
[ $Divino | $Dv ]
  [ PHRASE | SCHEMA ]
$.
```

The **$Divino** step relaxes the requirement of derivability, but not the structural well-formedness property: the conjectured item must still satisfy the conditions of that context.

Let s be the surface representation written by **$Divino** and C its category. Under *Unica*, the step is defined only when

$$VRep_C^\Gamma(s) = \{r\}$$

for a unique internal representation r. In this case, let p_r be the successor state obtained by establishing s with representation r. Thus,

$$Succ^\Gamma = p_r.$$

Schema-Proof Scripts

A schema-proof script is a non-empty sequence of proof steps

```
SCHEMA-PROOF-SCRIPT :=
[ SCHEMA-PROOF-STEP ]+
```

A **SCHEMA-PROOF-STEP** is either a **PHRASE-PROOF-STEP**, interpreted under the additional conditions of a schema proof, or a **$Praemitto** step.

Unlike a phrase proof, a schema proof carries an additional premise stack. The stack is extended only by **$Praemitto** and is not discharged while the proof script is being interpreted. Its assertions are discharged when the terminal result of the proof is reconstructed, as specified in *Terminal Results*.

Contextual Well-Formedness Let $T = [a_1, \ldots, a_k]$ be the current premise stack, where each a_i is the selected internal representation of an **ASSERTION** introduced by a **$Praemitto** step. Define

$$mut(T) = \bigcup_{i=1}^{k} mut(a_i),$$

with $mut(T) = \varnothing$ when T is empty.

In a schema proof, the premise stack determines the mutabilia available to the current proof step. A **PHRASE** is admissible only if all mutabilia in its internal representation belong to $mut(T)$. For a **SCHEMA**, all free mutabilia must belong to $mut(T)$. Therefore, when a **PHRASE-PROOF-STEP** is used in a schema proof, its internal representation must satisfy these additional contextual conditions, while the other conditions specific to the step remain unchanged.

The same condition is applied to the internal representations obtained by **$Obtineo** after assignments have been applied. Hence, contextual well-formedness is distinct from the independent safety conditions of a **PHRASE** or **SCHEMA**. A **PHRASE** may contain mutabilia as long as they are supplied by the current premise stack, and a **SCHEMA** may have free mutabilia as long as they are supplied by that stack. When T is empty, these conditions reduce to the usual *Phrase Safety* and *Schema Safety* conditions, respectively.

The Schema-Proof-Step $Praemitto The **$Praemitto** step introduces a new premise into the current schema proof and establishes the expression represented by that premise as a proof item.

```
[ $Praemitto | $Pm ]
  [ PHRASE | SCHEMA ]
$.
```

Let s be the surface representation written by **$Praemitto** and C its category. Under *Unica*, the step is defined only when

$$IRep_C^{\Gamma}(s) = \{r\}$$

for a unique internal representation r.

The premise is represented by an **ASSERTION** constructed from r: a canonical phrase assertion when C is **PHRASE**, or a canonical nested-schema assertion when C is **SCHEMA**. The resulting assertion is appended to the current premise stack, and s is established as a proof

item with representation r.

Unlike other proof steps used in a schema proof, **$Praemitto** is not subject to the contextual well-formedness condition determined by the current premise stack. This is because the step introduces a new premise rather than establishing an expression under the premises already present.

The introduced premise is not checked against the **SCHEMA** justified by the enclosing **$Derivatio**. The relationship between the premises introduced by the proof and the justified **SCHEMA** is checked when the terminal result of the proof is reconstructed.

Rigid Mutabilia Each proof item in a schema proof carries a set of rigid mutabilia, denoted by $rig(i)$. Rigidity is a property of the proof item, rather than of its selected internal representation alone. Therefore, two proof items carrying the same representation may have different rigid sets.

If a proof item i carries a **PHRASE** representation r, then

$$rig(i) = mut(r).$$

If it carries a **SCHEMA** representation r, then

$$free(r) \subseteq rig(i) \subseteq mut(r).$$

Hence, every mutabile of a phrase established in a proof is rigid, while a schema may also retain rigid mutabilia inherited from the way in which it was established.

The elementary proof steps determine and propagate rigidity as follows.

- **$Praemitto** gives a newly established **PHRASE** representation r the rigid set $rig(i) = mut(r)$. For a **SCHEMA** representation r, it gives $rig(i) = free(r)$.
- **$Assumo** establishes every proof item with

  $$rig(i) = \varnothing.$$

  This is possible because named phrases and schemas are safe independently of the current proof context.
- For **$Obtineo**, let i_A be the preceding proof item and r its selected **SCHEMA** representation. Only active mutabilia that are not rigid in i_A may be instantiated:

  $$dom(asg(q)) \subseteq act(r) \setminus rig(i_A).$$

  If an admissible assignment route q produces a fresh **SCHEMA** representation r', the new proof item has

  $$rig(i) = free(r') \cup \big(rig(i_A) \cap mut(r')\big) \cup dep_q(T, r').$$

  The first term makes the free mutabilia of the resulting schema rigid. The second preserves rigidity inherited from the preceding proof item when the corresponding mutabilia are still present. The last term records dependencies introduced by the assignment values.
- For **$Consequor**, let i_A and i_S be the penultimate and most recent proof items,

respectively, and let r_C be the internal representation carried by the conclusion assertion of the schema carried by i_S. If r_C is a **PHRASE** representation,

$$rig(i) = mut(r_C).$$

If r_C is a **SCHEMA** representation,

$$rig(i) = free(r_C) \cup \big((rig(i_A) \cup rig(i_S)) \cap mut(r_C)\big).$$

- **$Reitero** preserves the rigid set of the proof item that it re-establishes.
- **$Divino** gives a **PHRASE** representation r the rigid set

$$rig(i) = mut(r),$$

and a **SCHEMA** representation r the rigid set

$$rig(i) = free(r).$$

To define the dependency component used by the **$Obtineo** step, let r' be the unique fresh **SCHEMA** internal representation produced by the assignment application. The dependency set $dep(T, r')$ is determined from r' together with the transient provenance of the surface representation produced by assignment application.

A mutable lexeme belongs to $dep(T, r')$ exactly when it belongs to $mut(T)$ and at least one of its occurrences in r' is represented by surface material originating, at least in part, from an applied assignment value. Consequently,

$$dep(T, r') \subseteq mut(T) \cap mut(r').$$

If inserted material combines with surrounding material, the dependency is determined by the mutable lexeme recognised in the fresh interpretation r', rather than by what the assignment value might have represented in isolation. This provenance information is transient: it is discarded once the rigid set of the newly established proof item has been determined.

Example. Consider the following proof script:

```
$Praemitto xUIIIy $.
$Assumo xIIIy $- xUy $per $Regulam III $.
$Obtineo xUIIIy $- xUUy $cum x $= xU $.
$Reitero xUIIIy $.
$Reitero xUIIIy $- xUUy $.
$Consequor xUUy $.
```

After **$Praemitto**, the premise stack contains the phrase assertion carrying $xUIIIy$. Its free and mutable interfaces are both $\{x, y\}$, so the proof item established by **$Praemitto** carries rigid set

$$rig(i) = \{x, y\}.$$

The **$Assumo** step establishes Rule III with empty rigid set. It can therefore be instantiated by **$Obtineo** with the assignment

$$x\$ = xU.$$

In the resulting fresh interpretation, x belongs to $mut(T)$ and occurs in surface material contributed by the assignment value. Hence, for the selected fresh representation r' of $xUIIIy $- xUUy$,

$$dep_q(T, r') = \{x\}.$$

Since

$$free(r') = \varnothing,$$

the resulting proof item carries rigid set

$$rig(i) = \{x\}.$$

The two **$Reitero** steps preserve the rigid sets $\{x, y\}$ and $\{x\}$, respectively. Finally, **$Consequor** establishes the **PHRASE** $xUUy$. Its mutable set is

$$mut(xUUy) = \{x, y\},$$

so the new proof item carries rigid set

$$rig(i) = \{x, y\}.$$

Chapter 4

Design

In this chapter, we move on to the design of Aristarchus. The question at this stage is: how should the system be organised so that it can fulfil its requirements?

4.1 Requirements

4.1.1 Functional Requirements

Functional requirements define the core behaviour and the explicit operations the system must be able to perform. The specific functional requirements for the system are listed in Table 4.1.

Table 4.1: Functional requirements

ID	Requirement
FR1	The system shall accept a document, produce its verdict, and report the level at which verification fails for a rejected document.
FR2	The system shall construct a structured representation of the document that preserves all information required by subsequent checks.
FR3	The system shall check alphabet declarations, including symbols, separators, and delimiter pairs, together with the consistency of the roles assigned to them.
FR4	The system shall check declarations of immutable and mutable lexemes and reject any word that cannot be formed from vocabulary declared and in force at that point in the document.
FR5	The system shall check that the statement of every named item is well-formed with respect to the vocabulary in force, and that every schema made independently reusable by a declaration is safe.
FR6	The system shall check every proof step against the forms of proof step admitted by the dialect and against the state produced by the preceding steps.
FR7	The system shall check that each proof script establishes exactly the statement declared for it. For schema proofs, the comparison shall be performed after discharging the introduced premises.
FR8	The system shall resolve qualified and unqualified references to named items across nested fragments, and reject both references that identify no item and references that identify more than one item.
FR9	The system shall reject a proof that depends, directly or indirectly, on the item it is intended to justify.
FR10	The system shall determine, for each product and derivation, whether it is genuinely established or tainted, propagate this status through references, and report whether the document as a whole is pure.
FR11	The system shall report the diagnostics produced during verification.

4.1.2 Non-Functional Requirements

Non-functional requirements specify the quality attributes, design constraints, and architectural principles that the system must satisfy. Rather than describing specific actions, they dictate *how* the verifier must operate to ensure reliability and correctness of the Hypatia language. The non-functional requirements are listed in Table 4.2.

Table 4.2: Non-functional requirements

ID	Requirement
NFR1	**Soundness.** The system shall never accept an incorrect document. When the verifier cannot establish that a document is correct, it shall prefer rejection over acceptance.
NFR2	**Completeness.** The system shall accept every correct document that satisfies the specified semantics.
NFR3	**De Bruijn criterion.** The system shall explicitly identify its *Trusted Code Base*, comprising solely the AST, the parser, and the logical validator, on which the correctness of the verdict depends. This core shall remain small enough to be manually inspectable, excluding proof search algorithms, automated tactics, and diagnostic rendering. In future, this core shall be formalised and mathematically certified.
NFR4	**Diagnostic completeness.** The verifier shall report all problems that it can establish during a single execution rather than stopping at the first failure.
NFR5	**Diagnostic quality.** Each diagnostic shall identify what went wrong, where it occurred, and the context in which it occurred. A single underlying cause shall not produce redundant diagnostics at every location affected by it.
NFR6	**Determinism and termination.** The system shall terminate for every input and shall produce the same verdict for the same document.
NFR7	**Reproducibility.** Given the same document, composed of the exact same ordered sequence of files, the system shall produce the same verdict regardless of the execution environment or any configuration options that do not alter the semantics.
NFR8	**Logic independence.** The verification component shall not depend on a predefined logic or on a fixed set of axioms and inference rules.

4.2 System Architecture

4.2.1 The Command Line Interface

The system interfaces with the user through the command line. An execution is started by invoking the main tool followed by the validation operation and a list of ordered source files. The command syntax takes the following form:

```
hypatia validator <filePath>+
```

Where `<filePath>` is a string identifying the path of a source file.

The system receives the specified files and starts the reading and verification process. Multiple files listed in the same invocation are treated as a *single unified document*. If a file cannot be opened, the tool reports an error and exits with code 4.

The order in which the files are given matters: they are treated as consecutive parts of the same document, and the context carries over from one file to the next. Declarations made in an earlier file can therefore remain available in later ones, subject to the usual scope and qualification rules for fragments.

4.2.2 Overview: From Document to Verdict

Aristarchus is organised as a *pipeline* composed of two main stages: parsing and validation. Both stages use contextual information that is updated during the processing of the document. In Sunya, the alphabet and the vocabulary can change while the document is read.

The architecture therefore comprises three main elements: the Abstract Syntax Tree (AST), the parser, and the verifier. The parser builds the AST from the source document, while the verifier traverses its structure and checks the properties required by the language. The overall flow is illustrated in Figure 4.1.

The parser

The parser has the task of reading the source document and transforming it into the Abstract Syntax Tree. Due to the nature of Hypatia, the parser maintains a mutable vocabulary state that changes as declarations are read.

During reading, it must determine which active-terminals are available. This information is used to recognise and interpret the different constructs according to the linguistic environment defined up to that point.

The parser also checks the interpretation mode requested by the input file. If a mode other than *Unica*, such as *Omnes*, *Aliqua* or *Valida* is requested, it reports that this mode is not implemented and interrupts the processing.

The interpretation mode determines how a construct with more than one valid interpretation is handled. Under *Unica*, the parser reports an ambiguity and processing fails. When the interpretation is unique, the recognised construct is converted into the corresponding representation in the AST.

The verifier

Once the AST has been constructed, control passes to the verifier. At this stage, the system no longer operates directly on the source text, but traverses the tree according to a *top-down*

approach, maintaining an environment that contains the information available at the current point in the document.

The environment records, among other things, the active vocabulary, the language fragments currently in scope, and the theorems declared up to that point. During the traversal, this information is updated according to the constructs encountered.

The verifier checks that assertions and schemas use only symbols and words available in the current linguistic environment or among the declared mutabilia. It also verifies the logical correctness of derivations and the absence of circular dependencies between theorems.

Verdict, diagnostics, and purity

The validation process does not stop at the first error, so that the diagnostics produced during the traversal are instead collected, allowing multiple problems to be reported in the same execution.

At the end of the pipeline, three distinct types of output are produced:

1. **Verdict:** represents the overall outcome of the processing. The document is rejected if fatal errors are present among the collected diagnostics, including those of a syntactic or semantic nature.

2. **Diagnostic:** consists of a structured report of the errors, warnings, and other anomalies detected during the processing, associated with their respective positions in the source document.

3. **Purity:** is a property that describes the dependence of the verified theorems on conjectures introduced through the `$Divino` construct.

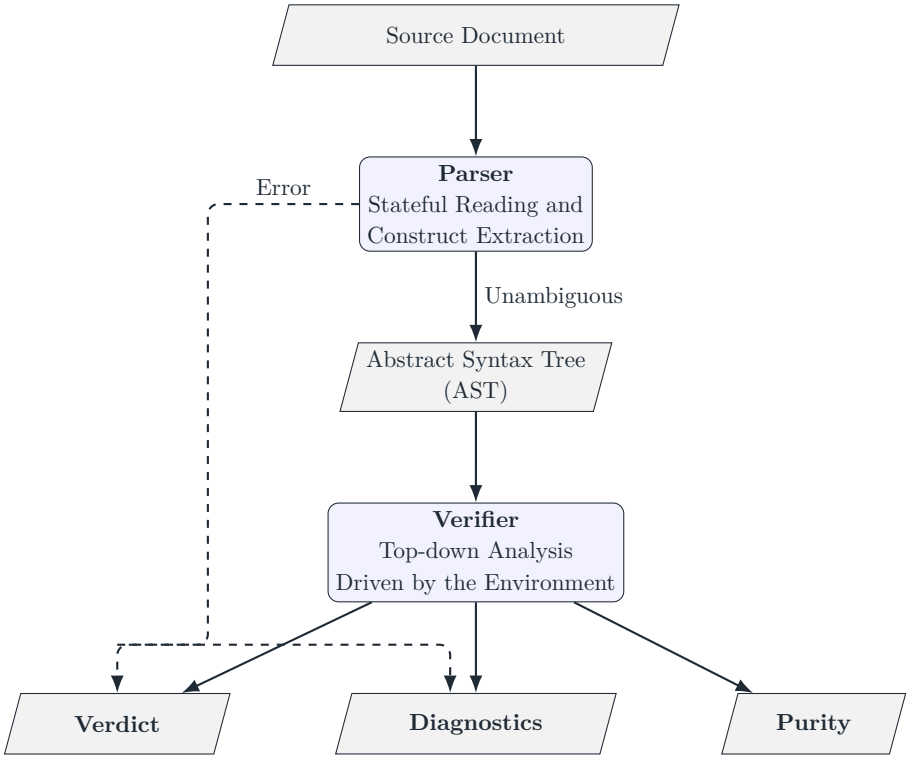

Figure 4.1: High-level architecture of the Aristarchus pipeline. The process is divided between resolving the dynamic structure in the Parser and logical validation in the Verifier. Parsing errors bypass intermediate stages to immediately report a negative verdict and structural diagnostics.

4.3 The Trusted Code Base

To satisfy the de Bruijn criterion (NFR2), Aristarchus isolates the components on whose correctness a reliable verdict depends. This minimal set forms the Trusted Code Base of the system.

The TCB consists of:

- The underlying runtime environment (GHC).
- The *AST* module: an error in the AST affects every check that reads it.
- The *Parser* module: it must guarantee the correct interpretation of word boundaries, ambiguity rejections, and escape sequences.
- The *Validator* module (excluding diagnostic rendering), the logical core responsible for schema safety, derivation validity, and purity checks.

The Command Line Interface and the diagnostic reporting system, by contrast, are not part of the TCB. The rendering of errors has no bearing on the verification logic: a bug in the reporting module cannot cause an invalid mathematical document to be accepted.

4.4 Abstract Syntax Tree Modelling

4.4.1 What the Representation Is For

One of the design decisions consists in determining which information should be preserved after parsing.

In this case, the decision was to represent the information required for validation and not to preserve information that is not used. In this way, the content that can influence the result of the validation is limited to what has actual meaning for the document.

4.4.2 What It Discards

Therefore, not all the information present in the source document is preserved.

For example, **informal prose** is excluded. A Hypatia document may contain free text between one construct and another, but this text has no formal meaning and is not used during validation. The same applies to **comments** and **whitespace**.

In addition, the constructs of the language can be written using either their extended name or their canonical form; likewise, a symbol can be represented directly or through its corresponding alias. In both cases, the AST preserves what has been represented and not the form used to write it.

It follows that two documents that differ only in notation produce the same structured representation and, consequently, the same validation result. A purely notational difference therefore cannot modify the verdict.

4.4.3 Object Text Delimiters

The object text is delimited by a *pair of delimiters* declared in the document, and the same document may define more than one. One might think that the delimiters are merely a simple writing convention, as is the case with the characters that delimit a string and that, in general, are not part of its content.

However, in Hypatia, the pair of delimiters contributes to the identity of the object text. Two texts with the same content, but delimited by different pairs, therefore represent distinct objects. A proof step that treats such objects as equivalent must therefore be considered incorrect. Since this information is necessary for the identity of the object, it was decided to include it in the AST.

4.4.4 The Shape of a Document

The structure of the AST reflects the hierarchical structure of a Hypatia document. A document specifies the dialect in which it is written and may contain language fragments and nested documents; each fragment contains constructs and may in turn contain fragments and documents. Since each document declares its own dialect, a nested document may also change it at any depth of the tree. Constructs are divided into declarations that define the vocabulary and declarations of named items; the latter include the item's statement and, when required, its proof scripts. Figure 4.2 summarises this organisation.

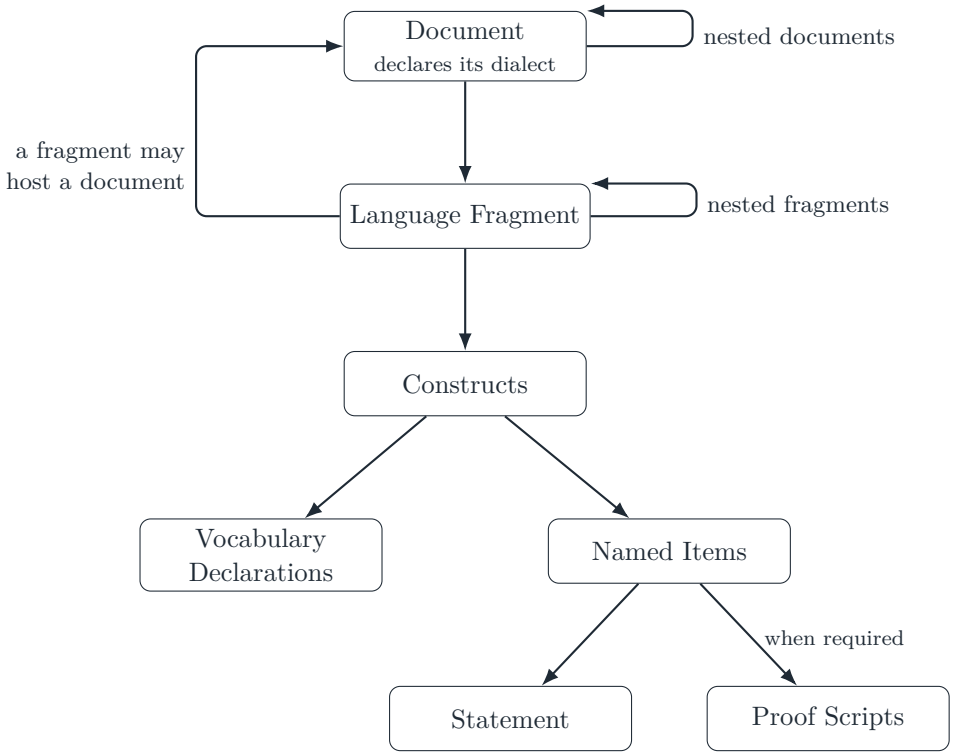

Figure 4.2: Conceptual structure of a Hypatia document represented by the AST. Documents and language fragments may contain one another, so a nested document — and with it a change of dialect — can occur at any depth.

4.4.5 Statements: Phrases, Object Text, and Schemas

What an item declares and what a proof step establishes is an assertion. An assertion can be either a **phrase** or a **schema**, consisting of a premise and a conclusion.

The structure of the schema is recursive, because both the premise and the conclusion are assertions and can therefore in turn be schemas.

In the case of a **phrase**, it consists of a sequence of segments. A segment can be a **word** belonging to the vocabulary or an **object text**. The latter is delimited by one of the pairs of delimiters declared in the document and contains symbols and, where applicable, **placeholders** that can be replaced during instantiation.

The distinction between the two types of segment is maintained in the AST because they must be interpreted according to different rules during validation and instantiation. In this way, the verifier does not have to reconstruct the distinction from the representation.

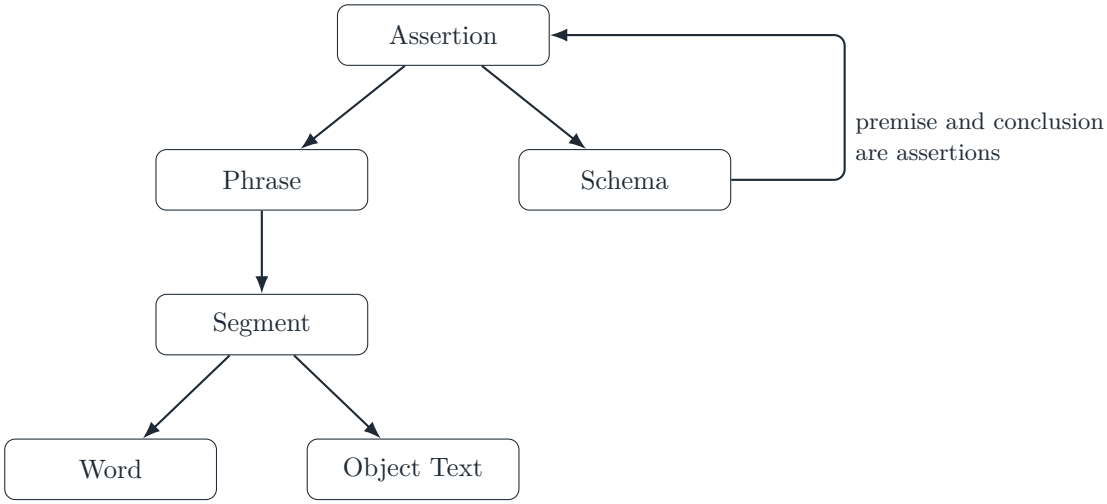

Figure 4.3: Conceptual structure of assertions and their components. An assertion is either a phrase or a schema; a schema is composed of two assertions, which is what allows nested implications.

4.4.6 Proof Scripts

A proof script is represented as an ordered sequence of steps. For each step, the AST preserves the type of step and the elements specified within it, such as the referenced item or the assignments provided. The effect of the step is not computed during parsing, but during verification.

The separation between representation and effect avoids incorporating into the structure built by the parser information that depends on verification. In this way, the AST preserves what the author has written in the script, while the verifier determines whether the steps are valid and what effects they produce.

4.4.7 Source Positions

The representation must preserve the information necessary to locate diagnostics in the source document. The position is not associated with every element of the representation but with two types:

- *constructs*, because an error can be traced back to the declaration to which it belongs;
- *proof steps*, because a single declaration can contain many steps and the position of the declaration is not sufficient to identify the step to which the diagnostic refers;

Lingua declarations do not require their own position, since they can be identified by their name, whereas a proof step does not have a similar identifier.

4.4.8 Single-Interpretation Representation

In the current implementation, it was decided to represent a single interpretation of the source document in the AST. This choice derives from the use of the *Unica* interpretation mode, in which a valid construct must admit only one interpretation.

If a construct admits multiple interpretations, the parser reports an ambiguity.

Supporting the *Omnes* or *Aliqua* modes would require a different representation, capable of preserving multiple interpretations and carrying their evaluation through the validation process. However, this is outside the scope of the present work.

4.5 The Verification Model

4.5.1 Three Channels

During verification, information is managed through three distinct channels: the input, that is, the representation of the document, the state of the context, and the diagnostics.

The *representation* of the document is used in read-only mode and is not modified during verification.

While the *state* represents the information available at the current point, such as the vocabulary and the items already declared. It is passed to the checks and updated every time constructs modify the context.

The *diagnostics* channel is like a second output produced by the checks and is propagated outwards without modifying the state.

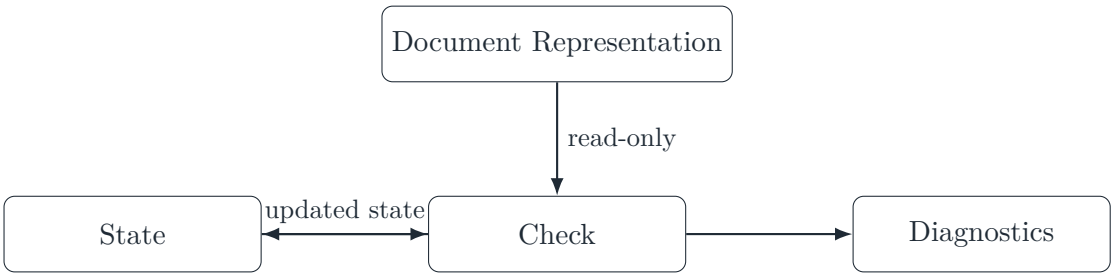

Figure 4.4: Three information channels used during verification. The document representation is read-only, the state is updated during traversal, and diagnostics are produced and accumulated.

4.5.2 Order of Traversal

The order in which the document is traversed depends on the information that must be maintained during verification.

The state is, in fact, updated from left to right, in the order in which the declarations appear in the document. This is because, in Hypatia, a declaration can be used by what follows it, but not by what precedes it. Respecting the order of the document is therefore necessary to maintain the correct visibility of the information during verification.

The diagnostics, on the other hand, follow the structure of the tree. The findings of the innermost nodes are first collected and then, in turn, propagated towards the constructs that contain them.

Chapter 5

Implementation

In this chapter, we focus on the implementation of Aristarchus, restricting our attention, as before, to Sunya under the *Unica* mode. We begin by introducing the programming language, before moving on to the parser and the validator.

5.1 The Haskell Programming Language

Aristarchus is developed in Haskell, a purely functional and statically typed programming language. Its functional paradigm relies on immutability and pure functions, which mirror mathematical definitions and keep execution deterministic by avoiding unintended side effects.

Algebraic Data Types and pattern matching offer a direct way to model an Abstract Syntax Tree in Haskell, since the shape of the data can mirror the grammar it represents. The language's own ecosystem illustrates much the same architectural principles that matter for theorem proving: GHC, Haskell's primary implementation, takes an abstract source language and translates it into Core, a strictly typed intermediate language based on a small lambda calculus with only a few core constructs.

5.2 System Architecture and Module Structure

The implementation of the Aristarchus project has been structured according to a modular, layered design. Figure 5.1 illustrates the component diagram of the system, highlighting the unidirectional flow of data and the strict separation of responsibilities. The architecture is divided into four main macro-packages. Below, a brief description will be provided for each of them, with a more detailed discussion for some of them following in the subsequent sections.

5.2.1 Imperative Shell: `App`

Represents the software's interface with the operating system and is the only module that performs I/O.

`CLI Dispatcher`: The entry point of the application that parses the command-line arguments.

`App.Toolkits.Validator`: Manages the lifecycle of the application. It reads the source files into raw byte streams, invokes the parser, passes the resulting tree to the validator and, finally, prints the diagnostic reports to the terminal, safely catching any I/O exceptions (by means of the `try :: IO (Either IOException ByteString)` construct).

5.2.2 Shared Data Structures: `Hypatia.AST`

Defines the domain of the problem. It is positioned at the base of the dependencies of the functional core so that the Parser and the Validator can communicate without being directly coupled.

`StructureFileHYP` and `Position`: Contain the Abstract Syntax Tree. The `Position` submodule introduces the `Located` functor. `Located` pairs an AST value with its coordinates in the source file (`Span`) without changing the type of the value.

`AST.Core`: Provides shared enumerators and `Trie` structures, chosen specifically for the *pattern-matching* on prefixes that a dynamically declared alphabet requires.

5.2.3 Construction: `Hypatia.Parser`

Translates the source bytes into the AST.

`Tokeniser` and `Segmentation`: The former extracts the delimited portions of text from the bytes, resolving escapes but without interpreting them; the latter, given a vocabulary and a text, enumerates every admissible reading without ever choosing one, if there is an ambiguity it is exposed and not resolved here.

`Fragment` and `Proofs`: The upper level of the parser, in which the text is finally transformed into logical constructs.

- `Fragment` extracts the fundamental structures of the language. It is in this module that the logic for handling Unica resides.
- `Proofs` is the module dedicated to reading the proofs, recognising its six steps (such as `$Assumo` or `$Obtineo`). It reads the formulae within each step using the functions provided by `Fragment` sub-module.

5.2.4 Logical Kernel: `Hypatia.Toolkits.Validator`

Represents the logical kernel of the system. Once the AST has been constructed, this package validates its constructs using the declarations and proof rules defined by the formal system. It is organised hierarchically:

`Types & Utils`: Defines the state of the system, encapsulated in the `Env` record, and the diagnostic structures. Unlike the parser, the environment does not travel within a monad transformer, every function receives it and returns it explicitly. It also provides the basic operations for navigating the lexical *scope* and resolving references to theorems.

`Algebra` and `Wellformed`: The mathematical level. `Wellformed` checks that every expression uses only symbols declared in the vocabulary (`validateAssertion`). `Algebra` models the rules of the formal systems: it extracts free variables, computes the safety of schemas and performs the formal substitutions on the expressions.

`Proof`: It checks the inference rules for the six possible `Move`s of a proof. `verifyOneProof` folds `verifyStep` over the moves of a script with a `ProofState` accumulator.

`Validator` and `Constructs`: These are the orchestrators of the kernel. They perform a *fold* over the AST, delegating the mathematical work to the underlying modules and accumulating a forest of outcomes (`LinguaIssues`).

`Report`: An isolated module dedicated to rendering the diagnostic text in *human-readable* form. Its separation from the `Validator` keeps the logical decision on the validity of the document (the verdict) separate from the visual formatting of the error messages.

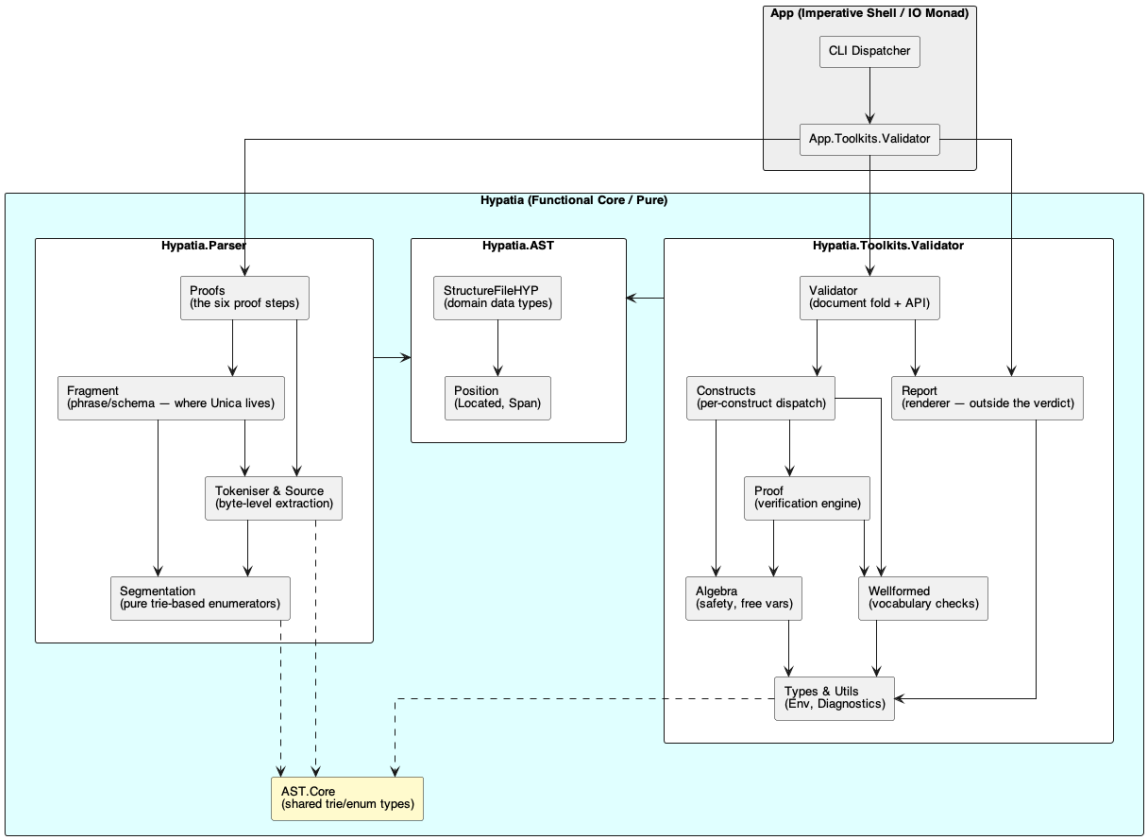

Figure 5.1: Component diagram of the Aristarchus toolkit. The figure illustrates the hierarchy of packages, the unidirectional dependencies, and the clear separation between the imperative shell (App) and the functional core (Hypatia).

5.3 The Command-Line Interface and the I/O Boundary

Aristarchus interfaces with the user through the command line, and therefore requires interaction with the external file system. In a purely functional language such as Haskell, functions with side effects (such as reading files or printing to standard output) are separated from the pure logic by means of the `IO` monad. To handle this constraint, the Command-Line Interface (CLI) acts as a shell, handling all impure interactions, while the validation logic remains pure (see Figure 5.2).

5.3.1 File Interfacing and Exception Handling

The entry point for the validation toolkit delegates the file-loading process to the `validateFile` function. Rather than using standard character-based reading, the CLI reads the source files as `ByteString` streams. This is a deliberate choice, working with byte streams turns out to be much faster than working with strings. To prevent the verifier from crashing due to missing files or permission errors, the read operation has been wrapped in a `try` block, which catches any `IOException` and translates it into an exit code, without interrupting the flow of execution.

5.3.2 Orchestrating Pure State within the IO Monad

Once a file has been loaded, the main responsibility of the imperative shell ends. The raw byte stream is immediately passed to `parseDocumentIn` and, subsequently, to

`validateDocumentIn`. These functions are pure and isolated from the file system. When a user specifies several files in a single execution, Aristarchus treats them as a single continuous logical document rather than as isolated runs. To support this behaviour, the `validateAll` function performs a sequential *fold* within the `IO` monad. It carries the `Vocabulary` and the logical environment (`Env`) from one file to the next, accumulating declarations and the results of the checks. In the event that a file cannot be read or parsed, the environment is passed on to the next file, so that a single faulty file does not corrupt the semantic context of the subsequent files.

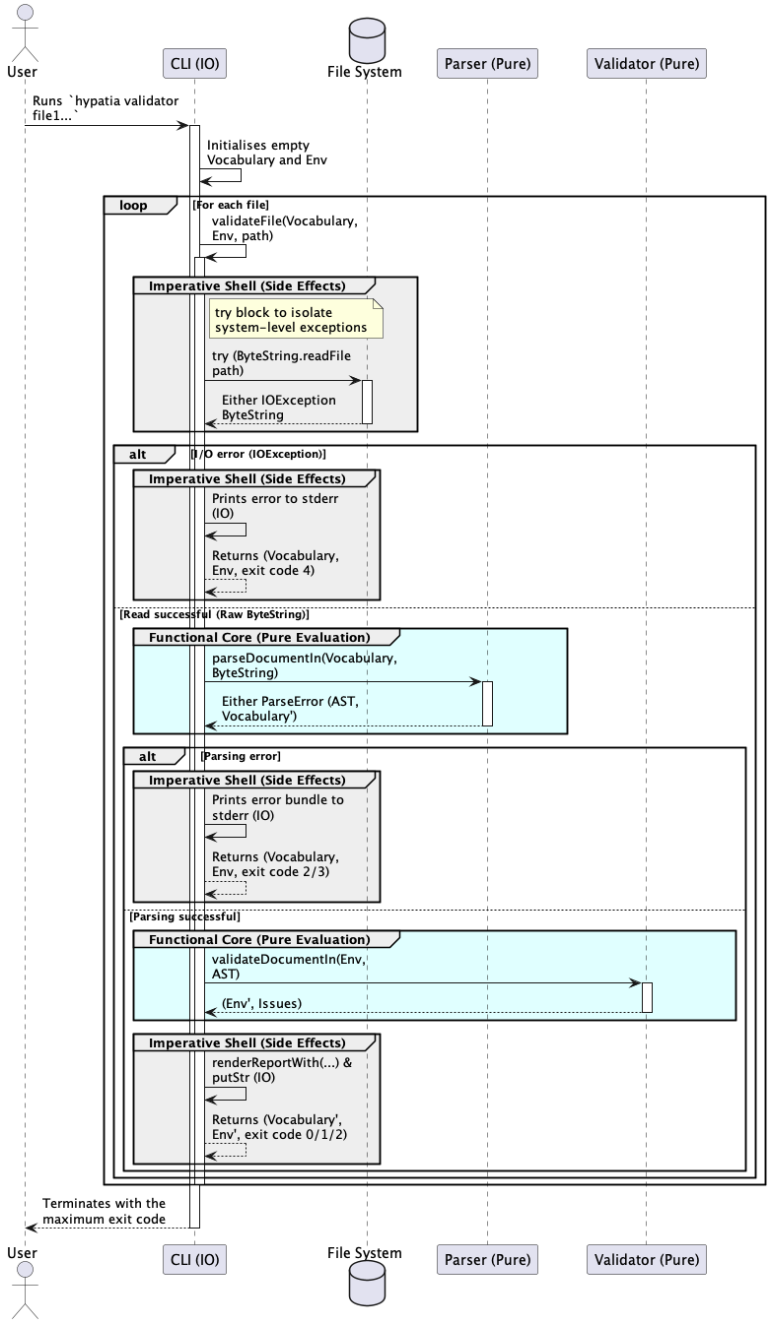

Figure 5.2: Sequence diagram illustrating the *Functional Core, Imperative Shell* architecture in Aristarchus. The imperative shell (grey) safely handles file system I/O and side effects, while the functional core (blue) remains strictly pure.

5.3.3 Exit Codes

The validation results are communicated to the host operating system through a hierarchy of exit codes. Aristarchus returns different exit codes based on the severity of the error. The codes are structured to reflect the level at which the document came to a halt:

- **0 (Success):** The document was read, parsed, and validated without encountering any error.
- **1 (Semantic Error):** The document is correct from the point of view of syntax, but fails one or more checks in the domain of semantics.
- **2 (Syntactic Error):** The document violates syntactic constraints or is malformed to the point of preventing the parser from generating the syntax tree.
- **3 (Out of Coverage):** The document invokes a feature that is not yet supported. In this case the file is not validated.
- **4 (System Error):** Unable to access the file due to I/O problems.

This enumeration is ordered by increasing severity. A single execution can include the sequential validation of multiple files, which Aristarchus interprets as portions of one large document. To determine the overall outcome, the aggregating function collects the exit codes of each individual file and returns the maximum value among these. This architectural choice makes the final verdict of the execution reflect the most error encountered, preventing the positive outcome of one file from masking the failure of another. The status code has nothing to do with the diagnostic report, which instead accumulates all errors/warnings. But this aspect will be discussed in detail further on.

5.4 Parser Architecture and Fundamental Types

`Hypatia.Parser` transforms the input `ByteString` into an Abstract Syntax Tree while maintaining the vocabulary-dependent parser state. The parser has to adapt to the dynamic nature of the Hypatia language. Hence, before analysing the algorithmic levels of decoding (Tokeniser, Segmentation and Fragment), this section describes the data structures on which the parser is based: the logical state, the chosen library and the monadic parsing stack.

5.4.1 The Dynamic State: The `Vocabulary`

Hypatia is a language whose decomposition rules change during reading, encountering constructs such as `$Alphabetum`, `$Lexicon`, `$Intervalla` or `$Inclusores` alters the set of symbols, separators and lexemes that the parser will have to recognise. To model this temporal dependency, the parser uses a dedicated record to encapsulate the syntactic context:

```
data Vocabulary = Vocabulary
{ vocAlphabetum  :: ASTAlphTrie
, vocIntervalla  :: ASTIntrTrie
, vocInclusores  :: ASTInclTrie
, vocSymbolNames :: IntMap Symbol
, vocLexicon     :: Set Word
, vocMutabilia   :: Set Word
, vocFragments   :: Map [String] Vocabulary
}
```

Each field of this record fulfils a specific architectural responsibility. Besides tracking the alphabet and the lexemes, the record maintains the inverse map `vocSymbolNames`, necessary for tracing a textual alias back to the canonical spelling of the corresponding symbol, and the map `vocFragments`. This map (`Map [String] Vocabulary`) stores the vocabularies of closed fragments. The key consists of the sequence of Lingua names from the current fragment outwards up to the root, so as to distinguish homonymous Linguas present at different points in the file. The associated value represents the state of the vocabulary at the moment of closure, ready to be reloaded when the fragment is reopened.

The choice of prefix trees (`Trie`) instead of plain `Set`s answers a practical need. Since the parser reads a continuous stream of text without knowing in advance the lengths of the symbols' spellings, a `Set` would force it to tentatively extract substrings of arbitrary length in order to test their membership. On the contrary, given that the admissible spellings of a symbol can share their initial part, for instance an alias and its extended form, the `Trie` allows the parser to descend along the nodes consuming the bytes one by one, gathering in a single pass all the overlapping candidate readings along the way. This structure does not serve to prevent or resolve ambiguities but is the tool that allows the parser to enumerate all the competing interpretations, leaving the final choice to be delegated to the upper layers. The same need does not arise again for the immutable lexicon and for the mutable lexemes, kept in plain sets (`Set Word`). At this level the problem that the `Trie` solves is no longer there: a `Word` is not a sequence of bytes to be segmented, but a sequence of already canonical symbols, obtained after the alphabet has traced every spelling, aliases included, back to its own identifier. Two different spellings of the same lexeme therefore arrive as the same sequence of symbols, and checking whether an already known sequence belongs to the declared lexicon is a simple membership test on an entirely determined key, for which a `Set` suffices.

5.4.2 The Choice of Parsing Engine: Megaparsec

The Haskell ecosystem offers several libraries for syntactic analysis, including classic generators such as Happy/Alex and combinator-based libraries such as Parsec, Attoparsec and Megaparsec. For the development of Aristarchus, the choice fell on **Megaparsec**. Megaparsec was chosen for three reasons:

- **Quality of Diagnostics:** Unlike libraries oriented towards performance such as Attoparsec, Megaparsec natively generates an informative *error bundle*. It precisely tracks the cursor position through line and column and the unexpected tokens, which proves useful for Aristarchus, which needs to return localised error messages.
- **Native Support for Raw Streams:** Megaparsec natively supports the parsing of `ByteString` streams, fitting the project's need to avoid a costly global decoding into UTF-8.
- **Monadic Integration:** The library provides `mtl` instances, so it can run under `StateT`.

5.4.3 The Monad Stack and the Input Flow

Instead of explicitly passing the `Vocabulary` as an input and output parameter for every single function, the parser exploits monad transformers to hide the complexity of the state *threading*. The entire parsing engine is defined by the following *type alias*:

```
type Parser a = StateT Vocabulary (Parsec Void ByteString) a
```

This signature combines several abstractions from the Haskell ecosystem:

- **`StateT Vocabulary`:** This monad transformer provides access to the current vocabulary state. The lower layers, such as `Segmentation`, can read and update the vocabulary without passing it explicitly as a parameter.
- **`Parsec Void ByteString`:** Represents the core of the Megaparsec engine. The `Void` parameter indicates that the library's standard error handling is used, delegating custom reporting to the upper levels. The `ByteString` type, instead, defines the nature of the input stream, which, as anticipated, operates directly on the raw bytes so as to avoid the overhead of global UTF-8 decoding, carrying it out only when necessary.
- **`a`:** Is the polymorphic type parameter that represents the result produced by the specific parsing operation currently in progress.

5.4.4 Blob Handling

Between one construct and the next, between the name of a construct and the colon that introduces its body, around the proofs listed in a Productum or a Derivatio, and between one proof step and the next, a document may contain a *blob*: informal text that the author writes and that the parser has to skip.

This is handled by `skipBlob`, called at each of these points and always parameterised on the terminals active at that point, which is what tells it where to stop; the search for the terminal is delegated to `activeSpelling`, which looks for the longest spelling of an active keyword starting from there. The function alternates two phases: first it consumes whitespace and comments by means of `blobSpace`, then it consumes one byte at a time of informal content, returning to `blobSpace` after each one, until an active terminal appears at the current point, at which point it stops, without consuming it.

Within a comment Hypatia's syntax is disabled entirely, terminals included: `blobSpace` recognises a line comment opened by `-` and skips it to the end of the line, or else delegates to `blockComment` a multiline comment `/- ... -/`, arbitrarily nestable; the latter was written by hand instead of relying on Megaparsec's standard combinator, because that one counts the nesting but there is no way to make it recognise a delimiter made literal by a *backslash*.

The informal byte that `skipBlob` consumes one at a time is never a dollar sign that introduces an active terminal, nor a *backslash* that opens a comment; both cases are intercepted beforehand, and treated as ordinary content.

- A `$` followed by the spelling of an active terminal is consumed in full as a blob, so that an author can name that terminal in prose without mistakenly closing the construct they are in.
- A `\` followed by `-` or `/-` is consumed in the same way, so that an author can write those two sequences without actually opening a comment.

5.4.5 Byte Extraction and Escape Handling

The first implementation problem faced by the pipeline is isolating, from the continuous stream of bytes, the portions of text that belong to the fragment (names, literals, phrases).

The recognition of a delimiter and the resolution of an *escape* never happen at the same point when the text is enclosed between a pair of inclusores: some decisions are made while the parser is still consuming the incoming bytes, others are deferred to ordinary functions, which have no access to the parser or to its state and receive as input only text that has already been extracted.

Escapes Resolved by the Parser Two categories of *escape* are resolved by internal functions of the parser itself, while it consumes the bytes, because the problem is purely structural and does not require knowledge of the vocabulary declared by the fragment:

- **Active terminals:** the local helper `escapedTerminal`, within `phraseLiteral`, `rawLiteral` and `skipBlob`, recognises a `$` that precedes the spelling of an active terminal and escapes it. The `$` character is then discarded and the spelling of the terminal is treated as content of the fragment, preventing it from being interpreted as an active terminal and from being able to prematurely close the construct that contains it.
- **Multiline comments:** the function `blockComment` handles the escaping of the `/-` and `-/` delimiters when it finds itself inside a comment that is already open. Outside a comment, in the prose of a blob, the handling of escapes is instead entrusted, as already seen, to `escapedComment`, defined in `skipBlob`, which allows the `-` and `/-` delimiters to be escaped. The `-/` delimiter is not considered, instead, since outside a comment it cannot perform any closing function.

The Object Text The text enclosed between a pair of inclusores instead requires two distinct passes.

In the first pass, still inside the parser, the helper `objectLanguageText` of `phraseLiteral` relies on the function `objectTextExtent` for the sole purpose of determining how many bytes the text occupies, not what it means. `objectTextExtent` is in itself a pure function, with no state and no access to the parser, but here it is invoked only to measure. In determining the extent, the function treats a doubled *backslash* as a single sequence, without splitting it, and recognises the cases in which a possible closing appears within an escape or an alias, so as not to consider it as the actual closing of the text. Once the extent has been determined, the parser takes those bytes as they are written, without interpreting them.

In the second pass, the same portion of bytes is re-read, this time by ordinary functions, with no notion whatsoever of parser or state. `objectTexts` recognises its opening and entrusts the reading of the content to `objectParts`. It is here that the sequences beginning with a *backslash* are analysed according to several possible interpretations:

- `aliased`: to decode an alias;
- `escapedClosing`: to interpret an escaped closing as literal content;
- `escapedBackslash`: to interpret an escaped backslash as literal content;
- `suppressed`: to suppress the reading of a mutable as a placeholder.

`objectParts` does not select one of these interpretations but produces all the readings that turn out to be possible, which therefore remain competing alternatives until `readingOf` counts them. Under Unica, if there are several readings, an ambiguity error is given.

5.4.6 Reopening fragments: *merge*, not restoration

A `$Lingua` block introduces its own lexical scope. In Hypatia, however, closing it does not erase the fragment's history. If another `$Lingua` with the same name and the same containment chain appears further on in the document, the parser does not create a new fragment but reopens the one already read. The handling of the state must therefore respect two constraints at once: applying the normal *rollback* towards the outer scope, and retaining enough information to resume the fragment in a later reading.

Tracking closed fragments The `vocFragments` field of `Vocabulary` preserves the vocabulary reached by each fragment at its closing. The structure is a *map* indexed not by the block's name alone, but by its full identifying chain, which would be the name of the current `$Lingua` followed, outwards, by the names of all its containers. `linguaConstruct` builds this chain at every opening. This choice makes it possible to separate homonymous blocks placed under different parents: two fragments called `E` are the same fragment only if they belong to the same point in the hierarchy. `vocFragments` is also the only field that does not undergo rollback on closing, unlike all the other elements of the vocabulary, which return to the state of the outer scope, while the map keeps accumulating the fragments already processed.

Opening the fragment and *three-way merge* On the opening of a `$Lingua`, `withFragment` saves the outer vocabulary and looks up the current chain in `vocFragments`. If it finds no entry, the fragment is new and starts directly from the vocabulary of the outer scope. If it finds a previous vocabulary instead, it invokes `restoredOnto`, which merge the three lexical states. The three components merged together are:

1. The **base state**: the vocabulary of the outer scope at the moment of the first opening, regularly recorded in the fragment's saved state;
2. The **local state**: the internal declarations processed by the fragment itself during its previous executions;
3. The **current outer state**: the vocabulary of the outer scope at the moment of reopening, updated with the new declarations.

In this way, a reopened fragment starts from its previous state, extended with the declarations that have become visible in the enclosing scope.

Collisions and *replay* of the alphabet `restoredOnto` does not merge all fields in the same way. `vocLexicon` and `vocMutabilia` are aggregated simply as sets; for `vocInclusores`, if the same opening is present in both vocabularies, all the associated closings are kept. The intervals in `vocIntervalla` are combined by means of a (*left-biased union*). Being an associative structure (*map*), a symmetric union would not be defined in the case of coinciding keys; left precedence explicitly implements the *shadowing* mechanism, ensuring that historicised local definitions shadow homonymous definitions coming from the outer scope, without generating ambiguity in the parser. The alphabet needs different handling. The *trie* `vocAlphabetum` associates each spelling with a progressive integer identifier, while `vocSymbolNames` uses that identifier to retrieve the corresponding symbol. The closed fragment and the outer scope may, however, have assigned the identifiers at different times and independently of each other. For example, the fragment may have assigned identifier

1 to the symbol `b`; after its closing, the outer scope may assign the same identifier 1 to the symbol `c`. A direct merging of the two alphabets would therefore leave both `b` and `c` associated with 1 in the *trie*, while `vocSymbolNames` might retain only one of them. During reading, the parser could then recognise `b`, obtain identifier 1, and wrongly reconstruct `c`. To avoid this collision, the fragment's alphabet is not merged directly. `addAlphabet` extracts the symbols remembered by the fragment and inserts them again on top of the current outer alphabet, as in a new declaration. The identifiers are thus assigned after those already present in the outer scope, and the associations actually reachable from `vocAlphabetum` remain consistent with `vocSymbolNames`.

Closing and consolidation At the end of the reading, `withFragment` records the updated vocabulary again under the same chain in `vocFragments`. The saved state also includes the entries that any sub-fragments have added to the map during processing. The parser then performs the rollback towards the outer vocabulary saved on entry, keeping `vocFragments` as the sole exception so as to allow the same fragment to be correctly reopened later on.

5.4.7 `ActiveTerminals` as a context parameter

`ActiveTerminals` represents the set of terminals that, at a precise point, must be recognised as elements of the host language. It is implemented as a list of `Terminal` and is passed explicitly to the functions that read unstructured content. Its content depends on the current grammatical production and on the constructs still open: it therefore includes the terminals that can begin a construct at that point and the terminators that must close the outer scopes.

```
type ActiveTerminals = [Terminal]
```

`Terminal` is the enumerated type of the host language's terminals, and thus contains the language's keywords. The type distinguishes a terminal's grammatical identity from its concrete spelling, which can have an extended form and, where provided for, an abbreviated form. An abbreviated spelling can moreover be shared by different terminals; for example, `$Rg` identifies both `TrmRegula` and `TrmRegulam`. It is the `ActiveTerminals` list that establishes which of these meanings is relevant at the point where the spelling is read.

The active terminals were intentionally excluded from the Vocabulary, and are instead propagated to the functions as a simple context parameter. This implementation choice is justified by the different nature and the different life cycle that characterise the two types of data. The Vocabulary is conceived to track the author's declarations: it is a dynamic state, subject to scoping rules and to complex accumulation and rollback operations. The active terminals, by contrast, constitute a structural constraint, determined by the current position in which the parser finds itself within the grammar.

If the active terminals had been embedded in the vocabulary, every entry into or exit from a nested construct would have forced the parser to perform continuous global state update operations. By passing them instead as a parameter, the validity of the terminals is delegated to the function call stack, the restrictions are established and removed during the evaluation of the individual syntactic node.

The tokenizer uses this parameter to establish the *lexical boundaries*. `activeSpelling` consults the global *trie* of the terminals' spellings, eliminates the matches that do not belong to the active list, and selects the longest active spelling. As a result, `skipBlob` stops the

reading of informal text before an active terminal, while `rawLiteral` and `phraseLiteral` terminate the textual fragment they are reading. A spelling that, in the current context, does not correspond to any active terminal does not constitute a *lexical boundary* and is treated as ordinary text. The list of active terminals also determines the behaviour of *escaping*: the `$` character protects a spelling only if it opens an active terminal; in other cases the `$` itself remains part of the text. `ActiveTerminals` does not itself recognise or consume the terminals, this task remains with the syntactic productions, by means of the `terminal` function. The parameter instead solves the lexical problem of establishing where free text must stop.

The propagation of the parameter that identifies the active terminals follows directly the nested structure of the document. At the outermost level, `$.` is not active because there is no construct to close; inside a `$Hypatia`, on the other hand, it becomes an active terminator. In the reading of a schema, `$-` is added to the active terminals during the premise, so as to separate it from the conclusion, but not during the reading of the latter. A nested schema adds `$ }` to the inherited context, while at the same time keeping the terminators of the outer scopes. The same mechanism is used in proof scripts, in the values of assignments and in the bodies of constructs: each caller supplies the terminals relevant to its own level, while each nested construct extends the context only with its own delimiters. The parser can thus handle arbitrary nesting without maintaining a depth counter or a separate stack for the terminals.

5.4.8 Parser errors

Purely syntactic errors, such as a missing terminal, an unexpected end of file or a spelling not admitted in a given production, are produced directly by Megaparsec. During the recognition of terminals and names, the functions associate the syntactic expectations with descriptive labels, so that the report indicates which element the parser was looking for. The cases that Megaparsec cannot describe on its own are instead represented by the `ParserError` type and raised as a *custom failure*. In this way the final report combines the library's syntactic information with errors that know the meaning of Hypatia's words, phrases and constructs.

Table 5.1: Parser errors and their description.

Error	Generation condition
`AmbiguousWord`	A `WORD` admits at least two segmentations in the declared alphabet. The error preserves the source text and up to two exemplary readings.
`UndeclaredWord`	A `WORD` cannot be segmented entirely into symbols declared in the alphabet.
`AmbiguousPhrase`	A phrase admits several valid readings through lexicon, mutable lexemes and object text.
`UnreadablePhrase`	A phrase admits no valid reading.
`AmbiguousValue`	An assignment value admits several readings, for example due to an ambiguity in the symbolic segmentation or in the object text.
`UnreadableValue`	An assignment value cannot be read as a valid sequence of `WORD` and object texts.
`BareNestedSchema`	A schema nested between braces appears in a position that requires a phrase or a complete schema, but is not followed by the `$-` separator.
`NotImplemented`	The document uses an element admitted by the language but not supported by the implementation, such as dialects and modes not yet supported.
`UnknownItem`	The document declares a name that is not recognised, for example an unforeseen dialect or interpretation mode.

The first family of errors concerns the unique-reading requirement imposed by the `Unica` mode. A `WORD`, that is a continuous portion of text read as a sequence of symbols, must admit exactly one segmentation in the declared alphabet:

- no reading produces `UndeclaredWord`;
- several readings produce `AmbiguousWord`.

The same uniqueness requirement applies to phrases and to assignment values, which however read the portions of meta-language differently:

- **Phrases.** Every portion of meta-language is read with respect to the entire declared vocabulary, made up of lexicon and mutable lexemes. The possible validity of the use of a mutable lexeme is checked afterwards by the validator and does not alter the reading performed by the parser.
- **Assignment values.** The `values` function does not consult the lexicon for the portions of meta-language. Each of them is read as a single `WORD`: it must therefore admit a segmentation into the symbols of the alphabet, but it is not broken down into several lexemes. This choice avoids spurious ambiguities, since the substitution does not preserve the lexical boundaries of the value: several adjacent `WORD`s and a single longer `WORD` can produce the same result.
- **Object text.** In both cases, a portion delimited by inclusores is read by means of `objectTexts`, which interprets the object text recognising aliases, escapes and mutable lexemes. In assignment values this reading is applied when an inclusore explicitly

opens a portion of object text. In phrases, instead, the same opening can generate both a reading as object text and a reading as lexeme; the two possibilities compete in determining any ambiguity.

For phrases and assignment values, the parser finally distinguishes two failure cases:

- **no reading**: `UnreadablePhrase` or `UnreadableValue`;
- **several readings**: `AmbiguousPhrase` or `AmbiguousValue`.

In the case of ambiguity, the report preserves the readings found, so as to show the different segmentations that produce the conflict. The enumeration stops as soon as it finds two readings, which is enough to distinguish an unambiguous case from an ambiguous one and avoids fully enumerating a space of segmentations that can grow exponentially with the length of the text.

5.4.9 Positions: byte offsets and UTF-8 rendering

The parser does not store source positions as row-column pairs. The canonical position is instead a *byte offset*, that is the number of UTF-8 bytes preceding an element in the file. This choice is consistent with the *scannerless* nature of the parser: the input is read as `ByteString`, the symbols and the inclusores are sequences of bytes, and the parsing primitives advance directly within this representation. A byte offset therefore makes it possible to locate and extract a portion of the source without intermediate conversions, guaranteeing better performance; row and column, instead, constitute a projection intended for the report, calculated only when it has to be shown to a user.

```
data Span
= Span
    { spanStart :: Int
    , spanEnd   :: Int
    }

data Located a
= Located
    { locSpan  :: Span
    , locValue :: a
    }

newtype Source
= Source ByteString
```

`Span` describes a half-open interval [`spanStart`, `spanEnd`): the first offset identifies the first byte of the construct, while the second identifies the first byte following its reading. This convention makes it possible to retrieve exactly the corresponding source by means of the initial offset and the difference between the two extremes. `Located` associates the *span* with the AST value without altering its semantic structure. The `located` function detects the offset before and after the parsing of an element and constructs the relevant interval. The dialect's constructs and the individual proof steps are located; a position is not, however, added to every node of the AST. This is sufficient for the verifier's reports, where an error in a construct is associated with the construct itself, while an error in a

proof can indicate either the step's number or its interval. Since the `located` combinator is applied to the step's parser, the *span* starts from the terminal that opens the step and ends after everything the step consumes. It thus includes, for example, any spaces immediately following `$.`, since the closing terminal consumes them; it does not, however, include blobs.

The custom stream `Source` Storing the offsets in bytes does not imply showing columns in bytes. A UTF-8 encoding in fact assigns several bytes to many Unicode characters: the symbol `¢`, for example, occupies two bytes in memory but must occupy a single column in the error message. Using `ByteString` directly as Megaparsec's *stream* to generate the reports would have produced three problems.

Firstly, the offending line would have been printed as a sequence of bytes. Secondly, the error cursor would have advanced by one column for each byte, ending up visually skewed to the right after every multibyte character. Finally, a further problem emerged during the *testing* phases: the direct printing of raw bytes to the terminal exposes the system to the risk of collision with system control codes. Some continuation bytes of UTF-8 sequences in fact correspond to reserved commands; by sending the terminal the undecoded bytes of some symbols, the system interpreted the byte `0x9D` (OSC, *Operating System Command*) not as part of a character, but as a system instruction, prematurely interrupting the printing and arbitrarily altering the title of the terminal emulator's window.

The `Source` newtype fixes these problems without changing parsing. Its implementation delegates all the consumption operations to the `ByteString` *stream*: the token remains a `Word8`, the chunks remain `ByteString` and the offsets therefore continue to be byte offsets. The only customisations concern the display of errors and the conversion of an offset into the corresponding row and column. To show an error, `Source` decodes the UTF-8 bytes and presents them as characters; to draw the underline, it measures the width of the sequence in characters rather than in bytes.

When Megaparsec has to show an error, `Source`'s implementation traverses the bytes of the offending line up to the requested offset, calculating the visual column incrementally. To do this efficiently, it analyses the individual bytes at bit level. In the UTF-8 encoding, the continuation bytes (the bytes following the first one within a multibyte character) are uniquely recognisable by the binary prefix `10`. Since they represent only a structural fraction of a character already begun, encountering them advances the byte counter but does not advance the on-screen column counter.

The effectiveness of this mechanism is explicitly validated in the parser's unit tests, which verify the decoupling between *offset* and column. Imagining a line composed of 16 ordinary ASCII characters (1 byte each) followed by two `¢` symbols (2 bytes each), the parser's internal cursor reaches byte 20 ($16+2+2$). However, by reading the binary prefixes, `Source` recognises that 2 of these 20 bytes are continuation bytes. The report will therefore not indicate the error at column 20, but at column 18 ($16+1+1$), positioning the graphical indicator in alignment with the characters rendered by the terminal.

From an architectural point of view, during this traversal the line is handled as a pure numeric sequence (contextually expanding tabs) and is decoded into actual characters only once, at the end of the operation. In this way the whole process remains optimised for *byte-oriented parsing*, with the conversion into visual coordinates only when a report has to be generated for the user.

From the span to the verifier's report The *spans* produced by the parser traverse the AST and remain available at the application level that generates the final report. The verifier's rendering module cannot, however, convert a *span* into a visual position on its own, this is because it does not have the original source and therefore receives a resolution function, of type `Span -> String`, as an external parameter. In the absence of such a function, the report is nonetheless generated, but without position indications. The actual conversion is carried out in the executable's module, at the point where both the source and the diagnoses to be shown are available.

Concretely, the `lineColumn` function uses only `spanStart`, because the report must indicate the point where the flagged construct begins and not its full extent; `spanEnd` is not currently used by the report; it is retained for future diagnostics that will underline a whole construct.

This `lineColumn` function therefore examines the portion of source preceding that offset: it counts the newline characters to obtain the row number and counts the characters following the last newline to obtain the column. As already explained, in the column count a UTF-8 continuation byte does not increment the value, since it belongs to the character begun by the preceding byte. The same rule is separately implemented in the `Source` *stream* as well; in both cases, the aim is to make the position shown in the report coincide with the column the user sees in the file. The conversion is carried out only when a construct produces a diagnosis, while for a valid document the offsets remain in the AST without being transformed into visual coordinates.

Interactions between the parser modules

Below are reported the sequence diagrams that describe the behaviour considered. For reasons of readability, some support functions have been omitted.

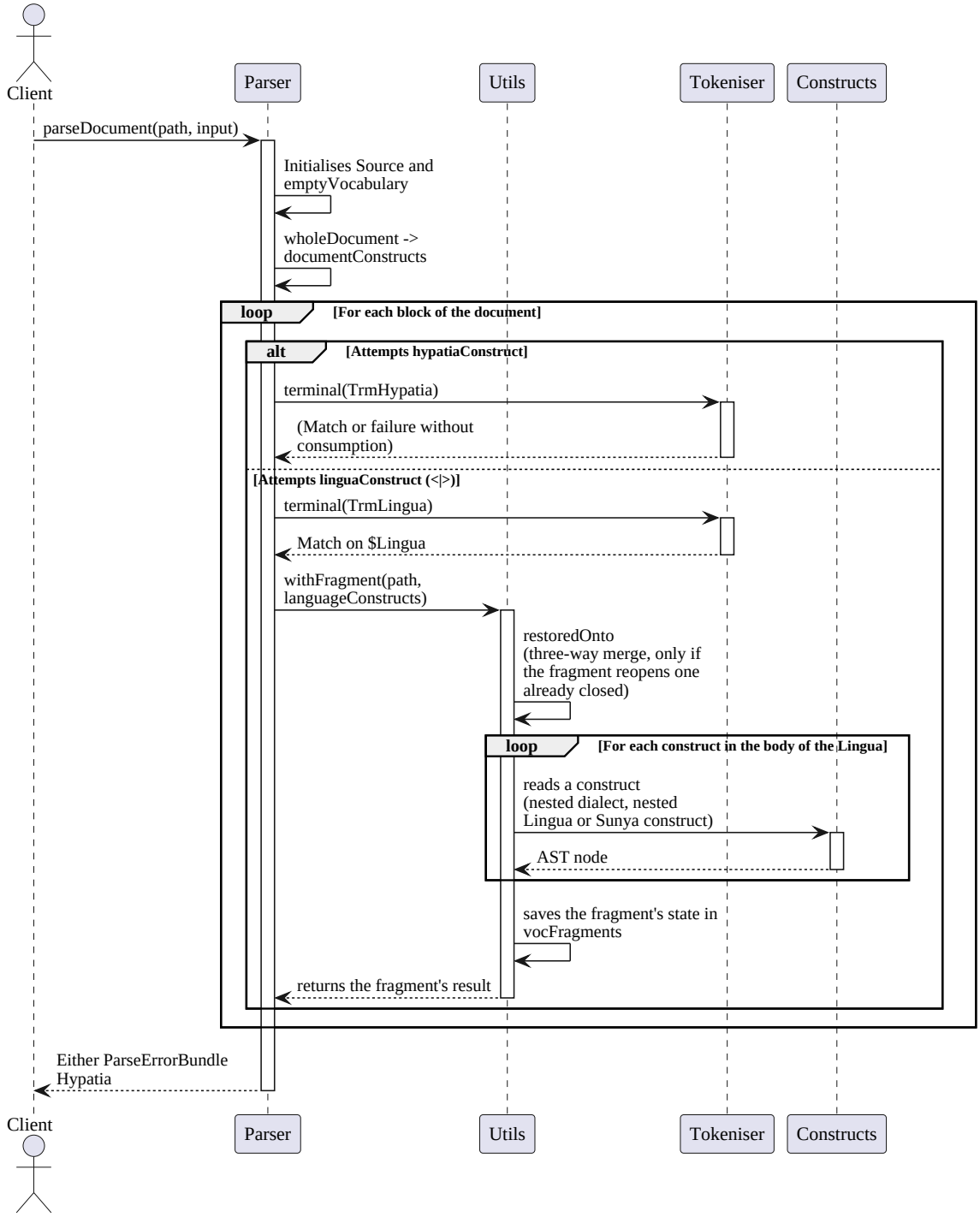

Figure 5.3: Parsing of a document and handling of `$Lingua` fragments, including reopening by means of `withFragment`.

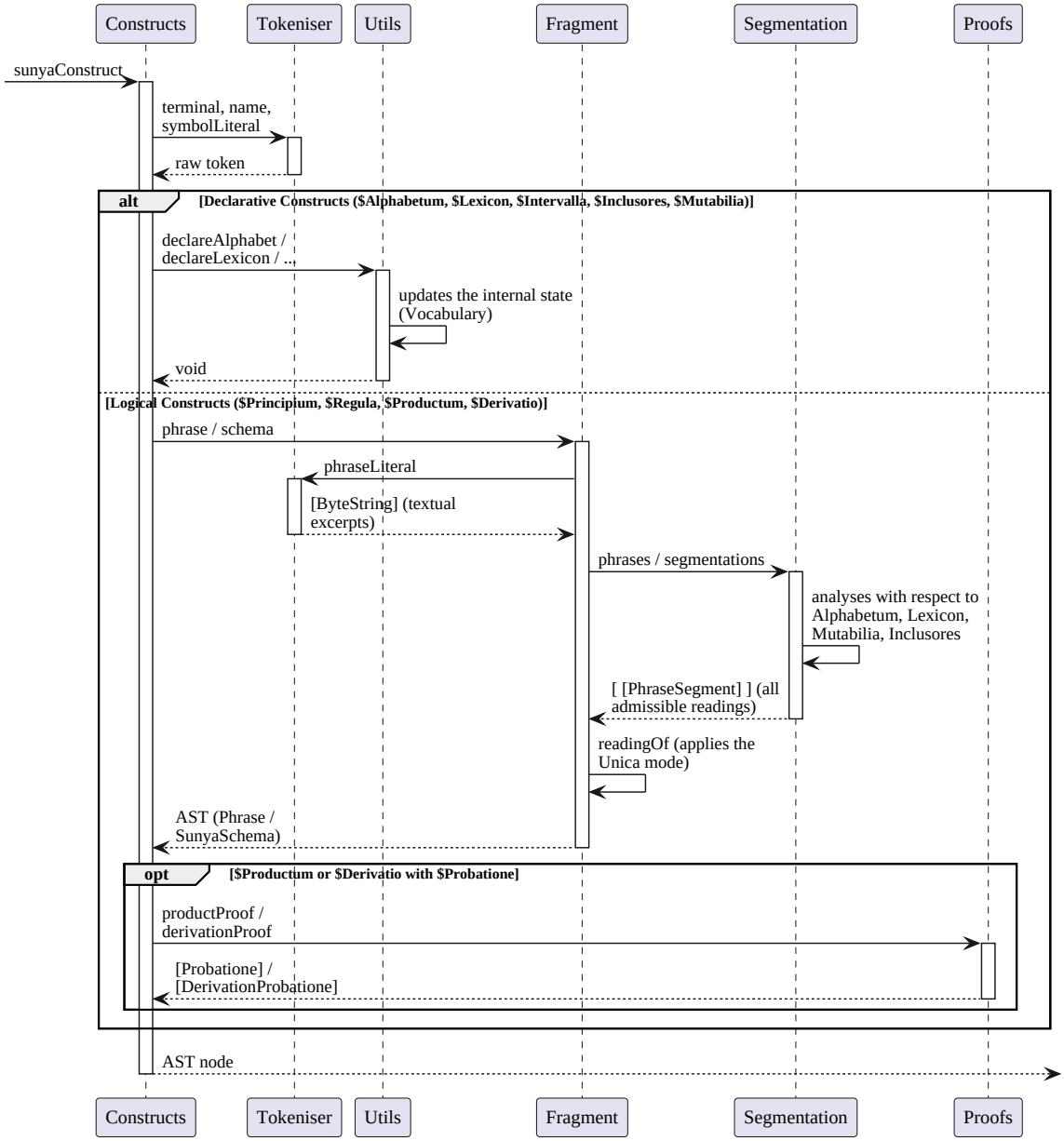

Figure 5.4: Parsing of the Sunya constructs: updating the vocabulary and reading phrases, schemas and proofs.

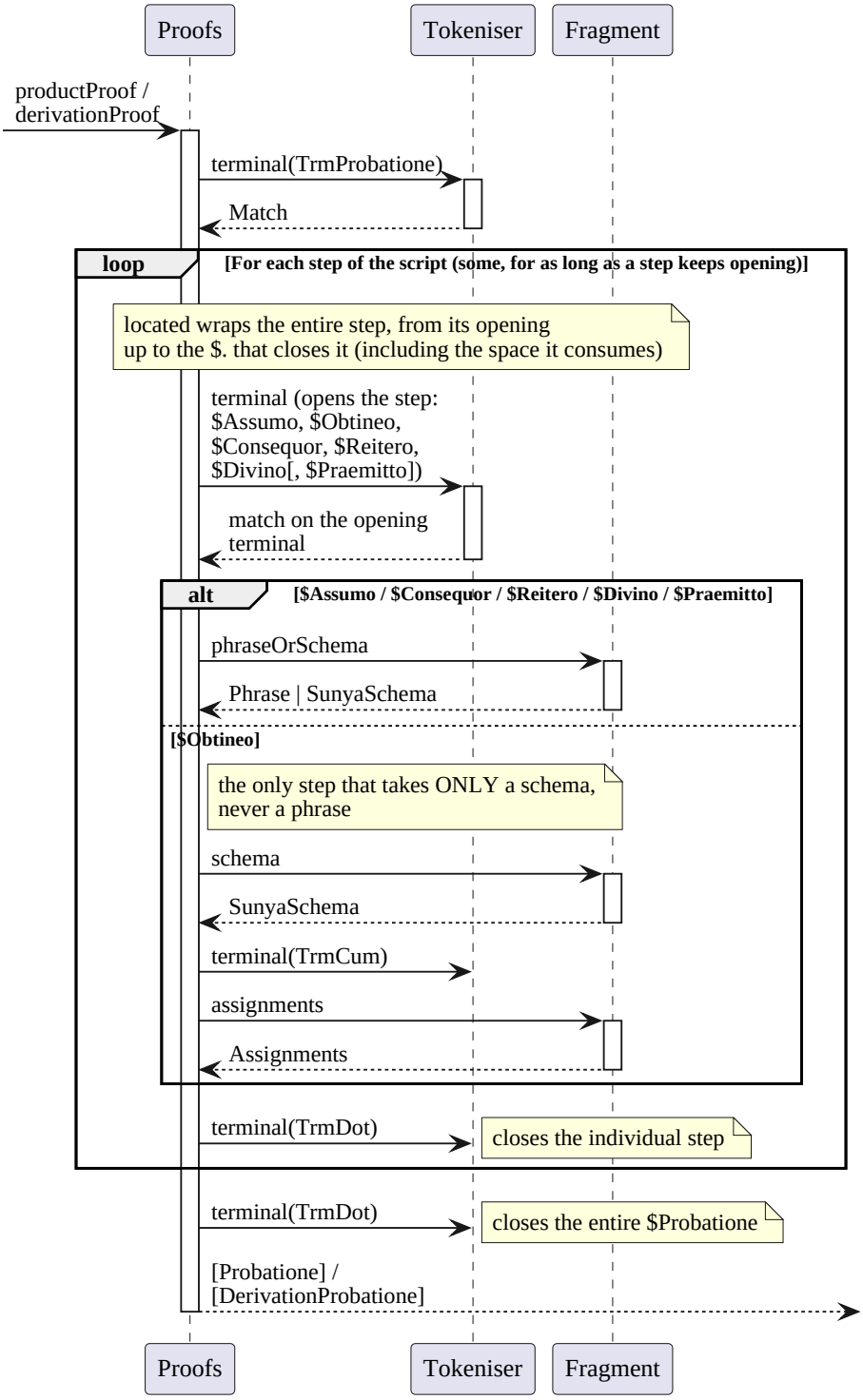

Figure 5.5: Parsing of a `$Probatione` script: reading, locating and closing the individual proof steps.

5.5 Verifier Architecture and Fundamental Types

The verifier does not reuse the parser's `Vocabulary` record. After the parser has transformed the source into the AST, the verifier builds and updates its own semantic environment, called `Env`. This environment gathers the declarations visible in the current fragment, the

resolvable names and the state needed to reopen an already closed fragment. No *state monad* is used: `Env` is passed explicitly to the validation functions, which return the updated environment together with the diagnoses produced. During the first approaches to the problem it was decided, for simplicity and clarity, not to use a state monad but to make the state explicit by passing it to the function; in future this mechanism will be refined with the use of a state monad.

```
type ScopeChain = [Map (ItemKind, Reference) NameEntry]

data Env
 = MkEnv
    { envAlphabetum :: Set Symbol
    , envAlphTrie   :: (ASTAlphTrie, IntMap Symbol)
    , envIntervalla :: Set Symbol
    , envInclusores :: Set (Symbol, Symbol, Symbol)
    , envLexicon    :: Set Word
    , envMutabilia  :: Set Word
    , envNames      :: ScopeChain
    , envPath       :: [String]
    , envFragments  :: Map [String] Env
    }
```

The fields of `Env` perform distinct functions:

- `envAlphabetum` contains the set of symbols declared in the alphabet. The choice of a `Set` supports both membership checks and the cumulative union of declarations.
- `envAlphTrie` is the representation of the alphabet needed for prefix reading. The *trie* associates a spelling with an identifier, while the `IntMap` reconstructs the corresponding symbol. The field is derived from `envAlphabetum`, but its evaluation is lazy: a proof that performs no substitutions does not build it, while a proof that performs at least one builds it only once. We will go into the reason behind this field in later sections, when we discuss the re-interpretation of instances after substitution with Obtineo.
- `envIntervalla` stores the symbols declared as intervals. The verifier uses it to check that an interval does not also take on the role of an inclusore.
- `envInclusores` keeps the triples (opening, closing, separator) with which the grammar admits the declaration of an inclusore. Only the first two symbols describe the inclusore proper: the separator is part of the declaration syntax, but reading under the `Unica` mode does not consult it, though it has nonetheless been implemented with future extensions in mind.
- `envLexicon` contains the immutable lexemes declared in the lexicon. It serves to check that the portions of meta-language comply with the declared vocabulary.
- `envMutabilia` contains the mutable lexemes. Besides checking phrases, it is needed to check the domain of substitutions and to compute the free parameters in proofs.
- `envNames` is a stack of *maps*, one for each `$Lingua` *scope*. Each entry associates the item's type and the reference by which it is reachable with a `NameEntry`, which preserves the declaration and any defects already known. The structure allows the

resolution of references, the checking of redefinitions and the propagation of defects to later citations.

- `envPath` contains the chain of fragments, from the innermost outwards. It stays aligned with `envNames`, with one element for each *scope*, and serves to complete the qualification of a qualified reference during resolution.
- `envFragments` stores the environment reached at the closing of each fragment, indexed by its full chain. It is the only field that does not undergo rollback on the closing of a `$Lingua`; it therefore allows a later block with the same chain to reopen the fragment.

`envAlphabetum` and `envAlphTrie` have different responsibilities. The former answers membership questions, while the latter allows a sequence of bytes to be reread, identifying the symbols that make it up. The verifier's *trie* contains only the canonical spellings of the symbols and not the aliases, which are instead recognised by the parser. The reason why this rereading is necessary after a substitution is discussed in the section devoted to the re-interpretation of instances.

The handling of fragments follows the same principle adopted by the parser, but is simpler. On reopening, the fields describing the vocabulary are combined by set union; `envAlphTrie` is not merged, but is redefined from scratch starting from the resulting alphabet. There is therefore no risk of collision between numeric identifiers, which in the parser required the *replay* of the alphabet. Names follow a distinct rule: each fragment opens a new *scope* in `envNames`, while the exportable items are explicitly imported into the containing fragment and acquire the qualification needed to preserve their origin. Declarations remain recorded even when they are erroneous; the related defects are preserved in the `NameEntry`, so that a later citation receives a precise diagnosis. This too will be discussed again in later paragraphs.

The initial environment is initialised by `emptyEnv`, and contains empty sets and maps, an empty stack of name *scopes* and no stored fragment.

5.5.1 Declarative traversal and updating of the environment

The validation of the document follows the order of appearance of the constructs in the source file. The entry point, `validateDocument`, initialises the *environment* with `emptyEnv`; within each fragment, `collectContents` performs a *traversal* and delegates the checking of each construct to `validateSingleConstruct`. The latter returns both the diagnostics produced and a new `Env`, which becomes the current *environment* for the next construct. Declarations therefore have visibility going forward (*forward-only*). The `$Lingua` fragments are traversed recursively; the handling of their *scopes* will be described in the next block.

The updating of the *environment* is coupled to the checking of the individual construct. Within `validateSingleConstruct`, the `ALPHABETUM` case extends the set of symbols and keeps the derived representation `envAlphTrie` consistent, used when a later re-interpretation requires segmentation by prefix. The `INTERVALLA` and `INCLUSORES` cases check, by means of `roleClashes`, that the same symbol does not receive incompatible delimiting roles. The `LEXICON` and `MUTABILIA` cases check instead that every `WORD` is made up only of symbols already present in the alphabet and that a lexeme does not belong to both categories at once. Principles and products validate a phrase, while rules and derivations validate a schema; in both cases the verifier also applies the *safety* conditions required by the form of the declaration. For `PRODUCTUM` and `DERIVATIO`, the checking of the respective proof scripts

is added to the validation of the statement.

5.5.2 Representation of declared items and error tolerance

One principle of the architecture is that a declarative error does not interrupt the *traversal* of the AST. The vocabulary declarations are nonetheless incorporated into the `Env` and the related diagnostics are collected separately. The `PRINCIPIUM`, `PRODUCTUM`, `REGULA` and `DERIVATIO` constructs are likewise registered in the associative map of names of the current *scope*, as values of type `NameEntry`, even when their declaration presents defects.

The types involved are the following:

```
data ItemKind
= ItemPrincipium
| ItemProductum
| ItemRegula
| ItemDerivatio

data NameKind
= KindPrincipium Phrase
| KindProductum Phrase
| KindRegula SunyaSchema
| KindDerivatio SunyaSchema

data NameDefect
= StatementBroken
| ProofFailed
| ProofConjectural

type NameDefects = Set NameDefect

data NameEntry = NameEntry
{ neKind    :: NameKind
, neDefects :: NameDefects
}

type ScopeChain = [Map (ItemKind, Reference) NameEntry]
```

This separation between the *lookup* key, the content of the item and the state of the defects answers three design needs:

- **Unique identification**: `NameKind` records the category of the construct together with its statement, a `Phrase` or a `SunyaSchema`. The distinction between homonymous items belonging to different logical categories, and the unambiguous resolution of a citation, are instead delegated to `ItemKind`. The latter represents the content-free projection of `NameKind` which, in conjunction with `Reference`, constitutes the lookup key within the map.
- **Independence of defects**: The use of a set (`Set NameDefect`), as an alternative to a mutually exclusive enumeration, is motivated by the fact that several defects can coexist. A single item can, in fact, present simultaneously a malformed statement (`StatementBroken`) and a failing proof script (`ProofFailed`).
- **Prevention of cascading errors**: Should a statement fail validation, the item

is nonetheless registered with the `StatementBroken` defect. A later citation of the item therefore still resolves correctly, avoiding raising the misleading unknown-name error (`ErrUnknownName`); the verifier will instead issue the `WarnUncheckedCitation` warning. For the `PRODUCTUM` and `DERIVATIO` constructs, any anomalies in the proof script generate the `ProofFailed` or `ProofConjectural` defects, in every case keeping the statement registered and referenceable.

Finally, during the updating of the associative map, the verifier handles collisions between keys characterised by the same name and the same `ItemKind`. If the statements differ, the system preserves the first declaration and reports `ErrRedeclarationMismatch`. In the case where the statements coincide, a new declaration of `PRODUCTUM` or `DERIVATIO` is interpreted as an alternative proof of the same item; the `mergeDefects` function then intervenes, combining the new outcome with the pre-existing one, so as to preserve the validity state. A `PRINCIPIUM` or `REGULA` has no proof script, so repeating it produces only `WarnDuplicateName`.

5.5.3 Qualified references and export between fragments

Qualification is not handled as a plain string, but is represented by the type:

```
type Reference = (String, [String])
```

The first element contains the name of the item, while the second contains the chain of fragments that qualify it, ordered from the innermost outwards. In the associative map of names, this `Reference` is paired with `ItemKind`: a principle, a product, a rule and a derivation can therefore share the same name without colliding. In the *scope* that declares an item, the chain is initially empty. When the fragment closes, `envImportInto` makes the exported items accessible in the parent *scope*, appending to their chain the name of the fragment just left. If `P` is declared in `B`, nested in `A`, the transformation takes place as follows:

`P`	$\longrightarrow$	`P $@ B`	$\longrightarrow$	`P $@ B $@ A`
`("P", [])`	$\longrightarrow$	`("P", ["B"])`	$\longrightarrow$	`("P", ["B", "A"])`
in the *scope* of `B`	$\longrightarrow$	after the closing of `B`	$\longrightarrow$	after the closing of `A`

The chain stored by a map is relative to the *scope* that contains the item, it records the boundaries crossed by the item during exports, without duplicating each time the entire path down to the root of the document. The import operation therefore requires only the name of the fragment being closed. Moreover, an imported item keeps a different key from that of a local homonym, so `P $@ B` and `P` can coexist in the same associative map. *Shadowing* is decided only during resolution, without eliminating either of the two declarations.

Before the import, the items declared in a nested fragment are filtered on the basis of the vocabulary used in their statement, so that only what uses symbols and lexemes already available in the parent fragment is exported. An item dependent on local vocabulary therefore remains confined to the fragment that introduced it. An invalid statement constitutes an exception and is exported regardless, together with its defects, so as to avoid cascading errors in later citations. Top-level fragments, on the other hand, export all their own names, making them accessible to the sibling fragments that follow in the document. In every case, only the names and their validity state cross the boundary.

Resolution distinguishes two forms of citation:

- **Unqualified citation.** The verifier looks up the empty-chain key in each scope of the `envNames` stack, from the innermost outwards, stopping at the first result. This logic is handled by `resolveMatches` in the branch dedicated to references without qualifiers. A closer declaration thus hides any more distant homonym, and imported declarations can never satisfy this search, because they are stored with a non-empty chain.
- **Qualified citation.** In the other case of `resolveMatches`, the one with non-empty qualifiers, the verifier completes the relative chain present in the key with the portion of path preserved in `envPath`, which identifies the chain of fragments from the one under examination outwards. The chain supplied in the citation is then compared as a prefix of the candidate's full chain. In this way it becomes possible to cite items exported from nested fragments, from preceding sibling fragments, or from a fragment that encloses the point of the citation.

The qualification does not have to be complete, but it must identify only one item. If no item of the requested type satisfies the search, the verifier reports `ErrUnknownName` when the name does not exist at all, or `ErrNameKindMismatch` when it exists but belongs to a different logical category. If, instead, several items correspond to the same partial qualification, `ErrAmbiguousReference` is produced, and the author must therefore specify a longer chain.

5.5.4 Verification of proof scripts and local state

During the verification of a proof script, the fragment's *Env* is used as a read-only declarative context. The *Env* is indeed updated during the validation of the fragment's constructs, but it is not modified by the execution of the individual scripts. The state produced during the proof is instead kept in a local value, initialised separately for each script by the `verifyOneProof` function, starting from `initialProofState`. In this way, premises, intermediate results and diagnostics produced during a proof do not affect either later scripts or any alternative readings of the same item.

Local state and representation of the steps

The proofs of `PRODUCTUM` and `DERIVATIO` have distinct ASTs: only a derivation can introduce premises by means of `$Praemitto`. Before verification, both are translated into the internal `Move` type by means of the `fromProduct` and `fromDerivation` functions. This type represents the common operations of the verification engine. Each `Move` operates on a `SunyaAssertion`, that is on a mutable phrase or on a schema, and preserves the source position already associated with the step by the parser.

```
type ProofContext = [ProofItem]

data ProofItem = ProofItem
{ piAssertion :: SunyaAssertion
, piRigid     :: Set Word
}

data ProofState = ProofState
{ psContext     :: ProofContext
```

```
, psStep        :: Int
, psPremises    :: [SunyaAssertion]
, psFreeParams  :: Set Word
, psPremiseMut  :: Set Word
, psInherited   :: NameDefects
, psDiagnostics :: Diagnostics
, psPseudo      :: Bool
}

data Move
= MoveAssumo Assumo
| MoveObtineo SunyaAssertion Assignments
| MoveConsequor SunyaAssertion
| MoveReitero SunyaAssertion
| MoveDivino SunyaAssertion
| MovePraemitto SunyaAssertion

data ProofOutcome = ProofOutcome
{ poDiagnostics :: Diagnostics
, poDefects     :: NameDefects
}
```

The fields of `ProofState` perform distinct roles:

- `psContext` is a stack of the items established up to the current step, updated by the `pushChecked` and `pushExposed` functions. Each `ProofItem` preserves the assertion produced and the set of rigid mutabilia in that specific occurrence. Rigidity is therefore contextual metadata, not an intrinsic property of the assertion, and it limits the substitutions admitted by later steps.
- `psPremises` is a separate stack, updated only by `$Praemitto` by means of the `pushPremise` function. Unlike the *proof context*, it does not indicate what is usable in the current step, but what will have to be discharged at the end to reconstruct the proven schema.
- `psFreeParams` and `psPremiseMut` respectively accumulate the free mutabilia and all the mutabilia present in the premises already introduced. The two sets remain separate because they answer different checks: the former plays a part in the computation of rigidity, the latter in the *contextual well-formedness* of the intermediate items.
- `psStep` keeps the index of the step, starting from one. It serves to link a diagnostic to the specific step and to its *span* in the source. The `advance` function increments it, while the `atStep` function uses it to mark the diagnostics.
- `psInherited` gathers the defects inherited from the items cited by means of `$Assumo`, through the `inheritDefects` function; `psPseudo` instead records the direct use of `$Divino` in the current script. The distinction makes it possible to differentiate a pseudo-proof produced locally from a proof that depends on an external conjecture.
- `psDiagnostics` is the accumulator of the diagnostics produced during execution, updated by the `addDiag` function.

Execution of the scripts and collection of the outcomes

Each script starts from an empty `ProofState` and is traversed in order. The engine executes one step at a time, updates the state and associates the new errors with the step number and its position in the source (`verifyStep`, then `atStep`, then `advance`, in this order); the counter advances only after the current step has been processed. An error does not interrupt the script: when a step nonetheless produces an item, it is inserted into the *proof context* together with its own diagnostics by means of the `pushChecked` and `pushExposed` functions, so that the following steps can be verified and independent errors can be reported in the same run.

The internal operations correspond to the language's instructions, all handled by `verifyStep`: `$Assumo` resolves a named item and inherits any of its defects by means of `citeName`; `$Praemitto` introduces a premise; `$Obtineo` applies a substitution to the most recent item; `$Consequor` applies a schema to the two most recent items; `$Reitero` reuses an already established item; `$Divino` introduces an item without justification and marks the proof as a pseudo-proof. Substitution and the subsequent re-interpretation of the instance are described in the next section.

At the end of the script, the verifier compares what has actually been established with the target of the declaration by means of the `finalMatch` function. For a `PRODUCTUM`, the last item of the *proof context* must coincide with the declared phrase. For a `DERIVATIO`, the premises collected in `psPremises` are discharged, in their order of presentation, around the last item established, by means of the `spineErrors` function; the schema thus reconstructed must coincide with the declared one. This final check is separate from the individual steps, since it concerns the script as a whole.

The same `PRODUCTUM` or `DERIVATIO` can contain alternative scripts. The verifier executes all of them and preserves all the diagnostics, even if a valid script has already been found, by means of `verifyProofs`. `ProofOutcome` separates these diagnostics from the defects that the item will pass on to later citations: an alternative proof free of errors and of inherited defects renders the item fully established; if no script terminates without errors, the item acquires `ProofFailed`. In this way the report remains complete, while the logical state of the item reflects the best available proof.

5.5.5 Re-interpretation of instances after substitution

The `$Obtineo` step builds an instance of the most recent item by applying the assignments declared after `$cum`. The `buildSubstitution` function transforms the assignments read by the parser into a `Substitution`, an associative map of type `Map Word MutablePhrase`: each `WORD` used as a key is associated with the value that will have to replace it. A single assignment can indicate several keys and only one value; for example, `x, y $= M` generates two associations, one for `x` and one for `y`, both pointing to `M`. The value is a `MutablePhrase`, which can contain both meta-language and object text.

The verifier checks that the keys of the substitution are declared mutabilia, active in the preceding item and not rigid, besides reporting any repeated or conflicting assignments.

The initial application of the substitution is structural: an unquoted mutable is replaced by the associated value, while a placeholder in the object text is replaced by the symbols represented by that value. This step, however, cannot constitute the final result of the instance.

The reason is a change of grammar. In the parser, a portion of meta-language not

enclosed between inclusores, within an assignment value, is read as a single `WORD`: the parser checks its symbols against the alphabet, but does not break it down into declared lexemes. After the substitution, the same portion, however, enters a `MUTABLE-PHRASE`, where every segment of meta-language must be interpreted with respect to the union of `LEXICON` and `MUTABILIA`. A value that was previously a single `WORD` can therefore correspond, in the new context, to several distinct lexemes. The object text, too, must be reread: the substitution flattens it into a sequence of symbols, while a resulting sequence may now contain a mutable that must be recognised again as a placeholder.

The `reinterpret` function performs this second reading, distinguishing the two forms of segment. For an `Unquoted` meta-language segment, `spellWord` reconstructs the bytes of the single `WORD` by concatenating the spellings of its symbols; `reinterpretSegment` can thus check a new segmentation into lexemes. For a `Quoted` segment, `spellOut` instead serialises the content of the object text: both the verbatim portions and the placeholders are traced back to the sequence of symbols they represent, while the pair of inclusores remains unchanged. The content thus obtained is reread to recognise any placeholders again.

In both cases, the reading uses `envAlphTrie`. The *trie* supplies the possible readings by prefix in the current alphabet, while the subsequent segmentation retains only the decompositions compatible with the declared vocabulary. For the meta-language, both lexicon and mutabilia are considered; for the object text, the reading that recognises the placeholders again is applied. If there is a single reading, the instance's AST is updated with the segmentation obtained. If, instead, an unquoted portion does not correspond to any lexeme, it is kept unchanged: the *well-formedness* check will then produce the appropriate lexical diagnosis. In the object text, by contrast, there is always at least the original reading, since the reread content is the serialisation of symbols belonging to the same alphabet.

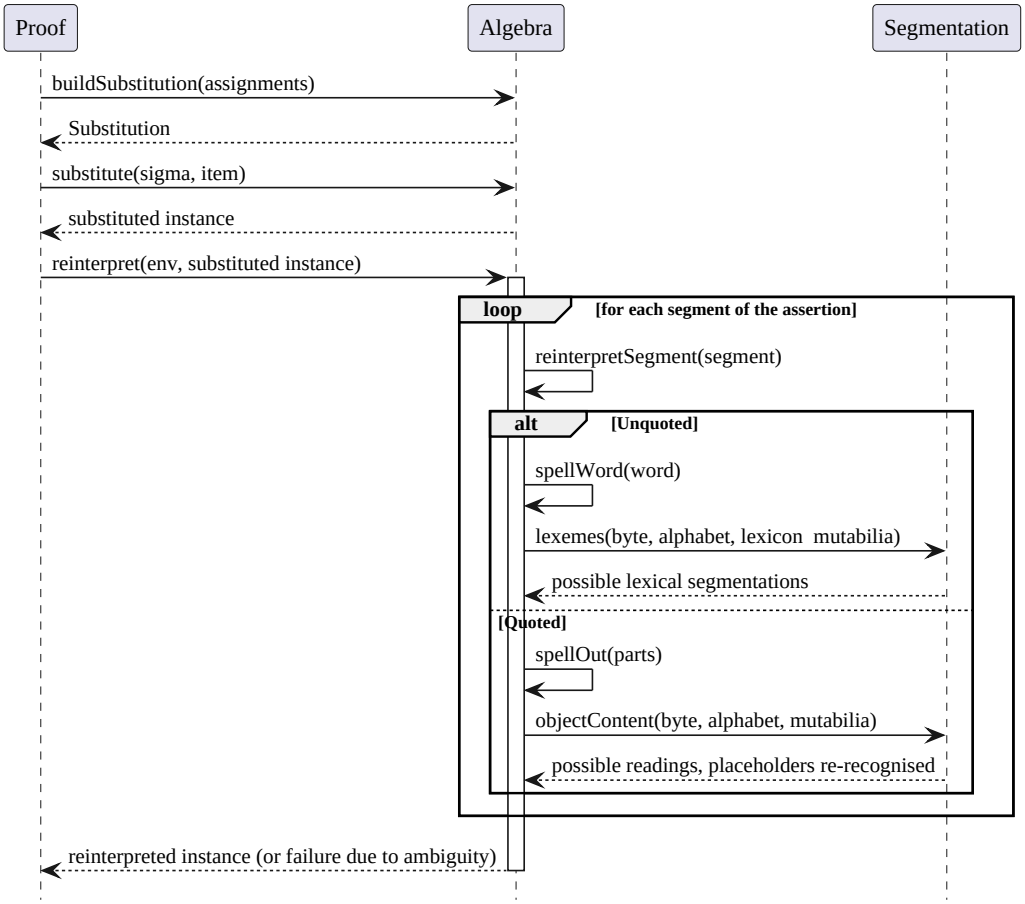

Figure 5.6: Substitution and re-interpretation mechanism at the $Obtineo step. The diagram is simplified.

5.5.6 Prevention of proof circularity

Circularity is checked during the resolution of `$Assumo`, that is at the only point where a script can use a declared item as justification. The verifier does not build a global dependency graph: it instead exploits the order of the traversal.

Order of declarations. A `PRODUCTUM` or a `DERIVATIO` is registered in the map of names only after its scripts have been verified. Consequently, during the verification of a script only already closed declarations are accessible; two distinct items therefore cannot form a mutual cycle through future declarations.

Self-citation. Self-citation still needs to be treated explicitly. Before resolving the reference written in an `$Assumo` step, the verifier compares the identity of the requested item with that of the item that the current script is justifying. The identity comprises `ItemKind` and `Reference`: a proof of `PRODUCTUM P` that cites `PRODUCTUM P` is therefore rejected with `ErrCircularProof`. The check is performed before the search in the map of names, thus making the diagnosis independent of the order of declarations. In the first declaration of `P`, self-citation is therefore recognised as `ErrCircularProof` even though `P` is not yet present in the map; in a redeclaration of `P`, the same check instead prevents circularity from being bypassed by referring to the previous declaration already registered.

Identity of the item. The comparison includes the type of the item, not just the name. It is therefore legitimate for a proof of `PRODUCTUM P` to cite a `PRINCIPIUM P`: the two constructs share the name but have different `ItemKind`s and represent distinct items.

Continuation of verification. When a circular citation is detected, the step receives the related diagnostic, but the assertion it declares is nonetheless inserted into the *proof context*. In this way the script continues to be verified and can report further independent errors; the presence of `ErrCircularProof` nonetheless prevents that script from constituting a valid proof of the item.

Chapter 6

Validation and Testing

The tests presented in this chapter provide empirical evidence on the behaviour of the verifier in the cases examined. For each document the expected verdict is known, and is compared with the one produced by the verifier, but these results do not constitute a proof of *soundness* nor of *completeness*. With this evaluation it is not possible to rule out any errors in documents not examined.

6.1 A case study: elementary ring theory

To exercise the parser and the verifier on a complete document, the case study formalises a small fragment of elementary ring theory and derives the standard result $a \cdot 0 = 0$ (zero absorbs multiplication).

The document is a single fragment (`$Lingua Rings`) that declares the vocabulary:

```
$Alphabetum @ § T E a b c d e g l m n o q r u z $.

$Lexicon @Trm @Eq @zero @one @add @mul @neg $.

$Mutabilia §a §b §c §d $.
```

Eighteen symbols in the alphabet, with the two characters `@` and `§` plus every letter used to write a lexeme or a metavariable; seven immutable lexemes: (`@Trm`, `@Eq`, the constants `@zero` and `@one`, the operations `@add`, `@mul`, `@neg`), and four mutable lexemes (`§a`, `§b`, `§c`, `§d`) used as metavariables for terms.

Every meta-language lexeme begins with `@` and every metavariable with `§`; for convenience we will call these two symbols "class characters". The two class characters thus separate the two categories of lexemes. This separation alone is not enough to guarantee that a phrase decomposes in only one way: if the vocabulary declared `@a`, `@b` and `@a@b`, the text `@a@b` would admit two complete decompositions: the single lexeme, or the two in sequence.

What makes the reading unique is a property of the declared vocabulary:

- The declared *symbols* are all one character long and pairwise distinct, none is a prefix of another, so the reading of a text as a sequence of symbols is already forced.
- At the level of *lexemes*, every declared word carries one class character in the initial position, and no other symbol of the alphabet is a class character. The occurrences of `@` and `§` in a phrase therefore coincide one to one with the beginnings of words. Between one class character and the next there lies only one segment, which is a declared lexeme, and no alternative complete decomposition is possible.

Resting on this vocabulary are two principles (the terms `@zero` and `@one`), eighteen rules, one product and five derivations.

The document uses all four of the language's named constructs, five of the six proof moves provided for, nested schemas with several premises, and one derivation, which presents two

distinct premises rather than a single one. No proof resorts to `$Divino`, so the entire chain, up to the final theorem, is established without conjectures.

The document validates successfully:

```
$ hypatia validator "rings.hypatia"
[OK] rings.hypatia successfully validated.
```

The following sections follow step by step two of its proofs, the simplest one and one that presents a premise, to show how the mechanisms described in the previous chapters operate on a real case, and then summarise how the remaining derivations chain together up to the concluding theorem.

6.1.1 Notation: prefix, fixed arity

The document does not use parentheses, every symbol of the lexicon has a fixed arity, and the reading consumes from the left as many terms as that symbol requires. `@zero` and `@one` are constants of arity zero; `@neg` takes one, while `@add` and `@mul` take two; the same applies to `@Trm` which takes one term and `@Eq`, which takes two.

To read a nested expression the same rule is applied recursively. In the conclusion of `AddAssoc`,

```
@Eq @add @add §a §b §c @add §a @add §b §c
```

`@Eq` takes two terms: the first begins with `@add`, which in turn takes two terms (`@add §a §b`, that is $a + b$, and then `§c`), giving $(a + b) + c$; the second likewise begins with `@add`, which this time takes `§a` and then `@add §b §c` $= b + c$, giving $a + (b + c)$. The line therefore reads $(a + b) + c = a + (b + c)$, which is the associativity named by rule.

6.1.2 Principles and rules of the system

On the vocabulary just shown, the document fixes twenty declarations between principles and schemas: two `$Principium` and eighteen `$Regula`, organised in three groups.

6.1.3 Formation of terms

```
$Principium TrmZero $: @Trm @zero $.
$Principium TrmOne  $: @Trm @one  $.

$Regula TrmNeg $: @Trm §a $- @Trm @neg §a $.

$Regula TrmAdd $:
    @Trm §a
$-  ${  @Trm §b  $-  @Trm @add §a §b  $}
$.
$Regula TrmMul $:
    @Trm §a
$-  ${  @Trm §b  $-  @Trm @mul §a §b  $}
$.
```

The two constants are terms without conditions, so they are principles and not rules. Negation is unary and needs only one premise; addition and multiplication are binary, and each rule makes this explicit with a nested schema: the outer premise supplies `§a`, the inner one `§b`. Both are safe, because every metavariable free in the conclusion already appears in a premise.

6.1.4 Equality

```
$Regula EqRefl $: @Trm §a $- @Eq §a §a $.

$Regula EqSym $: @Eq §a §b $- @Eq §b §a $.

$Regula EqTrans $:
    @Eq §a §b
$-  ${  @Eq §b §c  $-  @Eq §a §c  $}
$.

$Regula EqAdd $:
    @Eq §a §b
$-  ${  @Eq §c §d  $-  @Eq @add §a §c @add §b §d  $}
$.
$Regula EqMul $:
    @Eq §a §b
$-  ${  @Eq §c §d  $-  @Eq @mul §a §c @mul §b §d  $}
$.
$Regula EqNeg $: @Eq §a §b $- @Eq @neg §a @neg §b $.
```

Reflexivity, symmetry and transitivity are the three properties expected, plus congruence with respect to each operation. `EqRefl` is stated on a term. `EqAdd` and `EqMul` have two independent premises, one for each argument of the binary operation, while `EqNeg`, unary, has only one.

6.1.5 The ring axioms

```
$Regula AddAssoc $:
    @Trm §a
$-  ${  @Trm §b
    $-  ${  @Trm §c
        $-  @Eq @add @add §a §b §c @add §a @add §b §c  $} $}
$.
$Regula AddComm $:
    @Trm §a
$-  ${  @Trm §b  $-  @Eq @add §a §b @add §b §a  $}
$.
$Regula AddZero $: @Trm §a $- @Eq @add §a @zero §a $.
$Regula AddNeg  $: @Trm §a $- @Eq @add §a @neg §a @zero $.

$Regula MulAssoc $:
    @Trm §a
$-  ${  @Trm §b
    $-  ${  @Trm §c
        $-  @Eq @mul @mul §a §b §c @mul §a @mul §b §c  $} $}
$.
$Regula MulOne $: @Trm §a $- @Eq @mul §a @one §a $.
$Regula OneMul $: @Trm §a $- @Eq @mul @one §a §a $.

$Regula DistL $:
    @Trm §a
$-  ${  @Trm §b
```

```
    $-  ${  @Trm §c
        $-  @Eq @mul §a @add §b §c @add @mul §a §b @mul §a §c  $} $}
$.
$Regula DistR $:
    @Trm §a
$-  ${  @Trm §b
    $-  ${  @Trm §c
        $-  @Eq @mul @add §a §b §c @add @mul §a §c @mul §b §c  $} $}
$.
```

Addition is associative and commutative, `@zero` is its identity, and every term has an additive inverse. Multiplication is associative, with `@one` as its unit, and distributes over addition. The document does not assume the commutativity of multiplication, which is why `MulOne` and `OneMul` are two distinct rules, as are `DistL` (on the left, $a(b+c) = ab + ac$) and `DistR` (on the right, $(a+b)c = ac + bc$). Addition, on the other hand, does not require the same duplication: `AddZero` fixes neutrality only on the right ($a + 0 = a$), while the left version is obtained from `AddComm`. This is what the derivation `AddZeroL`, discussed further on, proves explicitly rather than assuming it.

6.1.6 A guided proof: `ZeroPlusZero`

The simplest product in the document establishes a concrete fact, free of metavariables: $0 + 0 = 0$.

```
$Productum ZeroPlusZero $:

@Eq @add @zero @zero @zero

$ex

$Probatione

    $Assumo    @Trm @zero                                $per $Principium
  TrmZero $.
    $Assumo    @Trm §a    $- @Eq @add §a    @zero §a     $per $Regulam AddZero
  $.
    $Obtineo   @Trm @zero $- @Eq @add @zero @zero @zero  $cum §a $= @zero $.
    $Reitero   @Trm @zero                                $.
    $Reitero   @Trm @zero $- @Eq @add @zero @zero @zero  $.
    $Consequor               @Eq @add @zero @zero @zero  $.

$.

$.
```

The first step cites `$Principium TrmZero`: the verifier resolves the reference, compares the phrase written at the step with the one declared, they coincide, and it inherits the defects of the principle, none here. `@Trm @zero` enters the context.

The second step cites `$Regulam AddZero` in the same way, but here the item inserted into the context is a schema, not yet applied to anything: `@Trm §a $- @Eq @add §a @zero §a`. The schema is not yet instantiated.

The third step, `$Obtineo`, instantiates this schema. It takes the most recent item (the

schema just cited), applies the substitution `§a $= @zero`, and re-interprets it: here the substitution creates nothing new to recognise, `@zero` is already an atomic lexeme, so the reread instance coincides with the one applied structurally. The verifier compares this instance with what the step declares to establish, `@Trm @zero $- @Eq @add @zero @zero @zero`: they coincide. The instantiated schema, now free of any metavariable, enters the context.

Steps four and five establish nothing new: they bring back to the top of the context, in order, the term `@Trm @zero` and the schema just instantiated.

The sixth step, `$Consequor`, takes the two most recent items: the one on top must be a schema, and in this case it is, `@Trm @zero $- @Eq @add @zero @zero @zero`, and the second must coincide with its premise, here `@Trm @zero`, and it does coincide. It thereby establishes the conclusion, `@Eq @add @zero @zero @zero`.

This is also the last thing the proof context contains at the end of the script, and it is exactly the statement declared by the `Productum`: the final comparison coincides, and the proof is accepted.

6.1.7 A proof with a premise: `TrmMulZero`

The first derivation in the document establishes that the product of a term by `@zero` is still a term: $a\colon \mathrm{Trm} \vdash a \cdot 0\colon \mathrm{Trm}$. Unlike the previous product, the statement is a schema, which requires a premise, and the rule cited within it, `TrmMul`, itself has two nested premises rather than one.

```
$Derivatio TrmMulZero $:

@Trm §a $- @Trm @mul §a @zero

$ex

$Probatione

    $Praemitto @Trm §a $.

    $Assumo    @Trm §a $- ${ @Trm §b    $- @Trm @mul §a §b    $}  $per $Regulam
  TrmMul $.
    $Obtineo   @Trm §a $- ${ @Trm @zero $- @Trm @mul §a @zero $}  $cum §b $=
  @zero $.
    $Reitero   @Trm §a                                            $.
    $Reitero   @Trm §a $- ${ @Trm @zero $- @Trm @mul §a @zero $}  $.
    $Consequor              @Trm @zero $- @Trm @mul §a @zero      $.
    $Assumo    @Trm @zero                                         $per
  $Principium TrmZero $.
    $Reitero               @Trm @zero $- @Trm @mul §a @zero       $.
    $Consequor                            @Trm @mul §a @zero      $.

$.

$.
```

The first step, `$Praemitto`, presents `@Trm §a` as a premise. Let a_P be the assertion representation introduced as a premise. Since it carries a **PHRASE** representation, $free(a_P) = \{\S a\}$, and the corresponding proof item i has $rig(i) = free(a_P) = \{\S a\}$.

The second step cites `$Regulam TrmMul`, whose conclusion is itself a schema: `@Trm §a $- (@Trm §b $- @Trm @mul §a §b)`. The third, `$Obtineo`, instantiates only `§b` with `@zero`. The proof item established by `$Assumo` has an empty rigid set, so `§a` could be instantiated as well; it is left unchanged because the sixth step needs the premise of this schema to match the premise `@Trm §a` introduced at the first step.

Steps four and five bring back to the top of the context the premise `@Trm §a` and the schema just instantiated, as in the previous proof, because step six needs them.

The sixth step, `$Consequor`, then applies `@Trm §a` as the premise of the instantiated schema and establishes its conclusion, which is itself a schema: `@Trm @zero $- @Trm @mul §a @zero`.

Steps seven, eight and nine repeat the same pattern one level deeper.

At this point the script is concluded. Since this is a derivation, the final comparison does not take place directly against the last item, but against the last item with the presented premises discharged around it: the single premise collected (`@Trm §a`, from the first step) is recomposed with the last item established (`@Trm @mul §a @zero`), reconstructing `@Trm §a $- @Trm @mul §a @zero`, which is the declared schema. The proof is accepted.

6.1.8 The remaining derivations, in summary

The other four derivations in the document follow the same pattern as the two just gone through: citation of a rule or a previous theorem, instantiation, application, discharge of the premises. I report the code in full and summarise its logic: those who want to follow them step by step can do so directly on the source given, those who only want the logical thread can skip the boxes.

AddZeroL

```
$Derivatio AddZeroL $:

@Trm §a $- @Eq @add @zero §a §a

$ex

$Probatione

    $Praemitto @Trm §a $.

    $Assumo    @Trm §a    $- ${ @Trm §b $- @Eq @add §a    §b @add §b §a    $}
  $per $Regulam AddComm $.
    $Obtineo   @Trm @zero $- ${ @Trm §a $- @Eq @add @zero §a @add §a @zero $}
  $cum §a $= @zero $, §b $= §a $.
    $Assumo    @Trm @zero
  $per $Principium TrmZero $.
    $Reitero   @Trm @zero $- ${ @Trm §a $- @Eq @add @zero §a @add §a @zero $}
  $.
    $Consequor                 @Trm §a $- @Eq @add @zero §a @add §a @zero
  $.
    $Reitero                   @Trm §a
  $.
    $Reitero                   @Trm §a $- @Eq @add @zero §a @add §a @zero
  $.
```

```
    $Consequor                          @Eq @add @zero §a @add §a @zero
  $.

    $Assumo    @Trm §a $- @Eq @add §a @zero §a  $per $Regulam AddZero $.
    $Reitero   @Trm §a                          $.
    $Reitero   @Trm §a $- @Eq @add §a @zero §a  $.
    $Consequor              @Eq @add §a @zero §a $.

    $Assumo    @Eq §a §b $- ${ @Eq §b §c $- @Eq §a §c $}  $per $Regulam EqTrans
  $.
    $Obtineo   @Eq @add @zero §a @add §a @zero
        $-  ${  @Eq @add §a @zero §a  $-  @Eq @add @zero §a §a  $}
        $cum §a $= @add @zero §a $, §b $= @add §a @zero $, §c $= §a $.
    $Reitero   @Eq @add @zero §a @add §a @zero
     $.
    $Reitero   @Eq @add @zero §a @add §a @zero
        $-  ${  @Eq @add §a @zero §a  $-  @Eq @add @zero §a §a  $}
  $.
    $Consequor         @Eq @add §a @zero §a  $-  @Eq @add @zero §a §a
     $.
    $Reitero           @Eq @add §a @zero §a
     $.
    $Reitero           @Eq @add §a @zero §a  $-  @Eq @add @zero §a §a
     $.
    $Consequor                                   @Eq @add @zero §a §a
     $.

$.

$.
```

`AddZeroL` establishes $0 + a = a$, the neutrality of `@zero` read from the left. The proof instantiates the commutativity of addition with two simultaneous substitutions (`$cum §a $= @zero $, §b $= §a`), obtaining $0 + a = a + 0$, then closes it with the neutrality already known from the right ($a + 0 = a$, the same rule used in `ZeroPlusZero`) by means of a single application of transitivity.

MulZeroIdem

```
$Derivatio MulZeroIdem $:

@Trm §a $- @Eq @mul §a @zero @add @mul §a @zero @mul §a @zero

$ex

$Probatione

    $Praemitto @Trm §a $.

    $Assumo    @Trm §a $- @Eq §a §a  $per $Regulam EqRefl $.
    $Reitero   @Trm §a               $.
    $Reitero   @Trm §a $- @Eq §a §a  $.
```

```
  $Consequor              @Eq §a §a $.

  $Assumo    @Eq @add @zero @zero @zero                                $per
$Productum ZeroPlusZero $.
  $Assumo    @Eq §a §b $- @Eq §b §a                                    $per
$Regulam EqSym $.
  $Obtineo   @Eq @add @zero @zero @zero $- @Eq @zero @add @zero @zero
      $cum §a $= @add @zero @zero $, §b $= @zero $.
  $Reitero   @Eq @add @zero @zero @zero                                $.
  $Reitero   @Eq @add @zero @zero @zero $- @Eq @zero @add @zero @zero   $.
  $Consequor                               @Eq @zero @add @zero @zero   $.

  $Assumo    @Eq §a §b $- ${ @Eq §c §d $- @Eq @mul §a §c @mul §b §d $}  $per
$Regulam EqMul $.
  $Obtineo   @Eq §a §a
      $-  ${  @Eq @zero @add @zero @zero
          $-  @Eq @mul §a @zero @mul §a @add @zero @zero  $}
      $cum §b $= §a $, §c $= @zero $, §d $= @add @zero @zero $.
  $Reitero   @Eq §a §a
    $.
  $Reitero   @Eq §a §a
      $-  ${  @Eq @zero @add @zero @zero
          $-  @Eq @mul §a @zero @mul §a @add @zero @zero  $}
 $.
  $Consequor         @Eq @zero @add @zero @zero
             $-  @Eq @mul §a @zero @mul §a @add @zero @zero
 $.
  $Reitero           @Eq @zero @add @zero @zero
    $.
  $Reitero           @Eq @zero @add @zero @zero
             $-  @Eq @mul §a @zero @mul §a @add @zero @zero
 $.
  $Consequor             @Eq @mul §a @zero @mul §a @add @zero @zero
    $.

  $Assumo    @Trm §a
      $-  ${  @Trm §b
          $-  ${  @Trm §c
              $-  @Eq @mul §a @add §b §c @add @mul §a §b @mul §a §c  $} $}
      $per $Regulam DistL $.
  $Obtineo   @Trm §a
      $-  ${  @Trm @zero
          $-  ${  @Trm @zero
              $-  @Eq @mul §a @add @zero @zero
                      @add @mul §a @zero @mul §a @zero  $} $}
      $cum §b $= @zero $, §c $= @zero $.
  $Reitero   @Trm §a
    $.
  $Reitero   @Trm §a
      $-  ${  @Trm @zero
          $-  ${  @Trm @zero
              $-  @Eq @mul §a @add @zero @zero
```

```
                        @add @mul §a @zero @mul §a @zero  $} $}
 $.
  $Consequor          @Trm @zero
              $-  ${  @Trm @zero
                  $-  @Eq @mul §a @add @zero @zero
                          @add @mul §a @zero @mul §a @zero  $}
 $.
  $Assumo             @Trm @zero
    $per $Principium TrmZero $.
  $Reitero            @Trm @zero
              $-  ${  @Trm @zero
                  $-  @Eq @mul §a @add @zero @zero
                          @add @mul §a @zero @mul §a @zero  $}
 $.
  $Consequor                 @Trm @zero
                      $-  @Eq @mul §a @add @zero @zero
                              @add @mul §a @zero @mul §a @zero
 $.
  $Reitero                   @Trm @zero
    $.
  $Reitero                   @Trm @zero
                      $-  @Eq @mul §a @add @zero @zero
                              @add @mul §a @zero @mul §a @zero
 $.
  $Consequor                     @Eq @mul §a @add @zero @zero
                                  @add @mul §a @zero @mul §a @zero
 $.

  $Assumo    @Eq §a §b $- ${ @Eq §b §c $- @Eq §a §c $}  $per $Regulam EqTrans
$.
  $Obtineo   @Eq @mul §a @zero @mul §a @add @zero @zero
      $-  ${  @Eq @mul §a @add @zero @zero @add @mul §a @zero @mul §a @zero
          $-  @Eq @mul §a @zero @add @mul §a @zero @mul §a @zero  $}
      $cum §a $= @mul §a @zero $,
              §b $= @mul §a @add @zero @zero $,
              §c $= @add @mul §a @zero @mul §a @zero $.
  $Reitero   @Eq @mul §a @zero @mul §a @add @zero @zero
     $.
  $Reitero   @Eq @mul §a @zero @mul §a @add @zero @zero
      $-  ${  @Eq @mul §a @add @zero @zero @add @mul §a @zero @mul §a @zero
          $-  @Eq @mul §a @zero @add @mul §a @zero @mul §a @zero  $}
  $.
  $Consequor         @Eq @mul §a @add @zero @zero @add @mul §a @zero @mul §a
@zero
              $-  @Eq @mul §a @zero @add @mul §a @zero @mul §a @zero
  $.
  $Reitero           @Eq @mul §a @add @zero @zero @add @mul §a @zero @mul §a
@zero $.
  $Reitero           @Eq @mul §a @add @zero @zero @add @mul §a @zero @mul §a
@zero
              $-  @Eq @mul §a @zero @add @mul §a @zero @mul §a @zero
  $.
```

```
    $Consequor             @Eq @mul §a @zero @add @mul §a @zero @mul §a @zero
       $.

$.

$.
```

`MulZeroIdem` establishes $a \cdot 0 = a \cdot 0 + a \cdot 0$, the step that makes $a \cdot 0$ idempotent with respect to addition. This is where the document cites, for the first time, a `Productum` rather than a rule: `$per $Productum ZeroPlusZero` retrieves $0 + 0 = 0$, reversed by symmetry into $0 = 0 + 0$ and used as the second argument of a product congruence ($a = a$, $0 = 0 + 0$, hence $a \cdot 0 = a \cdot (0 + 0)$). Left distributivity, cited immediately after, has three nested premises rather than two and is consumed with three `$Consequor` in sequence rather than two, giving $a \cdot (0 + 0) = a \cdot 0 + a \cdot 0$. A transitivity closes the chain.

AddIdem

```
$Derivatio AddIdem $:

@Trm §a $- ${ @Eq §a @add §a §a $- @Eq §a @zero $}

$ex

$Probatione

    $Praemitto @Trm §a $.

    $Assumo    @Trm §a $- @Trm @neg §a  $per $Regulam TrmNeg $.
    $Reitero   @Trm §a                  $.
    $Reitero   @Trm §a $- @Trm @neg §a  $.
    $Consequor              @Trm @neg §a $.

    $Assumo    @Trm §a $- @Eq §a §a  $per $Regulam EqRefl $.
    $Reitero   @Trm §a               $.
    $Reitero   @Trm §a $- @Eq §a §a  $.
    $Consequor              @Eq §a §a $.
    $Assumo    @Trm §a      $- @Eq §a      §a       $per $Regulam EqRefl $.
    $Obtineo   @Trm @neg §a $- @Eq @neg §a @neg §a  $cum §a $= @neg §a $.
    $Reitero   @Trm @neg §a                         $.
    $Reitero   @Trm @neg §a $- @Eq @neg §a @neg §a  $.
    $Consequor                 @Eq @neg §a @neg §a  $.

    $Assumo    @Trm §a $- @Eq @add §a @zero §a          $per $Regulam AddZero $.
    $Reitero   @Trm §a                                  $.
    $Reitero   @Trm §a $- @Eq @add §a @zero §a          $.
    $Consequor              @Eq @add §a @zero §a        $.
    $Assumo    @Eq §a §b $- @Eq §b §a                   $per $Regulam EqSym $.
    $Obtineo   @Eq @add §a @zero §a $- @Eq §a @add §a @zero
        $cum §a $= @add §a @zero $, §b $= §a $.
    $Reitero   @Eq @add §a @zero §a                     $.
    $Reitero   @Eq @add §a @zero §a $- @Eq §a @add §a @zero $.
    $Consequor                         @Eq §a @add §a @zero $.
```

```
  $Assumo    @Trm §a $- @Eq @add §a @neg §a @zero     $per $Regulam AddNeg $.
  $Reitero   @Trm §a                                  $.
  $Reitero   @Trm §a $- @Eq @add §a @neg §a @zero     $.
  $Consequor              @Eq @add §a @neg §a @zero   $.
  $Assumo    @Eq §a §b $- @Eq §b §a                   $per $Regulam EqSym $.
  $Obtineo   @Eq @add §a @neg §a @zero $- @Eq @zero @add §a @neg §a
      $cum §a $= @add §a @neg §a $, §b $= @zero $.
  $Reitero   @Eq @add §a @neg §a @zero                $.
  $Reitero   @Eq @add §a @neg §a @zero $- @Eq @zero @add §a @neg §a $.
  $Consequor                              @Eq @zero @add §a @neg §a $.

  $Assumo    @Eq §a §b $- ${ @Eq §c §d $- @Eq @add §a §c @add §b §d $}  $per
$Regulam EqAdd $.
  $Obtineo   @Eq §a §a
      $-  ${  @Eq @zero @add §a @neg §a
          $-  @Eq @add §a @zero @add §a @add §a @neg §a  $}
      $cum §b $= §a $, §c $= @zero $, §d $= @add §a @neg §a $.
  $Reitero   @Eq §a §a                                                  $.
  $Reitero   @Eq §a §a
      $-  ${  @Eq @zero @add §a @neg §a
          $-  @Eq @add §a @zero @add §a @add §a @neg §a  $}          $.
  $Consequor         @Eq @zero @add §a @neg §a
             $-  @Eq @add §a @zero @add §a @add §a @neg §a          $.
  $Reitero           @Eq @zero @add §a @neg §a                          $.
  $Reitero           @Eq @zero @add §a @neg §a
             $-  @Eq @add §a @zero @add §a @add §a @neg §a          $.
  $Consequor             @Eq @add §a @zero @add §a @add §a @neg §a       $.

  $Assumo    @Trm §a
      $-  ${  @Trm §b
          $-  ${  @Trm §c
              $-  @Eq @add @add §a §b §c @add §a @add §b §c  $} $}
      $per $Regulam AddAssoc $.
  $Obtineo   @Trm §a
      $-  ${  @Trm §a
          $-  ${  @Trm @neg §a
              $-  @Eq @add @add §a §a @neg §a @add §a @add §a @neg §a  $} $}
      $cum §b $= §a $, §c $= @neg §a $.
  $Reitero   @Trm §a
    $.
  $Reitero   @Trm §a
      $-  ${  @Trm §a
          $-  ${  @Trm @neg §a
              $-  @Eq @add @add §a §a @neg §a @add §a @add §a @neg §a  $} $}
 $.
  $Consequor         @Trm §a
             $-  ${  @Trm @neg §a
                 $-  @Eq @add @add §a §a @neg §a @add §a @add §a @neg §a  $}
 $.
  $Reitero           @Trm §a
    $.
```

```
  $Reitero           @Trm §a
             $-  ${  @Trm @neg §a
                 $-  @Eq @add @add §a §a @neg §a @add §a @add §a @neg §a  $}
 $.
  $Consequor                 @Trm @neg §a
                   $-  @Eq @add @add §a §a @neg §a @add §a @add §a @neg §a
 $.
  $Reitero                   @Trm @neg §a
    $.
  $Reitero                   @Trm @neg §a
                   $-  @Eq @add @add §a §a @neg §a @add §a @add §a @neg §a
 $.
  $Consequor                     @Eq @add @add §a §a @neg §a @add §a @add §a
@neg §a $.
  $Assumo    @Eq §a §b $- @Eq §b §a  $per $Regulam EqSym $.
  $Obtineo   @Eq @add @add §a §a @neg §a @add §a @add §a @neg §a
      $-  @Eq @add §a @add §a @neg §a @add @add §a §a @neg §a
      $cum §a $= @add @add §a §a @neg §a $, §b $= @add §a @add §a @neg §a $.
  $Reitero   @Eq @add @add §a §a @neg §a @add §a @add §a @neg §a
    $.
  $Reitero   @Eq @add @add §a §a @neg §a @add §a @add §a @neg §a
      $-  @Eq @add §a @add §a @neg §a @add @add §a §a @neg §a
 $.
  $Consequor     @Eq @add §a @add §a @neg §a @add @add §a §a @neg §a
    $.

  $Praemitto @Eq §a @add §a §a $.
  $Assumo    @Eq §a §b $- @Eq §b §a  $per $Regulam EqSym $.
  $Obtineo   @Eq §a @add §a §a $- @Eq @add §a §a §a
      $cum §a $= §a $, §b $= @add §a §a $.
  $Reitero   @Eq §a @add §a §a                        $.
  $Reitero   @Eq §a @add §a §a $- @Eq @add §a §a §a   $.
  $Consequor                      @Eq @add §a §a §a   $.
  $Assumo    @Eq §a §b $- ${ @Eq §c §d $- @Eq @add §a §c @add §b §d $}  $per
$Regulam EqAdd $.
  $Obtineo   @Eq @add §a §a §a
      $-  ${  @Eq @neg §a @neg §a
          $-  @Eq @add @add §a §a @neg §a @add §a @neg §a  $}
      $cum §a $= @add §a §a $, §b $= §a $, §c $= @neg §a $, §d $= @neg §a $.
  $Reitero   @Eq @add §a §a §a                                          $.
  $Reitero   @Eq @add §a §a §a
      $-  ${  @Eq @neg §a @neg §a
          $-  @Eq @add @add §a §a @neg §a @add §a @neg §a  $}          $.
  $Consequor         @Eq @neg §a @neg §a
             $-  @Eq @add @add §a §a @neg §a @add §a @neg §a          $.
  $Reitero           @Eq @neg §a @neg §a                                $.
  $Reitero           @Eq @neg §a @neg §a
             $-  @Eq @add @add §a §a @neg §a @add §a @neg §a          $.
  $Consequor             @Eq @add @add §a §a @neg §a @add §a @neg §a      $.

  $Assumo    @Eq §a §b $- ${ @Eq §b §c $- @Eq §a §c $}  $per $Regulam EqTrans
$.
```

```
    $Obtineo   @Eq §a @add §a @zero
        $-  ${  @Eq @add §a @zero @add §a @add §a @neg §a
            $-  @Eq §a @add §a @add §a @neg §a  $}
        $cum §a $= §a $, §b $= @add §a @zero $, §c $= @add §a @add §a @neg §a $.
    $Reitero   @Eq §a @add §a @zero                                          $.
    $Reitero   @Eq §a @add §a @zero
        $-  ${  @Eq @add §a @zero @add §a @add §a @neg §a
            $-  @Eq §a @add §a @add §a @neg §a  $}                       $.
    $Consequor         @Eq @add §a @zero @add §a @add §a @neg §a
                $-  @Eq §a @add §a @add §a @neg §a                       $.
    $Reitero           @Eq @add §a @zero @add §a @add §a @neg §a             $.
    $Reitero           @Eq @add §a @zero @add §a @add §a @neg §a
                $-  @Eq §a @add §a @add §a @neg §a                       $.
    $Consequor             @Eq §a @add §a @add §a @neg §a                   $.

    $Assumo    @Eq §a §b $- ${ @Eq §b §c $- @Eq §a §c $}  $per $Regulam EqTrans
  $.
    $Obtineo   @Eq §a @add §a @add §a @neg §a
        $-  ${  @Eq @add §a @add §a @neg §a @add @add §a §a @neg §a
            $-  @Eq §a @add @add §a §a @neg §a  $}
        $cum §a $= §a $,
                §b $= @add §a @add §a @neg §a $,
                §c $= @add @add §a §a @neg §a $.
    $Reitero   @Eq §a @add §a @add §a @neg §a                                $.
    $Reitero   @Eq §a @add §a @add §a @neg §a
        $-  ${  @Eq @add §a @add §a @neg §a @add @add §a §a @neg §a
            $-  @Eq §a @add @add §a §a @neg §a  $}                       $.
    $Consequor         @Eq @add §a @add §a @neg §a @add @add §a §a @neg §a
                $-  @Eq §a @add @add §a §a @neg §a                       $.
    $Reitero           @Eq @add §a @add §a @neg §a @add @add §a §a @neg §a   $.
    $Reitero           @Eq @add §a @add §a @neg §a @add @add §a §a @neg §a
                $-  @Eq §a @add @add §a §a @neg §a                       $.
    $Consequor             @Eq §a @add @add §a §a @neg §a                   $.

    $Assumo    @Eq §a §b $- ${ @Eq §b §c $- @Eq §a §c $}  $per $Regulam EqTrans
  $.
    $Obtineo   @Eq §a @add @add §a §a @neg §a
        $-  ${  @Eq @add @add §a §a @neg §a @add §a @neg §a
            $-  @Eq §a @add §a @neg §a  $}
        $cum §a $= §a $,
                §b $= @add @add §a §a @neg §a $,
                §c $= @add §a @neg §a $.
    $Reitero   @Eq §a @add @add §a §a @neg §a                                $.
    $Reitero   @Eq §a @add @add §a §a @neg §a
        $-  ${  @Eq @add @add §a §a @neg §a @add §a @neg §a
            $-  @Eq §a @add §a @neg §a  $}                               $.
    $Consequor         @Eq @add @add §a §a @neg §a @add §a @neg §a
                $-  @Eq §a @add §a @neg §a                               $.
    $Reitero           @Eq @add @add §a §a @neg §a @add §a @neg §a           $.
    $Reitero           @Eq @add @add §a §a @neg §a @add §a @neg §a
                $-  @Eq §a @add §a @neg §a                               $.
    $Consequor             @Eq §a @add §a @neg §a                           $.
```

```
    $Assumo    @Eq §a §b $- ${ @Eq §b §c $- @Eq §a §c $}  $per $Regulam EqTrans
  $.
    $Obtineo   @Eq §a @add §a @neg §a
        $-  ${  @Eq @add §a @neg §a @zero  $-  @Eq §a @zero  $}
        $cum §a $= §a $, §b $= @add §a @neg §a $, §c $= @zero $.
    $Reitero   @Eq §a @add §a @neg §a                                    $.
    $Reitero   @Eq §a @add §a @neg §a
        $-  ${  @Eq @add §a @neg §a @zero  $-  @Eq §a @zero  $}          $.
    $Consequor         @Eq @add §a @neg §a @zero  $-  @Eq §a @zero          $.
    $Reitero           @Eq @add §a @neg §a @zero                            $.
    $Reitero           @Eq @add §a @neg §a @zero  $-  @Eq §a @zero          $.
    $Consequor                                        @Eq §a @zero          $.

  $.

  $.
```

`AddIdem` is the longest derivation and the only one with two distinct premises: that `§a` is a term, and that `§a` is idempotent with respect to addition ($a = a + a$). It does not mention multiplication in any way – it is a general fact about any idempotent element of a ring, proved by building four equalities in succession ($a = a+0$, then $a+0 = a+(a+(-a))$ by congruence, then $a + (a + (-a)) = (a + a) + (-a)$ by associativity, finally $(a + a) + (-a) = a + (-a)$ using the second premise) and chaining them with four applications of transitivity, one at a time, down to $a = 0$.

MulZeroR

```
  $Derivatio MulZeroR $:

  @Trm §a $- @Eq @mul §a @zero @zero

  $ex

  $Probatione

      $Praemitto @Trm §a $.

      $Assumo    @Trm §a $- @Trm @mul §a @zero  $per $Derivationem TrmMulZero $.
      $Reitero   @Trm §a                        $.
      $Reitero   @Trm §a $- @Trm @mul §a @zero  $.
      $Consequor              @Trm @mul §a @zero $.

      $Assumo    @Trm §a
          $-  @Eq @mul §a @zero @add @mul §a @zero @mul §a @zero
          $per $Derivationem MulZeroIdem $.
      $Reitero   @Trm §a                                                  $.
      $Reitero   @Trm §a
          $-  @Eq @mul §a @zero @add @mul §a @zero @mul §a @zero          $.
      $Consequor     @Eq @mul §a @zero @add @mul §a @zero @mul §a @zero      $.

      $Assumo    @Trm §a $- ${ @Eq §a @add §a §a $- @Eq §a @zero $}
```

```
        $per $Derivationem AddIdem $.
    $Obtineo   @Trm @mul §a @zero
        $-  ${  @Eq @mul §a @zero @add @mul §a @zero @mul §a @zero
            $-  @Eq @mul §a @zero @zero  $}
        $cum §a $= @mul §a @zero $.
    $Reitero   @Trm @mul §a @zero                                          $.
    $Reitero   @Trm @mul §a @zero
        $-  ${  @Eq @mul §a @zero @add @mul §a @zero @mul §a @zero
            $-  @Eq @mul §a @zero @zero  $}                              $.
    $Consequor         @Eq @mul §a @zero @add @mul §a @zero @mul §a @zero
                $-  @Eq @mul §a @zero @zero                              $.
    $Reitero           @Eq @mul §a @zero @add @mul §a @zero @mul §a @zero     $.
    $Reitero           @Eq @mul §a @zero @add @mul §a @zero @mul §a @zero
                $-  @Eq @mul §a @zero @zero                              $.
    $Consequor                                @Eq @mul §a @zero @zero         $.

$.

$.
```

`MulZeroR` is the final theorem, $a \cdot 0 = 0$. MulZeroR concludes the derivation chain by combining TrmMulZero, MulZeroIdem and AddIdem. It cites `TrmMulZero` to know that $a \cdot 0$ is a term, `MulZeroIdem` to know that it is idempotent, and finally `AddIdem` instantiated precisely on $a \cdot 0$ by means of `$Obtineo`, all this to conclude that an element with these two properties is zero.

6.2 Testing with Hspec

The validation of the implementation was carried out by means of an automated test suite based on *Hspec*, a framework for testing Haskell programs. The suite adopts an *example-based* approach: each case builds a representative input and compares the observed behaviour with the expected result, by means of equality or membership assertions.

No *property-based testing* frameworks, such as QuickCheck, were employed. The aim is to verify in a targeted way the syntactic and semantic rules implemented in the parser and in the verifier, including both the valid cases and the expected error conditions. Alongside the Hspec suite there is a second suite, called `style`, which uses HLint to automatically check conventions and stylistic issues in the sources contained in the `src` directory.

File	Area	Tests
`Toolkits/ValidatorSpec`	verifier	151
`Printer/PrinterSpec`	pretty-printer	25
`Parser/ConstructsSpec`	parser	23
`Parser/SegmentationSpec`	parser	21
`Parser/ProofsSpec`	parser	19
`Parser/TokeniserSpec`	parser	19
`Parser/ParserSpec`	parser	18
`Parser/ErrorSpec`	parser	9
`Validator/Examples/MIUSpec`	end-to-end	4
`Parser/KeywordsSpec`	parser	3
`Validator/Examples/PCPSpec`	end-to-end	2
Total	verifier	151
Total	parser	112
Total	pretty-printer	25
Total	end-to-end	6
	Total	**294**

Table 6.1: Distribution of the 294 active tests by specification file and area.

Table 6.2: Organisation and scope of the test suite.

Module	Behaviours verified
`KeywordsSpec`	Correspondence between extended and canonical spellings of the terminals, including spellings shared by distinct terminals.
`TokeniserSpec`	Tokenisation of terminals, handling of active spellings, selection of the longest active match, nested comments with their *escaping*, and informal text between one construct and the next.
`SegmentationSpec`	Segmentation of `WORD`, multi-character symbols, non-ASCII symbols and aliases; handling of missing or ambiguous readings and laziness of the enumeration; reading of object text, including inclusores, escapes and aliases.
`ConstructsSpec`	Parsing of the `ALPHABETUM`, `INTERVALLA`, `INCLUSORES`, `LEXICON` and `MUTABILIA` declarations; intervals, aliases, Unicode symbols and segmentation ambiguity.
`ProofsSpec`	Parsing of `PRODUCTUM` and `DERIVATIO` scripts; proof instructions, assignments, `$Vacuum`, qualified references, distinction between phrase and schema, and *span* of the steps.

Module	Behaviours verified
`ParserSpec`	Structure of the document, nested and reopened fragments, isolation of the vocabulary, interpretation parameters, parsing errors and UTF-8 coordinates of the diagnostics.
`ErrorSpec`	Rendering of the parser's diagnostics: ambiguous readings, highlighting of the portions involved, ANSI output and rendering free of colour codes.
`ValidatorSpec`	Validation of declarations and vocabulary, *safety* of phrases and schemas, `PRODUCTUM` and `DERIVATIO` scripts, premises and their discharge, substitutions, reinterpretation, rigidity, circularity, qualified references, export between fragments, reopening of `$Lingua`, propagation of defects, purity, dependencies, certification levels of the diagnostics, step and source position, final verdict and report.
`MIUSpec`	*End-to-end* verification of derivations of the MIU formal system and of the entire corresponding fragment.
`PCPSpec`	*End-to-end* verification of a derivation of the PCP system, including a schema with three antecedents.
`PrinterSpec`	Serialisation of the AST elements, primitive tokens, dialects, identifiers, informal prose blobs and formatting edge cases.

6.3 End-to-End Validation

This section evaluates the complete Hypatia validation pipeline through the HSpec test suite and larger, independently constructed test cases. The latter include documents generated by the Metis translator, the official conformance corpus, and the layered formalisations of Hypatia. All tests discussed in this section completed successfully and produced the expected outcome.

6.3.1 Cases generated by the Metis Metamath translator

To broaden the *end-to-end* tests, the Metis component was employed, developed by Virginia Antonia Esposito [7], dedicated to translating Metamath files into Hypatia documents. The files produced by the translator were then run through Hypatia's normal pipeline, from reading the source to validating the declarations and the proof scripts.

The translated and tested files are four. Each translates a Metamath library. We have two small-sized databases (`miu.mm` and `demo0-includer.mm`, which in turn includes `demo0-includee.mm`), and two larger libraries (`big-unifier.mm`, a test of unification and substitution, and `hol.mm`).

For all four, the `No errors detected.` outcome of the Metamath verification preceding the translation holds, so they are proofs already ascertained, and the expectation is that the translated versions will also validate in Hypatia.

File	Lines	Size	Time	Outcome	Expected
`miu.hypatia`	266	12 KB	0.02 s	accepted	accepted
`demo0-includer.hypatia`	185	10 KB	0.20 s	accepted	accepted
`big-unifier.hypatia`	23,450	5.3 MB	27.6 s	accepted	accepted
`hol.hypatia`	170,936	17.1 MB	35.4 s	accepted	accepted

All files validate without errors, including `hol.hypatia`: almost 171,000 lines and 17 MB, the largest document submitted to the verifier in this work. `demo0-includer.hypatia` exercises nested fragments and qualified references such as `$per $Regulam a2 $@ demo0-includee`

6.3.2 Corpus-Based Testing

Besides the unit tests and the end-to-end cases produced by means of the Metis translator, the verification was extended to Hypatia's official conformance corpus, already available in the `hypatia-language` repository. The corpus was not created as part of this work: it constitutes an independent collection of complete documents, accompanied by the expected verdict.

Since the implementation developed concerns exclusively the `Sunya` dialect, the analysis was limited to the `tst/sunya` directory. It contains 106 documents, but only 87 of these are testable under Unica.

Each document was interpreted independently, starting from an empty *environment*:

```
hypatia validator "<file>.hypatia"
```

repeated once per document.

The directory structure expresses the expected result:

`valid/pure`	53	accepted documents, in which every construct is genuinely established;
`valid/pseudo`	3	accepted documents in which at least one construct rests solely on pseudo-proofs;
`invalid/syntax`	22	documents that do not reach `Syntx` certification;
`invalid/semantics`	28	documents that reach `Syntx`, but not `Semnt`.

	Documents	Verdict matches expected
Testable under `Unica`	87	87 OK
Out of coverage (`Omnes`, `Aliqua`, `Valida`)	19	—

6.3.3 Testing the Layered Formalisations

A further end-to-end test was run on the documents in the `hyp` directory of the `hypatia-language` repository. Unlike the conformance corpus, whose files are independent documents, these files constitute a cumulative hierarchy of formalisations of the Hypatia language. Each level introduces elements used by the following levels.

Since the implementation under consideration supports the `Sunya` dialect, the run was carried out by chaining, in this order, the formalisations written in Sunya of:

1. `Elementa.hypatia`
2. `Fundamenta.sunya.hypatia`

3. `Scriptum.sunya.hypatia`
4. `Hypatia.sunya.hypatia`
5. `Sunya.hypatia`
6. `Sifr.hypatia`
7. `Zero.hypatia`

The execution of several files on the same invocation of the tool constitutes a single logical execution. The test verifies that the export of names, the resolution of qualified references and the persistence of the state work correctly even across the boundary of a single file.

File	Lines	Size	Outcome
`Elementa.hypatia`	101	5.3 KB	OK
`Fundamenta.sunya.hypatia`	1,226	67.8 KB	OK
`Scriptum.sunya.hypatia`	97	3.6 KB	OK
`Hypatia.sunya.hypatia`	617	38.6 KB	OK, 16 warnings
`Sunya.hypatia`	391	17.4 KB	OK
`Sifr.hypatia`	98	3.7 KB	OK
`Zero.hypatia`	98	3.7 KB	OK

The 16 warnings, all on Hypatia.sunya.hypatia, are $Divino pseudo-proofs, none of which is an error.

```
hypatia validator "Elementa.hypatia" \
                  "Fundamenta.sunya.hypatia" \
                  "Scriptum.sunya.hypatia" \
                  "Hypatia.sunya.hypatia" \
                  "Sunya.hypatia" \
                  "Sifr.hypatia" \
                  "Zero.hypatia"
```

```
[OK] ../hypatia-language/hyp/Elementa.hypatia successfully validated.
[OK] ../hypatia-language/hyp/Fundamenta.sunya.hypatia successfully validated.
[OK] ../hypatia-language/hyp/Scriptum.sunya.hypatia successfully validated.

Lingua Elementa

└─ Lingua Fundamenta

  └─ Lingua Scriptum

    └─ Lingua Hypatia
      ├─ Derivatio Dcmn:Blob:Pre 158:3
      │  ⚠ warning: proof of Dcmn:Blob:Pre uses $Divino and is therefore a pseudo-proof
      ├─ Derivatio Dcmn:Blob:Post 173:3
      │  ⚠ warning: proof of Dcmn:Blob:Post uses $Divino and is therefore a pseudo-proof
      ├─ Derivatio Dcmn:Hypt:Pre 188:3
      │  ⚠ warning: proof of Dcmn:Hypt:Pre uses $Divino and is therefore a pseudo-proof
      ├─ Derivatio Dcmn:Hypt:Post 203:3
      │  ⚠ warning: proof of Dcmn:Hypt:Post uses $Divino and is therefore a pseudo-proof
      ├─ Derivatio Dcmn:Ling:Pre 218:3
      │  ⚠ warning: proof of Dcmn:Ling:Pre uses $Divino and is therefore a pseudo-proof
      ├─ Derivatio Dcmn:Ling:Post 233:3
      │  ⚠ warning: proof of Dcmn:Ling:Post uses $Divino and is therefore a pseudo-proof
      ├─ Derivatio Lang:Blob:Pre 453:3
      │  ⚠ warning: proof of Lang:Blob:Pre uses $Divino and is therefore a pseudo-proof
      ├─ Derivatio Lang:Blob:Post 468:3
      │  ⚠ warning: proof of Lang:Blob:Post uses $Divino and is therefore a pseudo-proof
      ├─ Derivatio Lang:Hypt:Pre 483:3
      │  ⚠ warning: proof of Lang:Hypt:Pre uses $Divino and is therefore a pseudo-proof
      ├─ Derivatio Lang:Hypt:Post 498:3
      │  ⚠ warning: proof of Lang:Hypt:Post uses $Divino and is therefore a pseudo-proof
      ├─ Derivatio Lang:Ling:Pre 513:3
      │  ⚠ warning: proof of Lang:Ling:Pre uses $Divino and is therefore a pseudo-proof
      ├─ Derivatio Lang:Ling:Post 528:3
      │  ⚠ warning: proof of Lang:Ling:Post uses $Divino and is therefore a pseudo-proof
      ├─ Derivatio Lang:Cnst:Pre 543:3
      │  ⚠ warning: proof of Lang:Cnst:Pre uses $Divino and is therefore a pseudo-proof
      ├─ Derivatio Lang:Cnst:Post 558:3
      │  ⚠ warning: proof of Lang:Cnst:Post uses $Divino and is therefore a pseudo-proof
      ├─ Derivatio PofC:Empty 584:3
      │  ⚠ warning: proof of PofC:Empty uses $Divino and is therefore a pseudo-proof
      └─ Derivatio PofC:Blob 598:3
         ⚠ warning: proof of PofC:Blob uses $Divino and is therefore a pseudo-proof
[OK] ../hypatia-language/hyp/Hypatia.sunya.hypatia successfully validated.
[OK] ../hypatia-language/hyp/Sunya.hypatia successfully validated.
[OK] ../hypatia-language/hyp/Sifr.hypatia successfully validated.
[OK] ../hypatia-language/hyp/Zero.hypatia successfully validated.
```

Figure 6.1: Output result of the validation process. The colours have been adjusted slightly to make them easier to see.

Chapter 7

Conclusions

Aristarchus verifies Sunya documents in Unica mode and reports the position of each diagnostic in the source. All tests of the suite pass. On the Sunya conformance corpus, the verifier was run on the 87 documents testable under Unica; the other 19 use Omnes, Aliqua or Valida and are outside the scope of this work. The verifier accepts the case study of Chapter 6 and the self-formalisation hierarchy, from Elementa to the three dialects.

7.1 Future developments

The development of Aristarchus is not finished, and future developments are planned, among which:

Diagnostics with suggestions. The diagnostics today report the position in the source, the number of the step when the error arises in a proof, and the tree structure of the document in which it is nested. What is missing is the suggestion of a correction. It could be arranged so that, for Aristarchus errors linked to a name, a reference that does not identify anything, or a word absent from the declared vocabulary, the most similar name or word among those in force at that point in the document is suggested.

State monad for the Env. Hypatia's parser represents the vocabulary that is built while reading the document as a `StateT`. The verifier, instead, has the `Env` pass explicitly through the arguments and the return value of every function, in the form `Env -> Input -> (Env, Output)`. The two parts of the code could share the same style, replacing the explicit passing in the verifier too with a state monad.

Other dialects and other modes. Aristarchus today verifies only Sunya in Unica mode. Support for the Sifr and Zero dialects and for the Omnes, Valida and Aliqua modes is left for future work

Bibliography

[1] J. Alper et al. *Leiden Declaration on Artificial Intelligence and Mathematics*. 2026. DOI: 10.5281/zenodo.20302944.

[2] Danil Annenkov et al. "Extracting Functional Programs from Coq, in Coq". In: *Journal of Functional Programming* 32 (2022), e11:1–60. DOI: 10.1017/S0956796822000077.

[3] Grzegorz Bancerek et al. "The Role of the Mizar Mathematical Library for Interactive Proof Development in Mizar". In: *Journal of Automated Reasoning* 61.1-4 (June 2018), pp. 9–32. ISSN: 0168-7433. DOI: 10.1007/s10817-017-9440-6.

[4] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. ISBN: 978-3-642-05880-6. DOI: 10.1007/978-3-662-07964-5.

[5] Robert S. Boyer et al. "The QED Manifesto". In: *Automated Deduction — CADE-12*. Ed. by Alan Bundy. Vol. 814. Lecture Notes in Artificial Intelligence. Springer-Verlag, 1994, pp. 238–251. URL: http://www.cs.kun.nl/~freek/qed/qed.html.

[6] Mario Carneiro. "Conversion of HOL Light Proofs into Metamath". In: *Journal of Formalized Reasoning* 9.1 (2016). Preprint: arXiv:1412.8091 (2014), pp. 187–200. DOI: 10.6092/issn.1972-5787/4596. arXiv: 1412.8091 [cs.LO].

[7] Virginia Esposito. "Hypatia Metis per Metamath: Un traduttore da Metamath a Hypatia". Tesi di Laurea Triennale. Università degli Studi di Napoli Federico II, 2026.

[8] Herman Geuvers. "Proof Assistants: History, Ideas and Future". In: *Sadhana* 34.1 (Feb. 2009), pp. 3–25. DOI: 10.1007/s12046-009-0001-5. URL: https://www.cs.ru.nl/~herman/PUBS/proofassistants.pdf.

[9] Adam Grabowski. "Mizar in a Nutshell". In: *Journal of Formalized Reasoning* 3.2 (2010), pp. 153–245. DOI: 10.6092/issn.1972-5787/1980.

[10] John Harrison. "Proof Style". In: *Types for Proofs and Programs: International Workshop TYPES'96*. Ed. by Eduardo Giménez and Christine Paulin-Mohring. Vol. 1512. Lecture Notes in Computer Science. Aussois, France: Springer-Verlag, 1996, pp. 154–172.

[11] Norman Megill and David A. Wheeler. *Metamath: A Computer Language for Mathematical Proofs*. Lulu Press, 2019. ISBN: 978-0-359-70223-7. URL: https://www.lulu.com/shop/norman-megill-and-david-a-wheeler/metamath-a-computer-language-for-mathematical-proofs/hardcover/product-24129769.html.

[12] Leonardo de Moura. *Postmortem for Lean Kernel Soundness Bug*. Lean FRO Blog. Aug. 2026. URL: https://leodemoura.github.io/blog/2026-8-1-postmortem-for-kernel-soundness-bug-14576/.

[13] Leonardo de Moura and Sebastian Ullrich. "The Lean 4 Theorem Prover and Programming Language". In: *Conference on Automated Deduction'21*. Lecture Notes in Computer Science 12699. Springer, 2021, pp. 625–635. DOI: 10.1007/978-3-030-79876-5_37.

[14] Leonardo de Moura et al. "The Lean Theorem Prover (System Description)". In: *Conference on Automated Deduction'25*. Lecture Notes in Computer Science 9195. Springer, 2015, pp. 378–388. DOI: 10.1007/978-3-319-21401-6_26.

[15] John Alan Robinson. "A Machine-Oriented Logic Based on the Resolution Principle". In: *Journal of the ACM* 12.1 (Jan. 1965), pp. 23–41. DOI: 10.1145/321250.321253.

[16] John Alan Robinson and Andrei Voronkov, eds. *Handbook of Automated Reasoning*. Elsevier and MIT Press, 2001. ISBN: 0-444-50813-9.

[17] Matthieu Sozeau. "Touring the MetaCoq Project (Invited Paper)". In: *Logical Frameworks and Meta-Languages: Theory and Practice'21*. Electronic Proceedings in Theoretical Computer Science 337. 2021, pp. 13–29. DOI: 10.4204/EPTCS.337.2.

[18] Freek Wiedijk. "A Synthesis of the Procedural and Declarative Styles of Interactive Theorem Proving". In: *Logical Methods in Computer Science* 8.1 (Mar. 2012). Article 30. ISSN: 1860-5974. DOI: 10.2168/LMCS-8(1:30)2012. arXiv: 1201.3601 [cs.LO].