

Foundations and Elaboration of Stella Based on Dependent Type Theory

Version 0.0.1 | Last Updated: 2026-10-08

ZHU ZHEHAO
Luna Flow
zhehao0827@163.com

Preface

The difficulty I initially encountered with dependent type theory did not arise from the formalism itself, but from the way its foundations are often presented. Many expositions assume a shared background in logic or type systems and proceed by introducing abstractions whose conceptual dependencies remain implicit. Without a clear account of how these ideas arise from computation, the theory can appear inaccessible, even when its individual rules are formally precise.

This treatise is written with a different assumption. Understanding dependent type theory should not require prior immersion in a particular academic tradition. When its fundamental notions are introduced in an order that reflects their actual dependencies, and when each abstraction is tied to a concrete computational interpretation, the theory becomes intelligible through ordinary reasoning rather than specialized training.

The approach taken here reconstructs the core of dependent type theory from first principles, making explicit the mechanisms that are often hidden behind automation or convention. By exposing how types, terms, identity, and induction interact at the level of the core calculus, the aim is to provide a path into the subject that is both rigorous and conceptually transparent.

With this perspective established, the following chapters develop the foundations and elaboration of the Stella kernel.

Abstract

The theoretical foundation of modern Proof Assistants (PAs) resides in Dependent Type Theory, a framework where the boundary between programs and mathematical proofs is erased. The profound expressive power of these systems enables mathematicians to formalize complex proofs as executable programs, effectively shifting the locus of logical trust from fallible manual verification to machine-certified type checking via the Curry-Howard correspondence. As the demand for formal verification grows in both software engineering and pure mathematics, the need for high-performance, accessible, and theoretically flexible kernels becomes paramount.

In this work, we introduce the design and implementation of Stella, a high-performance PA kernel developed in Moonbit. We demonstrate that Moonbit’s sophisticated functional programming paradigm, efficient memory management, and superior execution speed provide an ideal infrastructure for constructing a rigorous and maintainable core for formal verification. Drawing methodological inspiration from the seminal work of Ulf Norell on the foundations of Agda and the pedagogical frameworks of Andres Löf et al., Stella implements a unified term syntax where types are treated as first-class citizens. This design choice simplifies the internal calculus and facilitates a robust bidirectional elaboration engine that manages the transition from surface-level expressions to a fully annotated core representation.

Beyond its theoretical and engineering foundations, this treatise presents a systematic exposition aligned with the structure of the Stella kernel. Rather than assuming prior familiarity with formal methods or dependent type theory, the presentation reconstructs core concepts from first principles, such as dependent types, universe hierarchies, identity, and induction, closely aligning the exposition with the structure of the kernel itself.

Furthermore, distinguishing itself from contemporary systems like Lean 4, which prioritize the Calculus of Inductive Constructions (CIC), Stella is architecturally designed to maintain intrinsic compatibility with Homotopy Type Theory (HoTT). By prioritizing support for the Univalence Principle and Higher Inductive Types (HITs), Stella avoids the limitations of strict observational equality, treating identity as a path-based topological structure. This project serves as a cornerstone of the Luna Flow mathematical ecosystem, striving to bridge the gap between high-performance software engineering, educational clarity, and cutting-edge constructive mathematics within the Moonbit infrastructure.

As an evolving foundational document, this treatise will be updated iteratively to reflect the ongoing development of the Stella kernel and its formal verification capabilities. Readers are encouraged to consult the latest version for the most current theoretical specifications.

Contents

Preface	1
Abstract	2
1. Introduction	4
1.1. Background	4
1.2. Motivation	4
1.3. Pedagogical Scope and Intended Audience	4
1.4. Overview of the Treatise	5
2. Untyped Lambda Calculus (UTLC)	5
2.1. The Minimal Structure of Computation	6
2.2. Syntax and Formal Notation	6
2.3. Substitution and α -Equivalence	7
2.4. Evaluation and β -Reduction	8
3. Simply Typed Lambda Calculus (STLC)	8
4. The $\lambda\Pi$ Calculus and Dependency	8
5. The Curry-Howard Correspondence	8
6. Martin-Löf Type Theory (MLTT)	8
7. Universe Hierarchies	8
8. Implementation of the Stella Kernel	8

1. Introduction

The formal verification of software and mathematical theorems has evolved from a niche academic pursuit into a critical industrial necessity. However, a significant “conceptual barrier” prevents mainstream developers from achieving true proficiency in formal methods. This treatise introduces Stella, a high-performance Proof Assistant (PA) kernel implemented in Moonbit, designed to bridge this gap through a transparent, pedagogical architecture.

1.1. Background

Modern proof assistants such as Lean 4 and Coq often operate as opaque systems that prioritize high-level automation over architectural transparency. These systems utilize sophisticated “tactics” which allow users to complete proofs without understanding the underlying type-theoretic mechanisms. This reliance on automation creates a pedagogical challenge where users may learn the syntax of a tool without attaining formal fluency. Formal fluency is the ability to think natively in constructive logic and to understand the elaboration process where raw code is transformed into core terms. When automation fails, the lack of a deep mental model regarding the kernel’s internal behavior becomes a prohibitive bottleneck that extends the learning cycle to several months.

1.2. Motivation

To address the pedagogical challenges inherent in opaque systems, the design of Stella is informed by a concrete observation: the primary obstacle to learning dependent type theory is not syntactic complexity, but the lack of a transparent mental model of the underlying calculus and elaboration process. This journey followed a structured progression:

- Mastering the Untyped Lambda Calculus (UTLC) to understand the mechanics of recursion and substitution as the raw material of computation.
- Utilizing the Simply Typed Lambda Calculus (STLC) to learn how types function as static constraints on computational flow.
- Discovering that Π -type and Σ -type, which expanded the expressive power of the language to include dependent relationships, yet realizing that these alone were insufficient to express formal propositions.
- Understanding the Identity Type (*Id*-type) and the *J*-eliminator, which served as the true turning point. This allowed equality to be treated not as a hidden compiler check, but as a navigable computational path, finally enabling the system to function as a genuine logic for mathematical proof.
- Understanding the *W*-type to provide a general foundation for inductive types and well-founded data structures.

These stages reflect a minimal conceptual dependency graph for dependent type theory: each layer becomes intelligible only once the previous one is made explicit. Stella internalizes this dependency structure by exposing core calculi, eliminators, and elaboration rules directly, rather than concealing them behind tactic-level automation. By aligning the structure of the kernel with this progression, Stella aims to make formal reasoning approachable without sacrificing theoretical rigor.

1.3. Pedagogical Scope and Intended Audience

While Stella is designed as a rigorous proof assistant kernel, this treatise intentionally prioritizes conceptual transparency over surface-level automation. The intended audience includes readers without prior exposure to dependent type theory or formal proof assistants.

Rather than presenting dependent type theory as a collection of axioms or tactics, the exposition follows the structure of the kernel itself. Core notions such as typing judgments, elaboration, identity,

and induction are introduced as explicit computational constructions, allowing readers to observe how proofs arise from evaluation and normalization.

This approach is motivated by the belief that true fluency in formal methods emerges not from mastering tooling, but from understanding how programs, types, and proofs are unified at the level of the core calculus.

1.4. Overview of the Treatise

Readers already familiar with simply typed lambda calculus may choose to proceed directly to Chapter 4.

Readers already familiar with dependent type theory and the internal type systems of proof assistants such as Coq, Agda, or Lean may choose to proceed directly to Chapter 8.

To provide a systematic reconstruction of dependent type theory, this work is organized into chapters that mirror the conceptual dependencies of the Stella kernel:

- Chapter 2: Untyped Lambda Calculus (UTLC). We begin with the pure mechanics of computation, introducing terms, variables, and the process of β -reduction. This chapter establishes the foundation of substitution and recursion where computation exists without the constraints of a type system.
- Chapter 3: Simply Typed Lambda Calculus (STLC). We introduce the notion of type discipline by assigning static labels to terms. This chapter explores how types function as a structural “sanity check” and introduces the fundamental distinction between terms and types.
- Chapter 4: The $\lambda\Pi$ Calculus and Dependency. We transition into the core of the hierarchy by introducing dependent functions (Π -types). In this chapter, we extend the calculus to allow types to depend on terms and generalize these constructions to include dependent pairs (Σ -types).
- Chapter 5: The Curry-Howard Correspondence. We examine the profound isomorphism between computer programs and mathematical proofs. This chapter demonstrates how the type constructors introduced previously map directly to logical connectives, transforming our programming language into a formal system for intuitionistic logic.
- Chapter 6: Martin-Löf Type Theory (MLTT) and the Logic of Identity. We introduce the full expressive power of MLTT by incorporating the Identity type (Id -type) and W -types. Here, we discuss the implementation of the J -eliminator and the construction of inductive data structures, establishing Stella as a complete and rigorous proof assistant.
- Chapter 7: Universe Hierarchies. We address the problem of self-referential types and the potential for logical paradoxes. This chapter introduces a cumulative hierarchy of universes ($Type_0, Type_1, \dots$), ensuring the consistency of the kernel while maintaining the flexibility of higher-order reasoning.
- Chapter 8: Implementation of the Stella Kernel. We shift from theoretical foundations to engineering reality. This chapter details the internal architecture of Stella, including the bidirectional type-checking algorithm, the normalization engine, and the specific optimizations enabled by the Moonbit infrastructure.

2. Untyped Lambda Calculus (UTLC)

Before we introduce the formal machinery of the lambda calculus, we must clarify what we mean when we speak of a “program”. In many contexts, a program is seen as a collection of source files, a binary to be executed, or a tool for managing state. However, to understand the foundations of proof assistants, we must adopt a more primitive and abstract view.

Here, a program is simply a formal description of a process. This process consists of taking an initial expression and transforming it into another expression by following a set of clear, mechanical rules. Imagine a series of mathematical simplifications:

$$(1 + 2) \times 3 \rightarrow 3 \times 3 \rightarrow 9$$

The procedure of “reducing” complex structures into simpler ones is what constitutes the program. This process of transformation is inherently similar to the concept of a function we encounter in basic mathematics, where an input is mapped to an output according to a specific rule.

When we view a program through this lens, we see that computation is not about changing the state of a physical machine, but about applying rules to expressions until no further rules can be applied. For such a system to be reliable, it must be deterministic, meaning the rules are so clear that any two agents following them would reach the same result without relying on intuition. Furthermore, it must be structural, ensuring that each step of computation depends entirely on the expression currently being evaluated rather than on any hidden external state.

In this chapter, we explore the untyped lambda calculus as the smallest formal system that captures this essence.

2.1. The Minimal Structure of Computation

If computation is defined as the act of transforming expressions by applying rules, we must identify the smallest set of tools required to describe any possible computation. The untyped lambda calculus provides a surprising answer by demonstrating that only three fundamental structures are necessary. In our intuitive reasoning, we compute by using names to refer to data, by creating templates for actions, and by filling those templates with specific information. The lambda calculus formalizes these actions into the components of variables, abstractions, and applications.

A variable serves as a simple reference or placeholder, allowing an expression to point to a value produced elsewhere. An abstraction, denoted by the lambda symbol, provides the mechanism for generalization. It transforms a specific expression into a reusable template by binding a variable, effectively saying that for any input, the following rule should be applied. Finally, an application is the operation that activates the computation by supplying a specific argument to a template.

By stripping away specialized data types such as numbers or booleans, this system forces us to recognize that all computational structures can emerge purely from the interaction of these three elements. We do not begin with numbers as primitives; instead, we begin with the ability to define and apply rules, discovering that data itself is merely a specific pattern of computational behavior.

2.2. Syntax and Formal Notation

To communicate these structural components with the precision required for formal verification, we must move from conceptual descriptions to a formal grammar. In research literature, the set of lambda terms, denoted by Λ , is defined compactly using **Backus-Naur Form (BNF)**. This notation allows us to specify the recursive structure of the language in a single, rigorous statement:

$$\begin{aligned} t, u ::= & x \quad (\text{variable}) \\ & | \lambda x. t \quad (\text{abstraction}) \\ & | t \ u \quad (\text{application}) \end{aligned}$$

This BNF notation will appear repeatedly throughout the remainder of this treatise. At this stage, it should be read simply as a compact way of describing the shape of expressions, rather than as something to be mastered in full detail.

As new concepts are introduced, the same notation will be reused and gradually extended. By encountering it in multiple contexts, the reader will become familiar with both its form and its meaning through use, rather than through upfront memorization.

For now, it suffices to recognize that this definition enumerates all possible lambda terms and fixes their basic structure. The notation itself will become more natural as it is applied in subsequent chapters.

The variable x constitutes the most primitive unit of the UTLC, acting as an atomic reference or placeholder. Intuitively, it can be thought of as a “slot” in a formula, ready to be filled with any term in the UTLC. It does not merely point to data in a conventional sense. It represents any term in the UTLC that may later be substituted or applied.

When computation moves from a specific instance to a general rule, we utilize the abstraction $\lambda x.t$. This construction establishes a template by declaring a binder λx as the input and a term t as the computation using that input. The dot separates the declaration of the input from the body of the term, clarifying the scope of the bound variable.

Finally, the application tu represents the dynamic phase of computation, where a function term t is applied to an argument term u . For instance, applying the number n to the function $f(x)$ yields $f(n)$. Note that if t is not an abstraction, the application does not reduce, this situation will be discussed in Chapter 2.4.

2.3. Substitution and α -Equivalence

With the basic syntax of the untyped lambda calculus established, we can now address the mechanism that gives computation its dynamic character: substitution. If abstraction provides a template and application supplies an argument, substitution explains how the argument is integrated into the template.

Formally, we write $t[x := u]$ to denote the term obtained by replacing all free occurrences of the variable x in t with the term u .

To define substitution precisely, we must first clarify the role of binders. In the lambda calculus, an abstraction $\lambda x.t$ introduces a scope in which the name x is locally bound. Occurrences of x governed by this binder are bound; any occurrence of a variable not governed by a corresponding binder is free and refers to the surrounding context rather than to a local parameter.

We formalize this distinction using the set of free variables, denoted $FV(t)$, which is defined inductively as follows:

$$\begin{aligned} FV(x) &= \{x\} \\ FV(\lambda x.t) &= FV(t) \setminus \{x\} \\ FV(t \ u) &= FV(t) \cup FV(u) \end{aligned}$$

Intuitively, $FV(t)$ should be read as “the collection of free variables of the term t .” The notation $\{x\}$ denotes a collection containing the single variable x , $FV(t) \setminus \{x\}$ means removing x from the collection of free variables of the term t , and $FV(t) \cup FV(u)$ means combining the collections of free variables of t and u .

This distinction is the safeguard of substitution. We only substitute for free occurrences. If substitution proceeds into an abstraction $\lambda y.t$ where y is the same as our target variable x , the substitution stops, because any x inside that abstraction is now “shadowed” by a new, local x . Even with this precaution, a more subtle danger exists: Variable Capture. This occurs when we substitute a term u into t , and a free variable within u accidentally becomes trapped by a binder inside t . For instance, if

we substitute y for x in the term $\lambda y.x$, a naive replacement would yield $\lambda y.y$. We have inadvertently changed a function that returns its input's neighbor into an identity function. This error violates the principle of determinism and structural clarity.

To resolve this, we introduce α -equivalence ($\equiv_\alpha$). Here, the symbol $\equiv$ denotes equivalence, meaning that two terms are considered identical under a specified criterion. The subscript α simply indicates that this equivalence is judged according to the rules governing renaming of bound variables. Different subscripts on $\equiv$ will be used to indicate different notions of equivalence, each defined by its own governing principles.

This principle states that the names of bound variables are irrelevant, only the binding structure matters. The terms $\lambda x.x$ and $\lambda y.y$ are computationally identical.

$$\lambda x.t \equiv_\alpha \lambda y.t[x := y] \quad (\text{where } y \notin FV(t))$$

By treating α -equivalent terms as interchangeable, we can always rename a binder to a “fresh” name before performing substitution, thereby avoiding capture. Mastering this conceptual nuance is the first true step toward formal fluency. It ensures that computation depends only on the explicit structure of expressions, not on the arbitrary choice of names.

2.4. Evaluation and β -Reduction

While substitution defines how to plug an argument into a template, it does not, by itself, represent the execution of a program. Writing $f(n)$ is a statement of intent. It indicates that the function f is to be applied to the parameter n . Evaluation, however, is the dynamic process of actually calculating the result that $f(n)$ denotes. In the UTLC, this process of transformation is formalized as β -reduction. (notated as $\rightarrow$)

3. Simply Typed Lambda Calculus (STLC)

4. The $\lambda\Pi$ Calculus and Dependency

5. The Curry-Howard Correspondence

6. Martin-Löf Type Theory (MLTT)

7. Universe Hierarchies

8. Implementation of the Stella Kernel