Formalization and Mechanization of Gödel's System T

MIKEY BAKER, MELISA CIVELEKOGLU, SYLVIA WAN, and XINYI YU

1 INTRODUCTION

Gödel's System T is an extension of the Natural Deduction Calculus (ND) which introduces the natural numbers as a connective. System T also introduces a recursion scheme called Primitive Recursion over the natural numbers. Gödel's work took place in the mid 20th century when mathematicians were attempting to fit mathematics into a formal logical framework, one example of which is Dedekind-Peano arithmetic [1]. System T integrates the axioms laid out by Dedekind & Peano with ND. Although System T was first introduced in 1958[2], the logic has maintained relevance. For example, it was recently mechanized in Agda[3], and is often used as a jumping-off point for modern works (as seen in [4]). System T has many properties of note; studied here are the Weak Normalization, as well as the ability to encode typical arithmetic operations via primitive recursion. Weak normalization says that every proof either steps, or reaches a normal form[5]. One can also prove strong normalization, which says that every proof reaches a normal form[4], but that was not feasible given the time constraints of this project. The relevance of these properties are further explored in the formalization section. Finally, it seemed natural to include a mechanization of System T in Beluga; proof mechanization and the Curry-Howard correspondence have been central themes of this course, and the mechanization of System T in Agda motivates a similar mechanization in Beluga.

2 FORMALIZATION

To formalize the system, we begin by defining the set of types τ, the base type **nat** and the function type $\tau_1 \to \tau_2$. An expression **e** is then given by: a variable x, the constant zero **z**, the successor **succ(e)**, the primitive recursion operator **rec**(e;e_0;x.y.e_1), the λ-calculus constructs λ**(x:**τ**).e**, and the application $\mathbf{e_0}(\mathbf{e_1})$. The syntax setting is inspired by Harper [6]. Typing judgments are given in a context Γ by standard rules: a variable has the type declared in Γ. Zero has type **nat**, and if **e** has type **nat** then **succ(e)** has type **nat**. Recursion is well-typed when the initial value $\mathbf{e_0}$ and the step body $\mathbf{e_1}$ are appropriately typed. Abstraction and application follow the usual λ-calculus rules ensuring functions map arguments of the correct type to results of the correct type. Reduction rules specify operational semantics: β-reduction for application, base and step rules for recursion over zero and successors, and closure under evaluation context. Together, this provides the framework

Authors' address: Mikey Baker; Melisa Civelekoglu; Sylvia Wan; Xinyi Yu.

needed for the proof of properties such as normalization. To clarify how the syntax works in practice, we present a few examples of natural number operations encoded in this system: successor, predecessor, addition, subtraction, multiplication, and exponentiation.

The proof of weak normalization [7] is carried out in Tait's style [8]. First, we introduce the normal form, the reducibility predicate, which guarantees weak normalization (Definition 2-3). Then via Lemmas 1-4 we prove that $\mathbf{R}_\tau(\mathbf{e})$ is preserved both forward and backward. The major argument is the Fundamental Lemma (Lemma 5), which is proven by induction on typing derivations and implies weak normalization (Theorem 2).

3 MECHANIZATION

To mechanize System T in Beluga and integrate it into ND, **nat** is introduced as a proposition. To maintain syntactic similarity between the formalization and mechanization, **zero** and **succ** are thought of as the proof terms of two introduction rules of **nat**. Primitive recursion is the proof term for an elimination rule for **nat**. ND was recast in its programming formalism by rewriting introduction and elimination rules in the proof-term manner. Our mechanization is intrinsically typed, meaning that the type of each term is encoded within the term itself, which allows our programs to only be constructed if they are already well-typed. We chose intrinsic typing in order to ensure by construction that the base case of primitive recursion and the inductive case returned the same type (i.e. preserve type correctness). For consistency, we proceeded to intrinsically type the rest of the system. This approach simplifies the user's reasoning about type safety and fits with Beluga's dependent typing, although it is somewhat restrictive: namely, ill-typed terms cannot be modeled in this mechanization. Given that our objective was to mechanize well-typed proofs such as arithmetic operations, we found this tradeoff acceptable for our purposes. The mechanization could be re-written with extrinsic typing which would be equally valid. All operations encoded in the formalization section are mechanized as example proofs in Beluga without any use of Beluga's recursion scheme.

Although we did not originally plan to mechanize the proof of Weak Normalization, we decided to include it as an extension to our project. Our mechanization includes inductive definitions of normal forms (**Normal**), evaluation (**Eval**) and weak normalization (**WN**) We introduced a helper term constructor, **step**, to model the step case of primitive recursion within the evaluation relation. Intrinsic typing was critical to simplifying this meta-theoretic proof as we did not need to separately prove type preservation.

REFERENCES

[1] nLab authors. Peano arithmetic. https://ncatlab.org/nlab/show/Peano+arithmetic, April 2025. Revision 17.

[2] Von Kurt Gödel. Über eine bisher noch nicht benÜtzte erweiterung des finiten standpunktes. *Dialectica*, 12:280–287, 12 1958.

[3] Ana Bove and Peter Dybjer. *Dependent Types at Work*, pages 57–99. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[4] Ulrich Berger. Continuous semantics for strong normalization. In S. Barry Cooper, Benedikt Löwe, and Leen Torenvliet, editors, *New Computational Paradigms*, pages 23–34, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[5] The Stacks project authors. The stacks project. https://stacks.math.columbia.edu, 2025.

[6] Robert Harper. *Practical Foundations for programming languages Robert Harper.* Cambridge University Press, 2016.

[7] The Stacks project authors. The stacks project. https://stacks.math.columbia.edu, 2025.

[8] W. W. Tait. Intensional interpretations of functionals of finite type i. *Journal of Symbolic Logic*, 32(2):198–212, 1967.