Type Theories of Natural Numbers

A Study of Conservative Extension

Aleksandar Babunović

Supervisors: Robin Adams & Rasmus Blanck

A Thesis for Master's Degree in Logic, 30 credit points

University of Gothenburg

Faculty of Humanities

Department of Philosophy, Linguistics and Theory of Science

Abstract

We present some possible definitions for what it means for a type theory to be a conservative extension over another type theory. We do so by giving two type theories of natural numbers: $TT(\mathbb{N})$ and $TT(\mathbb{N}, \times)$. $TT(\mathbb{N})$ is a type theory that comprises only the natural numbers as a type, while $TT(\mathbb{N}, \times)$ is an extended theory of $TT(\mathbb{N})$ that includes an additional type constructor that can form pair types. Our aim is to search for a general definition of a conservative extension. Our approach does not only involve judgements in our definition of conservativity, but also convertibility between terms, which corresponds to the process of computation for each type theory. Understanding type theories and their computation have valuable implications in research areas, such as in Computer Science, particular in the context of proof assistants and other fields where type theory is considered.

We prove that $TT(\mathbb{N}, \times)$ is not a conservative extension over $TT(\mathbb{N})$ in the strongest sense, by giving a counter example, but that it is a conservative extension in three other senses. We do this by defining a translation from $TT(\mathbb{N}, \times)$ to $TT(\mathbb{N})$, we define terms in $TT(\mathbb{N})$ that will represent terms from $TT(\mathbb{N}, \times)$, and therefore use the translation to prove conservativity.

Our contribution lies in providing a deeper understanding of the relationships between type theories, specifically focusing on type theories that are fragments of Martin-Löf type theory, where Martin-Löf type theory serves as an alternative foundation of mathematics. We argue why one of the notions of conservative extension, proposed in this thesis, is the most suitable definition to establish a conservative extension between $TT(\mathbb{N})$ and $TT(\mathbb{N}, \times)$. From this notion of conservative extension, we discover that since they are conservative, they denote the same functions, specifically the primitive recursive functions. This motivates us to utilize this definition of conservative extension, because it provides the insight that if two theories are conservative, they denote the same functions. If this is the case in general between type theories then this deserves further research on our notions of conservativity, as it may be helpful for transferring theorems. This could serve as a new feature for proof assistants that are based on a type theory, with a particular focus on conservativity.

These findings prompt further questions: can these definitions of conservative extensions be regarded as a general definition within the context of type theory? Can these definitions be valuable in applications within type theory? Are there any significant implications that arise from conservative extensions in type theories overall? Having an understanding of conservative extensions and their implications has the potential to enhance our understanding of type theory.

Acknowledgements

First I would like to thank Robin for all his supervising and help throughout this thesis. I admire his knowledge when it comes to logic and type theory, and without his help and support I would not have been able to achieve the results of this thesis. I want to express my gratitude to Rasmus, who has been supervising me throughout this thesis as well. I want to especially thank him for teaching me the techniques for writing a good thesis, his feedback and support on the thesis. His supervising is inspiring and the thesis would not be the same if it was not for him.

I want to thank my family, for their support through my whole life. I want to thank my cat, Lizzy, for being around while I was working on the thesis. I want to express my gratitude to all the teachers in the master's programme, thank you, Fredrik, Martin, Graham, Bahareh and Rasmus for helping me through all the courses, you have taught me a lot about logic. Thank you to all the students in the logic group, including the master students and PhD students, this has been a journey for all of us and I can not imagine a greater team to share this experience with. I want to especially thank Camila and Yoann for being there for me when I needed their support. I really enjoy your friendship, and I am glad for all the moments I have shared with you.

Finally, I want to express my heartfelt gratitude for myself and the commitment towards this thesis. I want to thank myself for putting the huge amount of hours on proving theorems, and investigations on the research question. I appreciate all the effort and contribution I made on type theory, and I know that this thesis will be helpful for at least one person on this planet, namely me, since I need this for my degree. I am the whole engine for putting this thesis in front of you. Remember this:

Without me, you would not be reading this sentence.

Contents

1 Background

1.1 Introduction and Preliminaries

1.1.1 Introduction

Type theories are formal systems designed for reasoning about types and terms, which correspond to propositions and programs, which makes them significantly constructive and suitable for practical applications, such as programming languages. Some type theories provide an alternative foundation of mathematics, such as Martin-Löf type theory [14] developed by Per Martin-Löf. These type theories offer principles that allow us to construct statements and proofs, and provide a perspective for reasoning about mathematical structures, meaning that we are able to express complicated structures such as groups, rings, fields, etc. In this thesis, our focus will be on exploring conservative extensions between type theories, specifically some fragments of Martin-Löf type theory. Through this investigation, we aim to enhance our understanding of the foundation of mathematics and the applications of type theory.

In type theory, we interpret types as propositions, and terms are interpreted as proofs, meaning that a term of a specific type represents a proof for the corresponding proposition. One distinctive feature of type theory is that it provides not only information about propositions, but also the proofs of these propositions. Proofs can be seen as programs, for example, we can consider a term $\lambda x^A.x$ that denotes the identity function of type $A \to A$. This term serves as both a proof of $A \to A$ and as a program. By using β-reduction in Lambda calculus [6], we can evaluate the program $\lambda x^A.x$ by applying it to an input value a, to get the output a. This evaluation process is the computation part, and it is similar to how we use programs to obtain the desired outputs. This gives us a more constructive reasoning through type theory, which also grants us applications in various fields, including programming languages, particularly functional programming languages [23]. Moreover, type theory plays a role in proof assistants. Proof assistants assist users in creating formal proofs, and with type theory, it ensures the correctness of these proofs. Examples of proof assistants where type theory is used are Coq [5] and Agda [16]. Conservative extension between type theories could enable us to transfer theorems from one type theory to another in proof assistants. This allows us to extend a theory without altering the true statements that were valid in the original theory. This grants us practical contributions due to its connection to proof assistants.

Despite the wide range of applications of type theory, there has been relatively limited exploration concerning the relationships between different type theories, particularly in terms of determining when one type theory is a conservative extension over another. Consider two theories, one theory that extends the other. Then the extended theory is a conservative extension, if for every sentence of the language of the original theory that is provable in the extended theory, is also provable in the original theory. This tells us that if you extend a theory, then it should not change the behavior from the original theory. This concept deserves further investigation, as it holds the potential for a better understanding of the relationships between different type theories and can have valuable implications for the foundation of mathematics.

In this thesis, we explore the relationship between two type theories of natural numbers, where one of the type theories serves as an extension of the original theory.

- The first one, denoted as $TT(\mathbb{N})$, consists of the natural numbers as the only type. This means that all the terms we can construct in $TT(\mathbb{N})$ will only be of type $\mathbb{N}$, and nothing else.
- The second theory, denoted as $TT(\mathbb{N}, \times)$, is an extension of $TT(\mathbb{N})$ and includes an additional type constructor, $\times$, that can form pair types.

Some types that we can construct in the extended theory $TT(\mathbb{N}, \times)$, are all different combinations of pairs $A \times B$, for instance, $\mathbb{N} \times \mathbb{N}$, $\mathbb{N} \times (\mathbb{N} \times \mathbb{N})$, $(\mathbb{N} \times \mathbb{N}) \times (\mathbb{N} \times \mathbb{N})$, etc. In $TT(\mathbb{N}, \times)$, in addition to introducing a new type constructor, new terms corresponding to the pairing type are also introduced. These new terms include an ordered pair $\langle a, b\rangle$ and two projections (π_1, π_2). The projection $\pi_1(\langle a, b\rangle)$ retrieves a, while $\pi_2(\langle a, b\rangle)$ retrieves b.

We define four definitions of conservative extension, where two of them are proposed from us. Then we translate terms that originates from $TT(\mathbb{N}, \times)$ to $TT(\mathbb{N})$, in order to prove that they are conservative from

some of our notions of conservativity. One of the notions that we define, which is strong conservativity, has already been proposed in [25, Definition 7]. Because strong conservativity has been studied before, then this motivates us to investigate the relationship between $TT(\mathbb{N}, \times)$ and $TT(\mathbb{N})$ to that particular notion of conservative extension, due to prior research of the notion. When we are studying conservative extension between type theories, we are interested in how terms are computed, and for computation in type theories, we do reductions. Reduction is the process of computing terms according to specific reduction rules. For example, when we represent terms as factors $(a \cdot b)$ and define multiplication, we can compute the result (r) to determine the product of the factors $(a \cdot b = r)$. Most of the proposed definitions of a conservative extension involves the use of reductions, but there is one particular definition, called canonical conservativity, that uses a special form of terms. Canonical conservativity involves the use of canonical terms, which are terms without any variables and can not be further reduced. In $TT(\mathbb{N})$, the canonical terms are the numerals, for instance, the corresponding terms that represents the natural numbers $0, 1, 2, 3, 4, 5, ...$, but the term that represents the product $1 \cdot 2$ is not a canonical term. By proving that $TT(\mathbb{N}, \times)$ is a canonically conservative extension over $TT(\mathbb{N})$, we need to prove properties about canonical terms in both $TT(\mathbb{N})$ and $TT(\mathbb{N}, \times)$. However, an essential question that arises: can the definition of a canonically conservative extension be considered a general definition of a conservative extension? The pairing terms in $TT(\mathbb{N}, \times)$, that is ordered pairs, can be encoded in $TT(\mathbb{N})$. Therefore, we expect that $TT(\mathbb{N}, \times)$ is a conservative extension over $TT(\mathbb{N})$, in the strongest notion of conservativity that we define. However, they are not strongly conservative due to the presence of free variables, which we will illustrate with a counterexample. Nonetheless, they should be conservative at some weaker notion. What is important is that conservativity ensures that for each term in both type theories denote the same functions, which is valuable for theorem proving, particularly in proof assistants.

To establish a conservative extension between $TT(\mathbb{N}, \times)$ and $TT(\mathbb{N})$, we need to ensure the preservation of typing judgements from $TT(\mathbb{N}, \times)$ to $TT(\mathbb{N})$. A typing judgement asserts that a given term is of a specific type. Furthermore, we need to ensure that if we have a term of the type $\mathbb{N}$ in the extended theory, then there exists a corresponding term in the original theory with the same type, and the terms are convertible. Convertibility is the equivalence relation generated by applying multiple reduction steps, which correspond to the process of computing terms. To prove that $TT(\mathbb{N}, \times)$ is a conservative extension over $TT(\mathbb{N})$, we define a translation from $TT(\mathbb{N}, \times)$ to $TT(\mathbb{N})$, by defining terms in $TT(\mathbb{N})$ to represent terms originating from $TT(\mathbb{N}, \times)$. In the end, we demonstrate the implications of the canonically conservative relation between $TT(\mathbb{N})$ and $TT(\mathbb{N}, \times)$ on primitive recursive functions. Primitive recursive functions are obtained from basic functions through composition and recursion.

If a type theory is strongly conservative over another, then it gives us canonical conservativity, which in turn implies two other weaker notions of conservativity, between the type theories. The two of the weakest notions of conservativity are closed conservativity and propositional conservativity. Closed conservativity is a modification of strong conservativity, but we do not consider any free variables. It is not clear if closed conservativity implies propositional conservativity, therefore, we cannot determine which one is weaker. While in propositional conservativity we are only considering judgements, but nothing about reductions between terms. Can any of these definitions be considered a general definition of a conservative extension? What does it mean to have a conservative extension between type theories? What does conservative extension tell us about the relationships between type theories? By exploring these aspects, we may get valuable applications of type theory, paving a way for a greater understanding of the relationships between type theories and their implications, and further developments in the foundation of mathematics.

1.1.2 Motivation

Our motivation is to explore what could be a general definition of a conservative extension in the context of type theory. We argue that the standard approach of conservative extension, which is used in first-order logic, may not be suitable for establishing the computational aspect of type theory. This means that, in type theory, we allow for computations, while in the standard notion of conservativity, computations are not considered.

By having an understanding of the computation in type theories, we gain a deeper insight into their

behavior, and ensure that the computations also remains unchanged when extending a theory compared to its original. A stronger notion of conservativity will grant us more than what types are inhabited between the theories, as it will also grant us insights about the reductions between terms.

There have been previous explorations of conservative extensions between type theories, and we highlight one particular approach that shares similarities with our exploration. Additionally, there is another notable approach that uses higher-order categorical logic to formulate a definition of a conservative extension.

In [12, p. 197], a definition is provided for when a translation is a conservative extension, by using category theory. Category theory utilizes objects and morphisms to establish relationships between objects within a category, which allows the construction of mathematical structures. To address conservative extension, they define a translation as a morphism in a category called Lang, of type theories, meaning that type theories in this category are treated as objects, while the morphisms between these objects are translations. Then they define conservative extension as: for every closed formula, if the translation of the formula is provable in the extended theory, then the formula itself is provable in the original theory.

In this thesis, our motivation is to provide a definition of conservativity without relying on category theory and mentioning a translation in our notions of conservativity. There are several ways of providing a translation between two type theories, rather than one single method. In our notions of conservativity, we only say that there exist a term in the original theory that is convertible to a term in the extended theory. We will use a translation though for proving conservativity, but there may be multiple ways of defining a translation between two type theories.

In [25], Xue presents a definition of conservative extension that is similar to the formulation of conservativity for first-order logic. However, he also introduces another definition of what he calls a definitional extension [25, Definition 7]. Definitional extension is similar to a strong notion of conservative extension, that we present in this thesis, since he utilizes derivations in the extended definition. In the conclusion of his paper, Xue mentions that conservativity alone is not enough to capture the relation between the system under study for his paper. This motivates us on proposing several stronger definitions of conservative extension, and explore their applicability in the relationship between the two type theories studied in this thesis.

There has been other explorations of conservative extensions in the context of type theory, that gives more motivations on why conservative extension is important in type theory. In [1, p. 18] they prove that Peano arithmetic is conservative over Heyting arithmetic for some particular sentences, and they use Gödel's System T to establish these results. In that study, they establish that every provable total recursive function in Peano arithmetic and Heyting arithmetic can be denoted by a term in System T. By exploring the properties of System T and its relation to arithmetic, they demonstrate several conservative results in arithmetic. To study conservative extension in type theory is therefore a motivation by its potential for the implications on arithmetic.

One major motivation of this investigation is to enhance the design of proof assistants and encourage further exploration in that field. With a general definition of a conservative extension, we could develop a new proof assistant or update existing ones to incorporate these notions of conservativity. This would help us to construct type theories within the proof assistant, establish translations between them, fulfilling the requirements for conservativity and therefore enable to transfer theorems between type theories. For instance, both $TT(\mathbb{N})$ and $TT(\mathbb{N}, \times)$, the type theories that are investigated in this thesis, denote the same functions, namely the primitive recursive functions. Therefore, if a general definition of conservativity provides such insights, it would benefit the design of a proof assistant aimed for transferring theorems.

1.1.3 Outline

This thesis is highly technical, and thus we provide an overview of its structure:

- **Section 1.2**: We introduce type theory and the type theories that we will use for this thesis, namely $TT(\mathbb{N})$ and $TT(\mathbb{N}, \times)$.
- **Section 1.3**: We define four definitions, where two of them are proposed from us, of what it means for a type theory to be a conservative extension over another.

- **Section 2**: We define different encodings in $TT(\mathbb{N})$ that will represent terms that originates from $TT(\mathbb{N}, \times)$.
- **Section 3**: We prove Church-Rosser property for $TT(\mathbb{N})$ and $TT(\mathbb{N}, \times)$, that will be useful in Section 4.
- **Section 4**: From our encodings and properties of $TT(\mathbb{N})$ and $TT(\mathbb{N}, \times)$, we do a translation in order to present the conservative relationship between $TT(\mathbb{N})$ and $TT(\mathbb{N}, \times)$.
- **Section 5**: We conclude our results and propose some further work.

1.2 Type Theory

Type theory is the main focus and hence it is important that there is a great understanding of the subject before investigating any relations between type theories. Two influential type theories that has been studied in recent years are: the Simply-typed λ-calculus [6] and Martin-Löf type theory [14].

The simply-typed λ-calculus is a type theory that focuses on functional types. It serves as a basis for many functional programming languages and has been instrumental in understanding the principles of computation and programming. Even in imperative programming languages, one could use λ-expressions which comes from λ-calculus to represent anonymous functions in the programming language.

On the other hand, Martin-Löf type theory stands out by incorporating powerful features like dependent types and constructive logic. It serves as an alternative foundation of mathematics, meaning that we have enough rules in the formal system to construct mathematical structures. These additions enhance its ability to formalize mathematics and has strong connections to proof theory. Our primary focus will be on studying two specific fragments of Martin-Löf type theory. These fragments offer valuable insights into conservative extensions, thanks to the inherent constructiveness of type theory.

Through our investigation of the conservative relation between the two type theories within Martin-Löf type theory, we aim to gain a broader understanding on how such relations should be established across different type theories. For example, will the results of conservative relations be useful when exploring similar relationships in the simply-typed λ-calculus? Moreover, the constructiveness of Martin-Löf type theory offers valuable implications for the foundation of mathematics.

Some other type theories are System F [8, Chapter 11], System F_C [22] and System T [8, Chapter 7]. Many functional programming languages incorporates the concepts from type theory, for instance, System F_C is used in the compiler for Haskell. Hence, not is type theory only important in logic and mathematics, it is also an important field in Computer Science.

1.2.1 Martin-Löf Type Theory

Martin-Löf Type Theory (MLTT), also known as Intuitionistic Type Theory, serves as an alternative foundation for mathematics. Developed by Per Martin-Löf, MLTT has become a cornerstone in multiple areas of research, this including computer science, formal verification, and mathematics. Notably, it has found practical utility in formal verification, enabling the verification of program correctness [15]. By utilizing MLTT, we can ensure that programs operate without encountering runtime errors. This motivates us to delve deeper into MLTT and explore conservative relations between its fragments. We will investigate two type theories of natural numbers, which are small fragments of MLTT. Hence, in the upcoming two sections, these two type theories are also expressible in MLTT. MLTT remains an interesting subject for research due to its constructiveness and wide-ranging applications. Gaining an understanding of conservative relations between MLTT fragments can provide valuable insights applicable to various type theories, including the simply-typed λ-calculus, and their respective applications.

Per Martin-Löf presented an idea in philosophy of logic with reasoning through judgements [14]. A judgement is the object of an act of knowledge, meaning that it is the thing we claim holds. In proof theory [24], we have $\Gamma \vdash B$, where $\vdash$ is an entailment, the conclusion B is entailed by the premises of Γ. We use the same notation $\vdash$ for judgements in type theory, which denotes the derivation of a term having a specific type. It is important to note that these judgements in type theory differ from entailments in

proof theory. In type theory, we assert that a term is of a specific type from a given context, rather than representing logical entailment or implication.

In Martin-Löf type theory, a judgement can have one of the following forms:

- A type, means A is a type.
- $A = B$, means A and B are the same types.
- $a : A$, means a is a term of type A.
- $a = b : A$, means a and b are the same terms of the same type A.

In the above definition, we have the lower-case letters a and b, which are terms. A term is a proof, while a type is a proposition. That is why we say that we interpret propositions as types. In proof theory, $\Gamma \vdash P$ expresses the existence of a proof of P. Whereas in type theory, we write $\Gamma \vdash t : P$, and t here is a proof of P, where in Γ we store all the hypotheses, $x_1 : A_1, ..., x_n : A_n$, that are required in order to have a proof of t. Below we show the difference between type theory and what it means in proof theory:

- A is a type.
- a is a term of type A.
- A is inhabited.
- If x_1 is a term of type A_1,...,x_n is a term of type A_n then b is a term of type B.

- A is a proposition.
- a is a proof of proposition A.
- A is provable.
- If x_1 is a proof of A_1,...,x_n is a proof of A_n then b is a proof of B.

If we would have a proof a of A, then we could also conclude that A is true. Therefore, we can derive the following:

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash A\ true}$$

Let's have a look at one of the introduction rules in natural deduction [24] and see how the corresponding judgement looks like. The rule below is a conjunction introduction rule:

$$\frac{A \qquad B}{A \wedge B} \wedge I$$

In Martin-Löf Type Theory we treat the types as propositions. To introduce the conjunction introduction rule, we use a pairing rule:

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\Gamma \vdash \langle a, b \rangle : A \times B}\ Pair$$

From the derivation of the judgements, we also have proofs, which are a and b of the types A and B respectively. In the $\wedge I$-rule, we only know the existence of such a proof. Similarly, we have the corresponding conjunction elimination rules in type theory: projection one and projection two (π_1 and π_2, respectively). These rules allow us to extract the left or right element of a pair, respectively. Below are the rules for conjunction elimination in natural deduction:

$$\frac{A \wedge B}{A} \wedge E_R \qquad\qquad \frac{A \wedge B}{B} \wedge E_L$$

And below are the corresponding judgement rules, which are the projection rules, that corresponds the conjunction elimination rules:

$$\frac{\Gamma \vdash p : A \times B}{\Gamma \vdash \pi_1(p) : A}\,\pi_1 \qquad\qquad \frac{\Gamma \vdash p : A \times B}{\Gamma \vdash \pi_2(p) : B}\,\pi_2$$

We will revisit the pairing and projection rules, as we explore their translation into a type theory with a reduced set of rules that can encode these rules.

1.2.2 Type Theory of Natural Numbers

A fragment of Intuitionistic Type Theory, which we shall explore, is the type theory of natural numbers, denoted $TT(\mathbb{N})$. We will explore $TT(\mathbb{N})$ and its relationship to its extension $TT(\mathbb{N}, \times)$, that we shall define after $TT(\mathbb{N})$. Our goal is to explore the conservative relations between these two theories. And thus, we want to keep the theories as simple as possible, both the extension and the original. The type theory of natural numbers uses the natural numbers as its only type, and nothing else. The terms, that will represent natural numbers, will be transformed through recursion, and we do this by defining reduction rules for recursion.

When defining a function by recursion $f(n)$, there is one base case $f(0)$, which is 0 in $TT(\mathbb{N})$. An inductive step $f(n)$, where the value at any natural number that is not equal to zero is defined recursively by calling the callee function again $f(n-1)$ with the argument that is one step less than the current value. For example, one could use the recursive definition of addition as depicted in Figure 1. We define addition by using the successor function. The successor function s takes a natural number and returns its successor, i.e., the next natural number in the sequence. For example, $s(0) = 1$, $s(1) = 2$, $s(2) = 3$, and so on. The result of $a + n$

- $Add(a, 0) := a$
- $Add(a, Succ(n)) := Succ(Add(a, n))$

Figure 1: Recursive definition of Addition

would be $\overbrace{(Succ...(Succ}^{n\ times}(a)...))$, where we call the function *add*, n times. Depending on the value of n, we do a pattern matching on n, for whether we should call $Add(a, 0)$ or $Add(a, Succ(n))$. It will work similarly in $TT(\mathbb{N})$, as for addition in Figure 1 with recursive calls inside a recursive term. To perform the computation with the natural numbers, a successor term s and a recursion term R will be required. The recursion term R is the term that we are going to use to specify the instructions for the computation/recursion. It will be used to define functions on natural numbers recursively, where just as in Figure 1 we do pattern matching, which specifies what instructions to compute if the argument is either $s(n)$ or 0. The general form of R is $R(a, [x, y]b, n)$, which is a term with three sub-terms, a, b and c. The term $[x, y]b$ is different from the others, hence we give a formal definition.

Definition 1.1. *$[x, y]b$ is a sub-term of $R(a, [x, y]b, n)$, where x and y are terms and bound within the term b.*

We say that x and y are bound variables. For example, consider the term $R(z, [x, y]x, n)$. The bound variables here are x and y, but only x has occurred within the scope of $[x, y]x$. On the other hand, the variables like z and n are not bound within any term, making them free variables instead. Another example is if we assume that we have defined an addition term $a + b$, then if we have a sub-term $[x, y](s(x) + z)$, then z is also within the scope of $[x, y]$ but z is still a free variable, while x serves as a bound variable.

Definition 1.2. *The syntax for $TT(\mathbb{N})$ is as following:*

- *There is one type in $TT(\mathbb{N})$, denoted $\mathbb{N}$.*
- *The terms for $TT(\mathbb{N})$ are: $a, b, c := 0 \mid z \mid s(a) \mid R(a, [x, y]b, c)$.*

In the syntax for $TT(\mathbb{N})$, z is a variable, the intended meaning of $s(a)$ is $a+1$ and the variables x and y are bound within b in the term $R(a,[x,y]b,c)$. The bound variables x and y are substituted within b, where x is substituted with the predecessor of c while y is substituted with the result of applying R again to the predecessor of c. By using only s and R, we can define many functions on natural numbers, such as addition, multiplication and exponential. Let us define addition, for illustration:

Definition 1.3. $a+b := R(a,[x,y]s(y),b)$

In Figure 1, the term $[x,y]s(y)$ represents the expression $Succ(Add(a,b))$, the term a represents the expression a in the case of $Add(a,0)$, and b is the term on which we perform pattern matching. However, this is just an expression of addition, we need to define the rules that will enable the computation for addition. For that we need reduction rules for $TT(\mathbb{N})$, which is a single-step computation of a term, transforming one term into another.

Remark. Some important notations:

- We write $a \equiv b$ if a and b are equivalent.
- We write $a \rightsquigarrow b$ to indicate that b can be obtained from a by one-step of reduction.
- If b can be obtained from a by applying the reduction rule ($\rightsquigarrow$) zero or more times inside a, then we write $a \twoheadrightarrow b$.
- We write $a \simeq b$ to indicate that a and b are convertible, meaning that they are equivalent under the equivalence relation generated by $\twoheadrightarrow$. More formally, $a \simeq b$ if and only if there is a sequence of terms $M_1, ..., M_n$ such that:
$$a \twoheadrightarrow M_1 \twoheadleftarrow M_2 \twoheadrightarrow M_3 \twoheadleftarrow M_4 \twoheadrightarrow ... \twoheadrightarrow M_n \twoheadleftarrow b$$

Assuming that we have encoded addition, multiplication, exponential, etc., some examples of convertible expressions in TT($\mathbb{N}$) are:

- $\overline{3}+\overline{1}$ and $\overline{4}\cdot\overline{1}$,
- $\overline{2}-\overline{2}$ and 0,
- $\overline{2}^{\overline{2}}$ and $\overline{2}\cdot\overline{2}$.

In $TT(\mathbb{N})$ the term $R(a,[x,y]b,c)$ will be the term to reduce. For that, we need to specify what is going to happen for each reduction step. Because we are using substitution for a reduction step, we define what substitution means for a term.

Definition 1.4. *Substitution $M[N/x]$ is defined as follows:*

- $x[N/x] := x$.
- $y[N/x] := y$ *(if* $x \neq y$*).*
- $M[N/x] :=$ *the result of replacing all free occurrences of* x *in* M *with* N.

Definition 1.5. *The reduction rules for $TT(\mathbb{N})$ are the following:*

- $R(a,[x,y]b,0) \rightsquigarrow a$
- $R(a,[x,y]b,s(n)) \rightsquigarrow b[n/x, R(a,[x,y]b,n)/y]$

Remark. For a natural number n, let $\overline{n} := \overbrace{s(s(...(s}^{n\ times}(0))...))$. We say that $\overline{n}$ is a numeral. For example, if we want to express the natural number 3 as a numeral, we write $s(s(s(0)))$.

Here is an example of how to compute $\overline{2}+\overline{2}$:

$$\overline{2} + \overline{2} \equiv R(\overline{2}, [x, y]s(y), \overline{2}) \equiv R(\overline{2}, [x, y]s(y), s(\overline{1})) \rightsquigarrow$$

$$\text{s(y)}[\overline{1}/x, R(\overline{2}, [x, y]s(y), \overline{1})/y] \equiv s(R(\overline{2}, [x, y]s(y), \overline{1})) \equiv s(R(\overline{2}, [x, y]s(y), s(0))) \rightsquigarrow$$

$$\text{s(s(y))}[0/\text{x, R}(\overline{2}, [x, y]s(y), 0)/y] \equiv s(s(R(\overline{2}, [x, y]s(y), 0))) \rightsquigarrow$$

$$\text{s(s}(\overline{2})) \equiv \overline{4}$$

Reduction enables computation. In lambda calculus, reductions have an important role, since the role of reductions is to express a computation for a given function. However, in general, reductions do not guarantee that we will reach our desired result. For example, in the lambda calculus, it is possible to reduce a term in different ways, by a one-step reduction. For instance, consider the lambda term: $(\lambda x.((\lambda y.x)M))N$. By using beta reduction, we can reduce this term to either $(\lambda y.N)M$ or $(\lambda x.x)N$, but both terms can find a common reduct N [20, Section 2.4].

In $TT(\mathbb{N})$, we would like reductions to be able to find a common reduct as well, for the sake of establishing convertibility between terms, which will have implications on proving a conservativity between $TT(\mathbb{N})$ and its extension. In other words, we want to have the Church-Rosser property for $TT(\mathbb{N})$. And furthermore, for the sake of conservative extension, it is beneficial if we would have normalization in the type theory. Normalization involves the normal forms of terms, and a normal form is a term that cannot be reduced any further by the reduction rules of the system. Examples of normal forms in $TT(\mathbb{N})$ are: $\overline{4}, \overline{9}$, 0 and $s(x)$ where x is a free variable, while the terms $R(0, [x, y]s(\overline{1}), s(0))$ and $R(0, [x, y]0, 0)$ are not normal forms, since they can be reduced to $s(\overline{1})$ and 0, respectively. There are two notions of normalizations [8, p. 22]:

- Weak normalization - every term reduces to a normal form.
- Strong normalization - guarantees that every reduction sequence will eventually terminate.

It is important that we do not confuse normalization and the Church-Rosser property. For instance, in the untyped-lambda calculus, we do have the Church-Rosser property, but we do not have Weak normalization. For instance, there are terms that do not have a normal form, since they can be reduced an infinite amount of times [20]. In $TT(\mathbb{N})$, we expect that every term terminates as well, meaning that it should be strongly normalizing.

Next we want to introduce the Judgement rules for $TT(\mathbb{N})$. Judgement rules provides us the conditions for when a term is of a specific type, given some context with hypothesises. Judgements will help us to reason about the correctness of our definitions and ensure that they meet the requirements of $TT(\mathbb{N})$. The judgement rules for $TT(\mathbb{N})$ can be found in Figure 2. The *Rec* rule allows us to define a function recursively

$$\frac{}{\Gamma, x : \mathbb{N}, \Delta \vdash x : \mathbb{N}}\ Var \qquad \frac{}{\Gamma \vdash 0 : \mathbb{N}}\ Zero \qquad \frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash s(n) : \mathbb{N}}\ Succ$$

$$\frac{\Gamma \vdash a : \mathbb{N} \qquad \Gamma, x : \mathbb{N}, y : \mathbb{N} \vdash b : \mathbb{N} \qquad \Gamma \vdash n : \mathbb{N}}{\Gamma \vdash R(a, [x, y]b, n) : \mathbb{N}}\ Rec$$

Figure 2: Judgement rules for $TT(\mathbb{N})$

over the natural numbers. The term $R(a, [x, y]b, n)$ denotes the natural number $f(n)$, where f is a function defined recursively by:

- $f(0) = a$

- $f(n+1) = b[n/x, f(n)/y]$.

Note that both a and b are terms of type $\mathbb{N}$. The difference between reduction and judgements is that we are not computing anything here, we are verifying that a given term is of a certain type. For example, by having a judgement of $\vdash \overline{3} + \overline{4} : \mathbb{N}$, we can be certain that $\overline{3} + \overline{4}$ is of type $\mathbb{N}$. From the judgement rules, we can ensure that the system is well-formed and can be used to reason about the correctness of our definitions.

1.2.3 Type Theory of Natural Numbers with Pairing

The type theory of natural numbers with pairing is an extension of $TT(\mathbb{N})$, denoted $TT(\mathbb{N}, \times)$. It shares the same syntax, reductions, and judgements as $TT(\mathbb{N})$, but incorporates an additional type and expanded rules. $TT(\mathbb{N}, \times)$ is also a fragment of MLTT that includes a type constructor for pairing. This allows us to define ordered pairs. Additionally, we can use projections to select the elements of a pair, which correspond to the conjunction introduction and elimination rules in MLTT.

The motivation behind investigating the relationship between $TT(\mathbb{N}, \times)$ and $TT(\mathbb{N})$ is their relative simplicity. We can represent pairing in $TT(\mathbb{N})$ with the use of a pairing function. By studying the conservative relation between $TT(\mathbb{N}, \times)$ and $TT(\mathbb{N})$, we expect to find that the former is a conservative extension of the latter. Although $TT(\mathbb{N}, \times)$ introduces an additional type, it also comes with additional rules and terms compared to $TT(\mathbb{N})$.

Definition 1.6. *The syntax for $TT(\mathbb{N}, \times)$ is as following:*

- *The types for $TT(\mathbb{N}, \times)$ are: $A, B ::= \mathbb{N} \mid A \times B$.*
- *The terms for $TT(\mathbb{N}, \times)$ are: $a, b, c ::= 0 \mid z \mid s(a) \mid R(a, [x, y]b, c) \mid \langle a, b\rangle \mid \pi_1(a) \mid \pi_2(a)$*

The terms 0, z, $s(a)$, $R(a, [x, y]b, c)$ are the same as in $TT(\mathbb{N})$(Definition 1.2). The newly added terms in $TT(\mathbb{N}, \times)$ are:

- $\langle a, b\rangle$ which represents an ordered pair of a followed by b,
- $\pi_1(a)$ is the first projection of the term a,
- $\pi_2(a)$ is the second projection of the term a.

However, the judgement rules in $TT(\mathbb{N}, \times)$ are different compared to the ones in $TT(\mathbb{N})$. As we can see in Figure 3, we do not know what type we might have for the *var* rule or in the *Rec* rule. $\times$ is a type constructor, meaning that it enables for constructing more complex terms, for instance, we can have the following types in $TT(\mathbb{N}, \times)$:

- $\mathbb{N}$
- $\mathbb{N} \times \mathbb{N}$
- $\mathbb{N} \times (\mathbb{N} \times \mathbb{N})$

For the reduction rules, see Definition 1.7, we added two additional reductions for when the terms are in the form of either $\pi_1(\langle a, b\rangle)$ or $\pi_2(\langle a, b\rangle)$. When reducing $\pi_1(\langle a, b\rangle)$ or $\pi_2(\langle a, b\rangle)$, we just use projection, meaning that we reduce to either a or b.

Definition 1.7. *The reduction rules for $TT(\mathbb{N}, \times)$ are the following:*

- $R(a, [x, y]b, 0) \rightsquigarrow a$
- $R(a, [x, y]b, s(n)) \rightsquigarrow b[n/x, R(a, [x, y]b, n)/y]$
- $\pi_1(\langle a, b\rangle) \rightsquigarrow a$
- $\pi_2(\langle a, b\rangle) \rightsquigarrow b$

$$\frac{}{\Gamma, x : A, \Delta \vdash x : A}\, Var \qquad \frac{}{\Gamma \vdash 0 : \mathbb{N}}\, Zero \qquad \frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash s(n) : \mathbb{N}}\, Succ$$

$$\frac{\Gamma \vdash p : A \times B}{\Gamma \vdash \pi_1(p) : A}\, \pi_1 \qquad \frac{\Gamma \vdash p : A \times B}{\Gamma \vdash \pi_2(p) : B}\, \pi_2$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\Gamma \vdash \langle a, b \rangle : A \times B}\, Pair \qquad \frac{\Gamma \vdash a : A \qquad \Gamma, x : \mathbb{N}, y : A \vdash b : A \qquad \Gamma \vdash n : \mathbb{N}}{\Gamma \vdash R(a, [x, y]b, n) : A}\, Rec$$

Figure 3: Judgement rules for $TT(\mathbb{N}, \times)$

1.3 Conservative Extension

When we talk about conservative extension, we are referring to the relationship between two formal systems where one system is an extension of the other. Consider two theories T_1 and T_2 with their respective types and terms. Then T_2 is conservative over T_1 if for every proposition P of T_1 we have that if P is provable in T_2 then P is provable in T_1, which will be our notion for propositional conservativity. Propositional conservativity has been studied before in [11, p. 160], and we include it here because some of our notions implies propositional conservativity. What should be noticed, is that no sentences that were true in the original theory, T_1, are altered in the extended theory, T_2. The behavior of the extended theory remains the same as in the original theory, but with additional rules. Conservative extension allow us to explore new features or capabilities without sacrificing the existing results and proofs.

When studying type theory, and for this thesis, we tackle the matter of conservative extension differently. We consider the preservation of judgements between type theories as well as convertibility between terms. We remind ourselves that we write $\Gamma \vdash a : A$ to assert that a is of type A from the terms of their respective types in Γ. We write $a \simeq b$, to say that a and b are convertible. We define four definitions of conservative extension in type theory, where two of them are proposed from us, these include:

- Strong Conservativity
- Canonical Conservativity
- Closed Conservativity
- Propositional Conservativity

The definition of strong conservativity, which involves reductions, is also discussed in [25, Definition 7], but Xue calls it a definitional extension instead. This study contributes to the ongoing investigation of conservative extension, aiming for a deeper exploration of the topic. Canonical conservativity and closed conservativity are the new definitions proposed in this thesis, and they are modifications of strong conservativity, and we will prove that they are weaker than strong conservativity. On the other hand, propositional conservativity matches the definition of conservative extension in first-order logic, which does not involve any reductions. In contrast to the standard formulation, the definition of conservative extension in type theory involves the use of reductions. In general, a conservative extension ensures that all theorems in the extended theory, that are expressible in the language of the original theory, are already theorems in the original theory. Propositional conservativity aligns with the standard formulation, while strong, canonical and closed conservativity incorporate reductions as an additional requirement.

We explore how the type theories of natural numbers align with these definitions and discuss if these would lead to a general definition of a conservative extension. Importantly, these definitions should ensure that existing results and proofs from the original theory remains unchanged.

Strong conservativity combines judgements and convertibility between terms. It ensures that for all types P in the original theory, if we have a judgement of P in the extended theory, then exists a judgement of P in the original theory, and the terms involved that are of type P are convertible. With convertibility, we gain an understanding of reductions between the type theories. If we consider terms as programs, this implies that they can be transformed into each other by using a set of computation rules. Hence, in practical usage, we can have more confidence in running our programs within an extended formal system based on a type theory. Because it ensures that the behavior of the programs computation remains unchanged, if it is strongly conservative over its original theory. Strongly conservative has been previously studied in [25]. However, further investigation is still required to gain a deeper understanding of conservative extensions, and its implications between type theories.

Definition 1.8. *T_2 is strongly conservative over T_1 if and only if for every type P of T_1, every context Γ in T_1 and every term t such that $\Gamma \vdash t : P$ in T_2, then there exists a term s such that:*

- $\Gamma \vdash s : P$ *in* T_1
- $s \simeq t$ *in* T_2

In the definition of canonical conservativity we add more conditions on the hypothesises. We require that all the free variable are substituted with a canonical term. A canonical term is a term that can not be reduced any further and has no free variables. For example, if we would have the term $\vdash \bar{m} + \bar{n} : \mathbb{N}$, then $\bar{m} + \bar{n}$ is not a canonical term, but it can be reduced to a numeral, which are the canonical terms of type $\mathbb{N}$. By using the reduction rules for $TT(\mathbb{N})$ or $TT(\mathbb{N}, \times)$, $\bar{m} + \bar{n} \twoheadrightarrow \overline{m+n}$, which we will prove in later. Because canonical terms are important for the definition of canonical conservativity, we define what it means for a term that is either of type $\mathbb{N}$ or constructed through the pairing type, to be canonical.

Definition 1.9. *A term t is a canonical term of type A, is defined by recursion on A as:*

- *The canonical terms of type $\mathbb{N}$ are the numerals.*
- *A canonical term of type $A \times B$ is a term $\langle t_1, t_2 \rangle$ where t_1 is canonical of type A and t_2 is canonical of type B.*

Some examples of canonical terms in $TT(\mathbb{N}, \times)$:

- $\bar{2}$
- $\langle \bar{1}, \bar{2} \rangle$
- $\langle \langle 0, \bar{5} \rangle, \bar{2} \rangle$

For canonical conservativity, we have a requirement that every term in the hypothesis must be substituted with a canonical term. If we consider a function $f(x)$ with x as a free variable, a type theory that is canonically conservative over its original theory cannot ensure the convertibility of $f(x)$ to a term in its original theory. However, if we substitute x with a canonical term t, then $f(t)$ would guarantee convertibility to a term in its original theory. The key difference here is that strong conservativity can ensure convertibility for a term representing a function $f(x)$ without the need of canonical terms. If we viewed terms as programs, as in the case of strong conservativity, we can transform two terms into each other by using a set of computation rules. However, we are required to substitute all free variables with canonical terms before performing the computation. Therefore, canonical conservativity is weaker than strong conservativity, since it is not always possible to find a term that is convertible with a given term when free variables are involved. However, it still holds valuable information on the relationships between different type theories and their extensions. It provides insights on how the preservation of statements can be achieved when canonical terms are used, even if it may not cover all cases involving functions with free variables.

Definition 1.10. *T_2 is canonically conservative over T_1 if and only if for any types $A_1, ..., A_n, B$ of T_1, every context Γ in T_1 and every term t such that $x_1 : A_1, ..., x_n : A_n \vdash t : B$ in T_2, then there exists a term s such that:*

- $x_1 : A_1, ..., x_n : A_n \vdash s : B$ *in* T_1 *and*
- *for all* $t_1, ..., t_n$ *that are canonical to the types* $A_1, ..., A_n$ *in* T_2*, we have that* $s[t_1/x_1, ..., t_n/x_n] \simeq t[t_1/x_1, ..., t_n/x_n]$ *in* T_2*.*

Closed conservativity is a weaker definition compared to canonical conservativity, as it only considers the empty context. This means that this definition does not take free variables into account. Considering terms as programs, we do not substitute any variables, since we do not consider them as part of the computation. Therefore, we can only guarantee computations for programs that do not involve any variables. However, this definition considers convertibility between terms, ensuring that reductions are taken into account.

Definition 1.11. *T_2 is closed conservative over T_1 if and only if for every type P of T_1, every context Γ in T_1 and every term t such that $\vdash t : P$ in T_2, then there exists a term s such that:*

- $\vdash s : P$ *in* T_1
- $s \simeq t$ *in* T_2

We include the definition of propositional conservativity in order to understand the relationships between the four different definitions conservative extensions, as illustrated in Figure 4. Propositional conservativity focuses on judgements and disregards relations in terms of reductions. Therefore, propositional conservativity is considered one of the weakest forms of conservative extension in the scope of these specific definitions presented in this section. As mentioned before, it has been studied in [11], on the relationship between intentional and extensional formulations of Martin-Löf type theory. The author uses the notion of conservative extension in line with the definition of propositional conservativity presented here.

Definition 1.12. *T_2 is propositionally conservative over T_1 if and only if*

- *For every type P of T_1,*
- *if P is inhabited in T_2 then P is inhabited in T_1.*

As mentioned, Definition 1.12 represents a relatively weak form of conservative relation in the context of type theory, as it solely focuses on the judgements. We want to find a stronger degree of conservative relation in type theory, and one way is to investigate how the reductions behave between type theories. As seen in Definitions 1.8 to 1.11, we involve convertibility, and we get more information on the terms for each type theory.

To demonstrate the strength of the different definitions of conservative extensions, we prove three theorems(Theorems 1.13 to 1.15). These theorems establish the implications of the different types of conservative extension, allowing us to illustrate their implications in Figure 4. Throughout the investigation of the conservative extension between $TT(\mathbb{N})$ and $TT(\mathbb{N}, \times)$, our aim is to establish a strong degree of conservative extension. This is because the other implications of the conservative extension will follow.

Theorem 1.13. *If T_2 is strongly conservative over T_1 then T_2 is canonically conservative over T_1*

Proof. Suppose that T_2 is strongly conservative over T_1, then we already have that for any types $A_1, ..., A_n, B$ and term t such that $x_1 : A_1, ..., x_n : A_n \vdash t : B$ in T_2 from the definition of strongly conservative. We also know that for any term s such that $x_1 : A_1, ..., x_n : A_n \vdash s : B$ in T_1 we have $s \simeq t$ in T_2. Take any canonical terms $t_1, ..., t_n$ of types $A_1, ..., A_n$, respectively, and we get that $s[t_1/x_1, ..., t_n/x_n] \simeq t[t_1/x_1, ..., t_n/x_n]$ from $s \simeq t$. □

Theorem 1.14. *If T_2 is canonically conservative over T_1 then T_2 is closed conservative over T_1*

Proof. In the definition of canonical conservativity(Definition 1.10), if $n = 0$ then we get closed conservativity. □

Theorem 1.15. *If T_2 is canonically conservative over T_1 then T_2 is propositionally conservative over T_1*

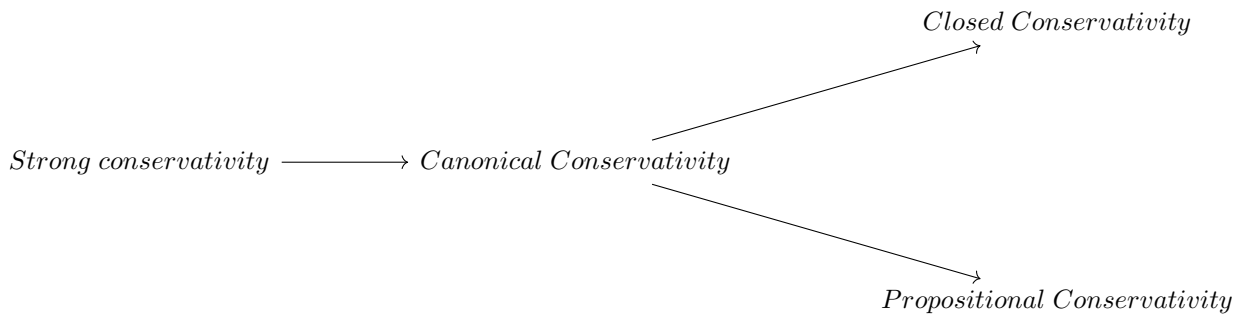

Figure 4: Implications for conservative extensions

Proof. We have that T_2 is canonically conservative over T_1 and from definition we have that for any type B in $T1$ such that $x_1 : A_1, ..., x_n : A_n \vdash t : B$ in T_2 we also have that there is a s such that $x_1 : A_1, ..., x_n : A_n \vdash s : B$ in T_1. Hence, every type that is inhabited in T_2 is also inhabited in T_1. □

Figure 4 explains the implications of the different notions of conservativity, prompted from Theorems 1.13 to 1.15. It is important to note that there are no arrows in the opposite direction. For example, Canonical conservativity does not imply strong conservativity. In a type theory with function types, closed conservativity implies propositional conservativity. However, it is unclear whether closed conservativity implies propositional conservativity without function types, so we cannot include an arrow from closed conservativity to propositional conservativity without further proof.

To illustrate non-conservative extensions, we provide three non-examples. A non-example of strong conservativity is the conservative relation between $TT(\mathbb{N})$ and $TT(\mathbb{N}, \times)$. The reason for this will be explained later as we explore their relationship, but it involves the presence of free variables.

Non-example. *Take $TT(\mathbb{N}, \to)$, that is an extension of $TT(\mathbb{N})$, known as Gödel's System T [1]. In a later section, we will show that the terms of $TT(\mathbb{N})$ and $TT(\mathbb{N}, \times)$ denote the same functions, which are the primitive recursive functions. This means that the terms of $TT(\mathbb{N})$ and $TT(\mathbb{N}, \times)$ can only express the primitive recursive functions, and nothing else. The proof of this relies on $TT(\mathbb{N}, \times)$ being a canonically conservative extension over $TT(\mathbb{N})$. However, $TT(\mathbb{N}, \to)$ can express more than the primitive recursive functions, for instance, the Ackermann function, which is a recursive function that is not primitive. This implies that $TT(\mathbb{N}, \to)$ is not canonically conservative over $TT(\mathbb{N})$. While canonical conservativity may be considered relative weak in terms of the gained information, it is still a valuable insight if it grants us that two theories denote the same functions. This can provide valuable insights about type theories and contribution to their application in fields like proof assistants.*

Non-example. *Let $T(\mathbb{N}, c)$ be an extension of $TT(\mathbb{N})$, with an additional term u, such that:*

$$\frac{}{\Gamma \vdash u : \mathbb{N}}\, u$$

Then $TT(\mathbb{N}, u)$ is not closed conservative over $TT(\mathbb{N})$, because there is no term s in $TT(\mathbb{N})$ such that $s \simeq c$ in $TT(\mathbb{N}, u)$.

Non-example. *Let T_0 be a type theory consisting of one single type $\mathbb{N}$ and no additional terms. Then $TT(\mathbb{N})$ is not propositionally conservative over T_0 since we have $\Gamma \vdash 0 : \mathbb{N}$ in $TT(\mathbb{N})$, but $\mathbb{N}$ is not inhabited in T_0, due to the fact that we have no judgement to assert the type $\mathbb{N}$.*

2 Encodings in $TT(\mathbb{N})$

2.1 Overview

In this section, our focus will be on encoding terms in $TT(\mathbb{N})$ to represent terms in $TT(\mathbb{N}, \times)$. To accomplish this, we will make use of recursion theory in the encoding process. Recursion theory is the study of computable functions. The aim is to provide a formal definition of a computable function, to understand the limitations of what can be computed by an algorithm. We will focus on basic recursive functions for encoding, and therefore, we limit our attention to the primitive recursive functions [18]. Primitive recursive functions can be used to define the most basic operators on natural numbers, such as addition and multiplication. To clarify the basic operators, these include the constant 0, successor function, projection, composition and all the functions h such that:

- $h(x_1, ..., x_n, 0) = f(x_1, ..., x_n)$
- $h(x_1, ..., x_n, succ(y)) = g(x_1, ..., x_n, y, h(x_1, ..., x_n, y))$

where f and g are also primitive recursive. More formally:

Definition 2.1. *The set of primitive recursive functions are defined inductively by:*

- *The constant* 0 *is a primitive recursive function.*
- *The successor function* $succ : \mathbb{N} \to \mathbb{N}$ *with* $succ(\bar{n}) = \overline{n+1}$ *is primitive recursive.*
- *Each projection function* $P_i : \mathbb{N}^n \to \mathbb{N}$ *with* $P_i(x_1, ..., x_n) = x_i$ *and* $i \leq n$ *are primitive recursive.*
- *If*
  - $f : \mathbb{N}^k \to \mathbb{N}$ *is primitive recursive and*
  - *the functions* $g_1, ..., g_k : \mathbb{N}^n \to \mathbb{N}$ *are primitive recursive,*
  - *then the composition* $h(x_1, ..., x_n) = f(g_1(x_1, ..., x_n), ..., g_k(x_1, ..., x_n))$ *is primitive recursive.*
- *If* $f : \mathbb{N}^n \to \mathbb{N}$ *and* $g : \mathbb{N}^{n+2} \to \mathbb{N}$ *are primitive recursive with* $n \geq 1$*, then the function* $h : \mathbb{N}^{n+1} \to \mathbb{N}$ *such that:*
  - $h(x_1, ..., x_n, 0) = f(x_1, ..., x_n)$
  - $h(x_1, ..., x_n, succ(y)) = g(x_1, ..., x_n, y, h(x_1, ..., x_n, y))$

  is primitive recursive.

From these set of rules you can prove that addition and multiplication are primitive recursive. There is a proof in [3, p. 67] which shows that addition is primitive recursive. However, the primitive recursive functions are limited, since there are some computable functions that are not primitive. For example, the Ackermann function, which is defined as:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ a(m-1,1) & \text{if } m \geq 1 \text{and } n = 0 \\ a(m-1, a(m,n-1)) & \text{if } m \geq 1 \text{and } n \geq 1 \end{cases}$$

The value of $A(m,n)$ grows fast, for example, a computer can calculate that

$$A(0,1) = 2, A(1,1) = 3, A(2,1) = 5, A(3,1) = 13$$

very fast, but for $A(4,1) = 65533$ it takes more time to compute due to the amount of recursive calls [17]. The Ackermann function grows faster than any primitive recursive function, and it is a classic example

for showing that there are recursive functions that are not primitive recursive. Some functions that are not primitive recursive include those involving quantification over the natural numbers, for instance, the universal and existential quantifiers, which should output true or false. We want to be able to express quantifiers in $TT(\mathbb{N})$, but since they are not primitive recursive, we can not do so. Therefore, we will limit ourselves to bounded quantifiers. Instead of quantifying over all the natural numbers, we quantify over a finite set of natural numbers.

2.2 The Encodings

We will begin with simple encodings and progress towards more complex ones. Our goal is to encode pairing in $TT(\mathbb{N})$, which we will achieve by using a pairing function that assigns a unique number to each pair. Using this number, we can extract each element of the pair individually. Our aim is to replicate the behavior of $TT(\mathbb{N}, \times)$, but in $TT(\mathbb{N})$, this requires more reduction steps. Throughout this process, we will gain a deeper understanding of how $TT(\mathbb{N})$ can be used. We provide below some basic definitions of operators, and we use the same definition of addition as defined in Definition 1.3:

Definition 2.2. *Define:*

- $a \cdot b := R(0, [x, y](y + a), b)$
- $a^b := R(\bar{1}, [x, y]a \cdot y, b)$
- $pred(n) := R(0, [x, y]x, n)$
- $a \dot{-} b := R(a, [x, y]pred(y), b)$

Multiplication and exponential are self-explanatory. However, since we have a successor term, we can also define a predecessor term as $pred(n)$, which returns 0 if $n = 0$, otherwise it returns $n - 1$. Because we are only working with natural numbers, we cannot define the usual subtraction in $TT(\mathbb{N})$. Instead, we use truncated subtraction, with the following definition:

$$a \dot{-} b = \begin{cases} 0 & \text{if } a \leq b \\ a - b & \text{otherwise} \end{cases}$$

Judgements on some encodings in $TT(\mathbb{N})$

Verifying that a term has a specific type is important in $TT(\mathbb{N})$, for example that addition, multiplication, and exponential is of type $\mathbb{N}$. To achieve this, we will use the judgement rules presented in Figure 2. A derivation of the judgement rules in $TT(\mathbb{N})$ can be significantly long. For instance, if we consider the encodings in Definition 2.2 and try to show the derivation tree of $\vdash (\bar{m} + \bar{n})^{\bar{p}} : \mathbb{N}$, then we would get the derivation tree for exponential, multiplication and addition as well. Hence, the derivation tree will be significantly large and complex. We still want to illustrate the process, so we give two examples of derivations for two basic operators (Examples 2.3 to 2.4).

Example 2.3. $\Gamma \vdash \bar{a} + \bar{b} : \mathbb{N}$ *has the following derivation:*

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash 0 : \mathbb{N}}\,Zero}{\vdots}}{\Gamma \vdash \bar{a} : \mathbb{N}}\,Succ \times a \qquad \dfrac{\dfrac{}{\Gamma, x : \mathbb{N}\,, y : \mathbb{N} \vdash y : \mathbb{N}}\,Var}{\Gamma, x : \mathbb{N}\,, y : \mathbb{N} \vdash s(y) : \mathbb{N}}\,Succ \qquad \dfrac{\dfrac{\dfrac{}{\Gamma \vdash 0 : \mathbb{N}}\,Zero}{\vdots}}{\Gamma \vdash \bar{b} : \mathbb{N}}\,Succ \times b}{\Gamma \vdash \bar{a} + \bar{b} : \mathbb{N}}\,Rec$$

Example 2.4. $\Gamma \vdash pred(\bar{n}) : \mathbb{N}$ *has the following derivation:*

$$\dfrac{\dfrac{}{\Gamma \vdash 0 : \mathbb{N}}\ Zero \quad \dfrac{}{\Gamma, x : \mathbb{N}\,, y : \mathbb{N} \vdash x : \mathbb{N}}\ Var \quad \dfrac{\dfrac{}{\Gamma \vdash 0 : \mathbb{N}}\ Zero}{\vdots}\ Succ \times n}{\Gamma \vdash \bar{n} : \mathbb{N}}$$
$$pred(\bar{n}) : \mathbb{N}$$

Reductions on some encodings in $TT(\mathbb{N})$

It is important to verify that the terms defined in Definition 2.2 reduce to their desired results. We are going to show that if we insert numerals into the basic terms that we have defined, then we will get the result as intended, for instance, we want $\bar{a} + \bar{b}$ to reduce to $\overline{a+b}$. We can have terms like $\bar{a} + (\bar{b} + \bar{c})$, but first we need to reduce it to $\bar{a} + \overline{b+c}$, and then reduce it to $\overline{a+b+c}$. In Theorems 2.5 to 2.9, we will prove that addition, multiplication, exponential, predecessor, and truncated subtraction ($\dot{-}$) perform their intended computation according to the rules we have defined.

Theorem 2.5. $\bar{a} + \bar{b} \twoheadrightarrow \overline{a+b}$.

Proof. By induction on b.

- If $b = 0$, then $\bar{a} + 0 \equiv R(\bar{a}, [x, y]s(y), 0) \rightsquigarrow \bar{a} \equiv \overline{a+0}$.
- If $b > 0$, then $\bar{a} + \bar{b} \equiv R(\bar{a}, [x, y]s(y), \bar{b}) \equiv R(\bar{a}, [x, y]s(y), s(\overline{b-1})) \rightsquigarrow s(R(\bar{a}, [x, y]s(y), \overline{b-1}))$. From the induction hypothesis, $s(R(\bar{a}, [x, y]s(y), \overline{b-1})) \twoheadrightarrow s(\overline{a + (b-1)}) \equiv \overline{a+b}$.

□

Theorem 2.6. $\bar{a}\bar{b} \twoheadrightarrow \overline{ab}$.

Proof. By induction on b.

- If $b = 0$, then $\bar{a} \cdot 0 \equiv R(0, [x, y](y + \bar{a}), 0) \rightsquigarrow 0 \equiv \overline{a \cdot 0}$.
- If $b > 0$, then $\bar{a}\bar{b} \equiv R(0, [x, y](y + \bar{a}), \bar{b}) \equiv R(0, [x, y](y + \bar{a}), s(\overline{b-1})) \rightsquigarrow R(0, [x, y](y + \bar{a}), \overline{b-1}) + \bar{a} \overset{I.H}{\twoheadrightarrow} \overline{a(b-1)} + \bar{a} \overset{Theorem\ 2.5}{\twoheadrightarrow} \overline{a(b-1) + a} \equiv \overline{ab}$.

□

Theorem 2.7. $\bar{a}^{\bar{b}} \twoheadrightarrow \overline{a^b}$.

Proof. By induction on b.

- If $b = 0$, then $\bar{a}^0 \equiv R(\bar{1}, [x, y]\bar{a}y, 0) \rightsquigarrow \bar{1} \equiv \overline{a^0}$.
- If $b > 0$, then $\bar{a}^{\bar{b}} \equiv R(\bar{1}, [x, y]\bar{a}y, \bar{b}) \equiv R(\bar{1}, [x, y]\bar{a}y, s(\overline{b-1})) \rightsquigarrow \bar{a} \cdot R(\bar{1}, [x, y]\bar{a}y, \overline{b-1}) \overset{I.H}{\twoheadrightarrow} \bar{a} \cdot \overline{a^{b-1}} \overset{Theorem\ 2.6}{\twoheadrightarrow} \overline{a \cdot a^{b-1}} \equiv \overline{a^b}$.

□

Theorem 2.8. $pred(\bar{n}) \twoheadrightarrow \overline{n \dot{-} 1}$.

Proof. By induction on n.

- If $n = 0$, then $pred(0) \equiv R(0, [x, y]x, 0) \rightsquigarrow 0$..
- If $n > 0$, then $pred(\bar{n}) \equiv R(0, [x, y]x, \bar{n}) \equiv R(0, [x, y]x, s(\overline{n-1})) \rightsquigarrow \overline{n-1}$.

□

Theorem 2.9. $\bar{a} \dot{-} \bar{b} \twoheadrightarrow \overline{a \dot{-} b}$.

Proof. By induction on b.

- If $b = 0$, then $\bar{a} \dot{-} 0 \equiv R(\bar{a}, [x, y]pred(y), 0) \rightsquigarrow \bar{a}$.
- If $b > 0$, then $\bar{a} \dot{-} \bar{b} \equiv R(\bar{a}, [x, y]pred(y), \bar{b}) \equiv R(\bar{a}, [x, y]pred(y), s(\overline{b-1})) \rightsquigarrow pred(R(\bar{a}, [x, y]pred(y), \overline{b-1})) \overset{I.H}{\twoheadrightarrow} pred(\overline{a \dot{-} (b-1)}) \overset{Theorem\ 2.8}{\twoheadrightarrow} \overline{(a \dot{-} (b-1)) \dot{-} 1} \equiv \overline{a \dot{-} b}$.

□

2.3 Encoding Boolean Operators in TT(ℕ)

In the previous section we saw how basic functions that handles natural numbers can be encoded and used for computations in $TT(\mathbb{N})$. Even if $TT(\mathbb{N})$ is the type theory of natural numbers, we are still able to handle boolean connectives. A boolean connective is a symbol that is used to connect propositions [24]. In $TT(\mathbb{N})$, we are going to define the boolean connectives as terms, and since we are working with the natural numbers as its only type, we interpret $\bar{1}$ as true and 0 as false.

Definition 2.10. *We define true ($\top$) and false ($\bot$) as:*

- $\top := \bar{1}$
- $\bot := 0$

When defining the connectives, we only expect that the inputs are $\top$ and $\bot$. In some cases we can consider 0 to be the only value that is considered as $false$, while everything greater than 0 are considered to be true. For simplicity, we will only think of $\bar{1}$ as being $true$.

The first operator to introduce when dealing with boolean connectives is going to be the **Signum function** [3, p. 69]. The signum function takes a value a, if $a > 0$ then $sg(a) = 1$ otherwise $sg(0) = 0$. The signum function is going to be helpful in order to transform all the numerals greater than 1 to just $\bar{1}$. Then there is a signum function that does the opposite, we will call it **Reverse signum function**. The reverse signum function takes a value a, if $a > 0$ then $\overline{sg}(a) = 0$ otherwise $\overline{sg}(0) = 1$. The reverse signum function is similar to negation, but as mentioned earlier, the reverse signum function deals with all the natural numbers, while negation only deals with $true$ and $false$ in propositional logic [24].

Definition 2.11. *The signum function and the reverse signum function [3]:*

- $sg(a) := R(0, [x, y]\bar{1}, a)$
- $\overline{sg}(a) := R(\bar{1}, [x, y]0, a)$

In order to express the boolean connectives used in propositional logic, we only need to define disjunction and use it together with the reverse signum function, since the two connectives are expressively adequate [21]. What expressively adequate means is that the symbols, in this case disjunction and negation, are capable of expressing all the other boolean connectives in the system. In this case, disjunction and negation are expressively adequate in propositional logic, which we are going to express within $TT(\mathbb{N})$.

Definition 2.12. *Define:* $a \vee b := R(sg(a), [x, y]sg(b), b)$

Then the rest of the connectives for propositional logic, which are conjunction and implication, could be defined by just re-using disjunction and negation.

Definition 2.13. *Define the logical connectives:*

- $a \wedge b := \overline{sg}(\overline{sg}(a) \vee \overline{sg}(b))$
- $a \Rightarrow b := \overline{sg}(a) \vee sg(b)$

Let us verify that disjunction behaves as intended.

- $\top \vee \bot$ should reduce to $\top$: $R(sg(\top), [x, y]sg(\bot), \bot) \rightsquigarrow sg(\top) \rightsquigarrow \top$
- $\bot \vee \bot$ should reduce to $\bot$: $R(sg(\bot), [x, y]sg(\bot), \bot) \rightsquigarrow sg(\bot) \rightsquigarrow \bot$
- $\bot \vee \top$ should reduce to $\top$: $R(sg(\bot), [x, y]sg(\top), \top) \rightsquigarrow sg(\top) \rightsquigarrow \top$
- $\top \vee \top$ should reduce to $\top$: $R(sg(\top), [x, y]sg(\top), \top) \rightsquigarrow sg(\top) \rightsquigarrow \top$

From the boolean connectives that we have defined as operators in $TT(\mathbb{N})$, we are now able to express operators that are able to determine whether the given input is true or false. For instance, the statement "n is even" can be written as $Even(n)$ which takes a natural number n and returns either true or false, depending on whether n is even or odd.

With the help of boolean connectives, we will be able to define more complex operators. We will be able to define operators as "n is less than or equal to m", "n divides m", etc. These operators will be used for the pairing function, as we will see later, which is crucial for translating the pairing term $\langle a, b\rangle$ that originates from $TT(\mathbb{N}, \times)$.

Definition 2.14. *Define the following boolean operators:*

- *Equality:* $a = b := \bar{1} \dot{-} (sg(a \dot{-} b) + (sg(b \dot{-} a))$
- *Inequality:* $a \neq b := \overline{sg}(a = b)$
- *If operator:* $If(a, b, c) := R(c, [x, y]b, a)$
- *Strictly less-than relation:* $a < b := sg(b \dot{-} a)$
- *Less-than or equal:* $a \leq b := (a < b) \vee (a = b)$
- *Is even:* $Even(n) = R(\bar{1}, [x, y]\overline{sg}(y), n)$

What needs to be done next is to show that the operators reduce to their expected result. This is done in the same way as we did for the basic operators, meaning addition, multiplication, exponential, etc. However, we need to state our theorems differently. When we were expecting numerals as a result we could just state the theorems as $a \twoheadrightarrow b$, but since we are working with operators that handle truth values like equality, we need to describe what we mean for two numerals to be equal. We will prove that equality behaves as intended. Verifying that the rest of the operators (Definition 2.14) behave as intended is trivial, and we leave it for the reader to prove.

Theorem 2.15. $\bar{a} = \bar{b} \twoheadrightarrow \bar{1}$ *if and only if* a *and* b *are identical.*

Proof. Suppose that a and b are not identical. Then we have $\bar{a} = \bar{b} \simeq \bar{1} \dot{-} (sg(\bar{a} \dot{-} \bar{b}) + sg(\bar{b} \dot{-} \bar{a}))$. Since a and b are not identical then either $sg(\bar{a} \dot{-} \bar{b}) \twoheadrightarrow \bar{1}$ and $sg(\bar{b} \dot{-} \bar{a}) \twoheadrightarrow 0$; or the other way around. In either way, the sum of them will reduce to $\bar{1}$. Thus, the result will in either way be $\bar{1} \dot{-} (sg(\bar{a} \dot{-} \bar{b}) + sg(\bar{b} \dot{-} \bar{a})) \twoheadrightarrow \bar{1} \dot{-} \bar{1} \twoheadrightarrow 0$. Hence, $\bar{a} = \bar{b} \not\twoheadrightarrow \bar{1}$

For the other direction. Suppose that a and b are identical. Then $\bar{a} = \bar{b} \simeq \bar{1} \dot{-} (sg(\bar{a} \dot{-} \bar{b}) + (sg(\bar{b} \dot{-} \bar{a})) \twoheadrightarrow \bar{1} \dot{-} (sg(0) + sg(0)) \twoheadrightarrow \bar{1} \dot{-} (0 + 0) \twoheadrightarrow \bar{1}$. Which shows that, $\bar{a} = \bar{b} \twoheadrightarrow \bar{1}$ □

The final operators to define in this section is the existential and universal quantifiers. The existential quantifier says that there exists a x in a domain such that a given sentence is true. While the universal quantifier says that the given sentence is true for all x in the domain. There is one issue with these two statements. In $TT(\mathbb{N})$ we can not express infinite terms and one requirement is to quantify over infinite domains, for the universal and existential quantifiers. This means that we can not express the whole set of natural numbers, since the set of natural numbers is infinite. However, we can still express the quantifiers in a smaller degree. We need to limit ourselves to a finite domain, meaning that we have a finite range over what we quantify over. The idea is to give that range to the input of the **bounded quantifier** operators.

Definition 2.16. *Bounded quantifiers:*

- *Existential:* $\exists x \leq n.P(x) := R(P(0), [x, y]If(P(s(x)), \bar{1}, y), n)$
- *Universal:* $\forall x \leq n.P(x) := R(P(0), [x, y]If(P(s(x)), y, 0), n)$

2.4 Pairing in $TT(\mathbb{N})$

As we have seen throughout the previous two sections, $TT(\mathbb{N})$ is expressive enough to define some basic operators and operators that handle truth values, and able to perform computation on them. We will now use the definitions for the basic operators and the operators that handle the boolean connectives, to encode an ordered pair in $TT(\mathbb{N})$. For instance, if we have the terms a and b, we want to be able to put them into an ordered pair as (a, b). Given a pair (a, b) we want to be able to pick the first or second element of the tuple by using a projection function.

The technique that is going to be used in order to encode pairing is by using a pairing function [3, p. 7]. A pair of (a, b) is going to be encoded into a *unique* number, and afterwards, we want to define two projection functions in order to compute the first and the second element of the pair represented by this unique number. The unique number of a pair (a, b) is going to be the number $2^a \cdot (2b + 1)$. The projection functions, first projection and second projection, are going to find the greatest value of x such that 2^x and $2x+a$ respectively, divides $2^a \cdot (2b + 1)$. In other words, the projection functions are going to find the greatest value such that 2^x or $2x + a$ divides the uniquely encoded pair, and the greatest value x is the element in the pair that we want to retrieve. To achieve this we need more definitions.

Definition 2.17. *The Divides operator is defined as:* $m|n := \exists x \leq n.n = mx$.

$m|n$ is an important operator in order to define the pairing function, therefore we prove that it actually returns 1 if m divides n:

Theorem 2.18. $\bar{m}|\bar{n} \twoheadrightarrow \top$ *if and only if* m *divides* n.

Proof. Suppose $\bar{m}|\bar{n} \twoheadrightarrow \top$. Then there is a x such that $n = mx$. If we solve the equation for x, we get that $x = n/m$, and we know that x is a natural number. Therefore, we get that m divides n, since the division does not give us any remainders.

For the other direction, suppose that m divides n. then $n/m = x$ with no remainders and x is a natural number. Therefore, $n = mx$, so there exists such an x, which is the same as our definition of $m|n$. Thus, $m|n \twoheadrightarrow \top$. □

As mentioned earlier, we will use $m|n$ in order to calculate the first and second element of an ordered pair. The only thing we need to prove is that the projection functions work as intended.

Definition 2.19. *Functions for pairing:*

- *Pairing:* $pr(a, b) := \bar{2}^a \cdot s(b + b)$
- *Projection 1:* $p_1(n) := R(0, [x, y]If(\bar{2}^x|n, x, y), n)$
- *Projection 2:* $p_2(n) := R(0, [x, y]If(s(x + x)|n, x, y), n)$

Theorem 2.20. $p_1(pr(\bar{a}, \bar{b})) \twoheadrightarrow \bar{a}$.

Proof. Recall that $p_1(pr(\bar{a}, \bar{b})) \equiv R(0, [x, y]If(\bar{2}^x|\bar{2}^{\bar{a}} \cdot s(\bar{b} + \bar{b}), x, y), \bar{2}^{\bar{a}} \cdot s(\bar{b} + \bar{b}))$. We already know that 2^a divides $2^a \cdot (2b + 1)$ and by Theorem 2.18, $\bar{2}^{\bar{a}}|\bar{2}^{\bar{a}} \cdot s(\bar{b} + \bar{b}) \twoheadrightarrow \top$. Then what we want to show is that there is no $m > a$ such that $\bar{2}^{\bar{m}}|\bar{2}^{\bar{a}} \cdot s(\bar{b} + \bar{b}) \twoheadrightarrow \top$. Suppose $m \equiv a + n$ for some $n > 0$. Then we can see that $\bar{2}^{\bar{m}} \equiv \bar{2}^{\bar{a}+\bar{n}} \equiv \bar{2}^{\bar{a}} \cdot \bar{2}^{\bar{n}}$. Notice now that $2^a \cdot 2^n$ does not divide $2^a \cdot (2b + 1)$, since 2^n does not divide $2b + 1$ because 2^n is even and $2b + 1$ is odd. From Theorem 2.18, $\bar{2}^{\bar{m}}|\bar{2}^{\bar{a}} \cdot s(\bar{b} + \bar{b}) \twoheadrightarrow \bot$ and therefore $R(0, [x, y]If(\bar{2}^x|\bar{2}^{\bar{a}} \cdot s(\bar{b} + \bar{b}), x, y), \bar{2}^{\bar{a}} \cdot s(\bar{b} + \bar{b})) \twoheadrightarrow \bar{a}$. □

Theorem 2.21. $p_2(pr(\bar{a}, \bar{b})) \twoheadrightarrow \bar{b}$.

Proof. Similar to the proof of Theorem 2.20. □

For illustration, we show here a practical example for an encoding of $TT(\mathbb{N})$ in Haskell. If we implemented pairing as we have defined in $TT(\mathbb{N})$, then the computation might take a lot of time. If we try to project a pair whose unique number is significantly large then we might not be able to compute it at all. For instance, consider the computation of the pairing function that is written in Haskell in Figure 5 (The implementation of $TT(\mathbb{N})$ can be found in Appendix A). The function i is utilized in i *smallPair* and i *largePair* to convert numerals into integers, making them easier to interpret. Here we see that the *smallPair* is an encoding of the pair $pr(\bar{1}, \bar{2})$. We are able to retrieve the first and second element of the *smallPair*, by using the projection functions. However, if we let *largePair* be the pair $pr(\bar{5}, \bar{3})$, we see that the value of *largePair* is significantly larger than *smallPair*. When we are trying to pick the first element of *largePair* by using projection, we get the error message "*** Exception: stack overflow". The stack gets too big because we are doing too many call by recursion.

```
*Main> let smallPair = pr (n 1) (n 2)
*Main> i smallPair
10
*Main> p1 smallPair
S Zero
*Main> p2 smallPair
S (S Zero)
*Main> let largePair = pr (n 5) (n 3)
*Main> i largePair
224
*Main> p1 largePair
*** Exception: stack overflow
```

Figure 5: Computing the projections of a pair in Haskell.

3 The Church-Rosser Theorem

The Church-Rosser Theorem says that for a given term M, if we reduce it in zero or multiple reduction steps to either N or P, then there exists a term Z such that N and P can reduce in zero or multiple reduction steps to Z. It gives us the guarantee that every reduction can always find a common reduct, meaning a common term. If the Church-Rosser theorem is true for a reduction, then it has the Church-Rosser Property. Figure 6 describes the Church-Rosser property, where Z is the common reduct that can be obtained from N and P. In Simply-typed Lambda Calculus, the Church-Rosser property holds for β-reduction [8, p. 22]. We are going to prove the Church-Rosser property for $TT(\mathbb{N})$ and $TT(\mathbb{N}, \times)$, separately, in the upcoming sections. When we are proving Church-Rosser property for $TT(\mathbb{N})$ and $TT(\mathbb{N}, \times)$, we employ a similar strategy as in the Simply-typed Lambda Calculus for β-reduction. However, the difference is that we consider different reduction rules that are in $TT(\mathbb{N})$, as well as different reduction rules that are in $TT(\mathbb{N}, \times)$. The Church-Rosser property is beneficial for proving properties about reductions, and moreover establishing convertibility between terms. Therefore, by having Church-Rosser property for a reduction can be valuable in the sense of convertibility, and has implications for proving conservative extensions between $TT(\mathbb{N})$ and $TT(\mathbb{N}, \times)$.

3.1 Church-Rosser Property for $TT(\mathbb{N})$

The Church-Rosser property is a fundamental property that guarantees that when applying reduction rules on a given term, we will always be able to find a common reduct between the terms. The Church-Rosser

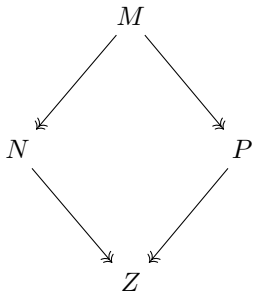

Figure 6: Church-Rosser Property

theorem states that if a term can be reduced to two different terms, in zero or several reduction steps, then there exists a way to reduce both of these terms to a common reduct.

More formally, we need to show that for any reductions $M \twoheadrightarrow N$ and $M \twoheadrightarrow P$, there is a term Z such that both $N \twoheadrightarrow Z$ and $P \twoheadrightarrow Z$. It would be convenient if the property as stated in Figure 7 held true. That is if for all one-step reductions $M \rightsquigarrow N$ and $M \rightsquigarrow P$ there is a term Z such that both $N \rightsquigarrow Z$ and $P \rightsquigarrow Z$. Unfortunately, that is not the case in $TT(\mathbb{N})$, consider the term $R(0, [x, y]R(s(0), [x', y']y', s(0)), s(0))$. We can perform one-step reductions in two ways:

$$R(0, [x, y]R(s(0), [x', y']y', s(0)), s(0)) \rightsquigarrow R(0, [x, y]R(s(0), [x', y']y', 0), s(0)) \rightsquigarrow R(0, [x, y]s(0), s(0)) \quad (1)$$

or we can do

$$R(0, [x, y]R(s(0), [x', y']y', s(0)), s(0)) \rightsquigarrow R(s(0), [x', y']y', s(0)) \rightsquigarrow R(s(0), [x', y']y', s(0)) \quad (2)$$

Although the resulting terms $R(0, [x, y]s(0), s(0))$ and $R(s(0), [x', y']y', s(0))$ are not the same, they do reduce to the normal form $s(0)$. This example shows that the one-step reduction does not always guarantee that we can find a common reduct for all possible reductions. We need to tackle this matter from a different angle.

Diamond property for One-step Reduction

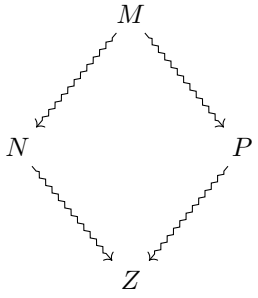

Figure 7: Diamond property for One-Step Reduction that we do not have for $TT(\mathbb{N})$

What we can agree on is that the terms can always reduce to a common reduct, but the computation may be different for each step. We are going to define a new reduction relation called **parallel one-step reduction** and we write it as $\rightsquigarrow'$. By showing that for any two terms, there exists a common reduct that both can be reduced to and that $\twoheadrightarrow$ is the reflexive-transitive closure of $\rightsquigarrow'$, we will establish the Church-Rosser property for $TT(\mathbb{N})$.

After defining the parallel one-step reduction, we need to show that a computation for a one-step reduction can be done in a parallel one-step reduction too. A property that is important when proving Church-Rosser

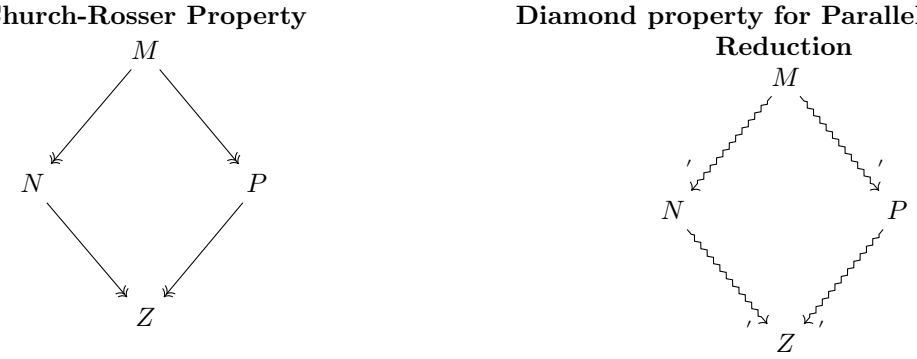

Figure 8: Church-Rosser Property and Diamond Property for Parallel One-Step Reduction

property is the diamond property, which is illustrated in the right-hand side of Figure 8. The consequence of having the diamond property and that $\twoheadrightarrow$ is the reflexive-transitive closure of $\rightsquigarrow'$, will be the Church-Rosser property. The diamond property is false for one-step reduction, so instead we need to prove the diamond property for parallel one-step reduction as shown in Figure 8. The rules for parallel-one step reduction is shown in Figure 9. The first step is to show that for any one-step reduction we also have a parallel one-step

$$\frac{}{t \rightsquigarrow' t}\ (1) \qquad \frac{t \rightsquigarrow' t'}{s(t) \rightsquigarrow' s(t')}\ (2) \qquad \frac{a \rightsquigarrow' a'}{R(a,[x,y]b,0) \rightsquigarrow' a'}\ (3)$$

$$\frac{a \rightsquigarrow' a' \qquad b \rightsquigarrow' b' \qquad c \rightsquigarrow' c'}{R(a,[x,y]b,c) \rightsquigarrow' R(a',[x,y]b',c')}\ (4) \qquad \frac{a \rightsquigarrow' a' \qquad b \rightsquigarrow' b' \qquad c \rightsquigarrow' c'}{R(a,[x,y]b,s(c)) \rightsquigarrow' b'[c'/x, R(a',[x,y]b',c')/y]}\ (5)$$

Figure 9: Rules for parallel one-step reduction for $TT(\mathbb{N})$

reduction in $TT(\mathbb{N})$. A minor clarification before we proceed to the proofs in this section: When we refer to rule (x) for $x = 1, 2, 3, 4$; we are referring to the rules from Figure 9.

Lemma 3.1. *For all terms M and M', if $M \rightsquigarrow M'$ then $M \rightsquigarrow' M'$.*

Proof. We proceed by case analysis on the reduction of $M \rightsquigarrow M'$:

1. If the rule is recursion of base 0, then we have $R(a,[x,y]v,0) \rightsquigarrow a$. By rule (1), we have $a \rightsquigarrow' a$. Then, using rule (3) together with $a \rightsquigarrow' a$, we get $R(a,[x,y]b,0) \rightsquigarrow' a$.

2. If the rule is $R(a,[x,y]b,s(n)) \rightsquigarrow b[n/x, R(a,[x,y]b,b)/y]$, then by rule (1), we have $a \rightsquigarrow' a$, $b \rightsquigarrow' b$, and $n \rightsquigarrow' n$. By applying rule (5), we get $R(a,[x,y]b,s(n)) \rightsquigarrow' b[n/x, R(a,[x,y]b,n)/y]$.

□

If we perform a parallell one-step reduction, then we want to know if we can do the same action but with zero or more one-step reductions instead. This means that if we can do one computation in parallel, then it should be possible to perform the same computation but with one-step reductions sequentially.

Lemma 3.2. *For all terms M and M', if $M \rightsquigarrow' M'$, then $M \twoheadrightarrow M'$.*

Proof. By induction over the reduction rules of $M \rightsquigarrow' M'$.

- If the last rule is (1), then we already have $t \twoheadrightarrow t$ from zero steps.
- If the last rule is (2), then from the I.H., we have that $t \twoheadrightarrow t'$. Thus, then we also have $s(t) \twoheadrightarrow s(t')$ because of the I.H.
- If the last rule is (3), then $R(a, [x, y]b, 0) \rightsquigarrow' a'$. Then, when we reduce, we can do the following: $R(a, [x, y]b, 0) \rightsquigarrow a$, and from the I.H., we have $a \twoheadrightarrow a'$. Therefore, we have $R(a, [x, y]b, 0) \twoheadrightarrow a'$.
- If the last rule is (4), then $R(a, [x, y]b, c) \rightsquigarrow' R(a', [x, y]b', c')$ with $a \rightsquigarrow' a'$, $b \rightsquigarrow' b'$, and $c \rightsquigarrow' c'$. By the I.H., we have $a \twoheadrightarrow a'$, $b \twoheadrightarrow b'$, and $c \twoheadrightarrow c'$. So we do $R(a, [x, y]b, c) \twoheadrightarrow R(a', [x, y]b, c) \twoheadrightarrow R(a', [x, y]b', c) \twoheadrightarrow R(a', [x, y]b', c')$.
- If the last rule is (5), then $R(a, [x, y]b, s(c)) \rightsquigarrow' b'[c'/x, R(a', [x, y]b', c')/y]$ with $a \rightsquigarrow' a'$, $b \rightsquigarrow' b'$, and $c \rightsquigarrow' c'$. By the I.H., we have $a \twoheadrightarrow a'$, $b \twoheadrightarrow b'$, and $c \twoheadrightarrow c'$. So we do $R(a, [x, y]b, s(c)) \rightsquigarrow b[c/x, R(a, [x, y]b, c)/y] \twoheadrightarrow b[c/x, R(a', [x, y]b, c)/y] \twoheadrightarrow b'[c/x, R(a', [x, y]b', c)/y] \twoheadrightarrow b'[c'/x, R(a', [x, y]b', c')/y]$.

□

The next step is to show that $\twoheadrightarrow$ is the reflexive-transitive closure of $\rightsquigarrow'$. In a more formal way, we define the reflexive-transitive closure as:

Definition 3.3. *Given a reduction relation R, we define the reflexive-transitive closure R^* of R to be the smallest relation that contains R and is reflexive and transitive.*

The following lemma will help us show that if the diamond property holds for $\rightsquigarrow'$, then its transitive closure should also have the diamond property.

Remark. In the following proof, we treat the reduction relations $\rightsquigarrow$, $\rightsquigarrow'$ and $\twoheadrightarrow$ as sets. When writing $A \subseteq B$, it means that A is a subset or equal to B.

Lemma 3.4. *$\twoheadrightarrow$ is the reflexive-transitive closure of $\rightsquigarrow'$*

Proof. By Lemma 3.1, we have $\rightsquigarrow \subseteq \rightsquigarrow'$. We know that $\twoheadrightarrow = \rightsquigarrow^*$, therefore, $\rightsquigarrow^* \subseteq (\rightsquigarrow')^*$, hence $\twoheadrightarrow \subseteq (\rightsquigarrow')^*$. By Lemma 3.2, we have $\rightsquigarrow' \subseteq \twoheadrightarrow$. Hence, $(\rightsquigarrow')^* \subseteq \twoheadrightarrow^* = \twoheadrightarrow$. Therefore, since $(\rightsquigarrow')^* \subseteq \twoheadrightarrow$ and $\twoheadrightarrow \subseteq (\rightsquigarrow')^*$ we get that $(\rightsquigarrow')^* = \twoheadrightarrow$. □

As stated before, with the help of Lemma 3.4, we are able to establish the Church-Rosser property for $TT(\mathbb{N})$. However, what we have not yet proved is the diamond property for the parallel one-step reduction in $TT(\mathbb{N})$. Therefore, we need to introduce a crucial concept called **maximal parallel one-step reduction**. The maximal parallel one-step reduction is a term that can be reached from two previous terms, both originating from a single term. Consider again Figure 8 for the $\rightsquigarrow'$ figure, the Z here will be our maximal parallel one-step reduction. In other words, this will give us the guarantee that any two parallel reduction sequences can always find a common reduct. Which is exactly what we need to establish the diamond property for $\rightsquigarrow'$.

Definition 3.5. *We define the maximal parallel one-step reduction of a term M as M^* inductively as follows::*

- $0^* := 0$
- $z^* := z$ *(for a variable z)*
- $(s(n))^* := s(n^*)$
- $(R(a, [x, y]b, c))^* := R(a^*, [x, y]b^*, c^*)$ *(If c is not 0 or $s(n)$)*
- $(R(a, [x, y]b, 0))^* := a^*$
- $(R(a, [x, y]b, s(c)))^* := b^*[c^*/x, R(a^*, [x, y]b^*, c^*)/y]$

Lemma 3.6. *Whenever $M \rightsquigarrow' M'$ then $M' \rightsquigarrow' M^*$*

Proof. By induction over the reduction rules of $M \rightsquigarrow' M'$. Assume that all bound variables have been renamed to avoid clashes.

- If (1) is the last rule, then $M \equiv M'$ and we want to show that $t \rightsquigarrow' t^*$, so we need to do another induction on the term t:
  - If $t \equiv 0$ then $t^* \equiv 0^* \equiv 0 \equiv t$, so from (1) $t \rightsquigarrow' t^*$.
  - If $t \equiv z$ is similar to when $t \equiv 0$.
  - If $t \equiv s(c)$ then we have that $s(c) \rightsquigarrow' s(c)$ and by I.H. $c \rightsquigarrow' c^*$, and we use (2) with $c \rightsquigarrow' c^*$ to obtain that $s(c) \rightsquigarrow' s(c^*) \equiv t^*$.
  - If $t \equiv R(a, [x, y]b, c)$ then we have that $R(a, [x, y]b, c) \rightsquigarrow' R(a, [x, y]b, c)$. We do a case analysis on c:
    - If $c = 0$ then $(R(a, [x, y]b, 0))^* = a^*$ and from I.H. $a \rightsquigarrow' a^*$ and $b \rightsquigarrow' b^*$, use (3) with $a \rightsquigarrow' a^*$ to get that $R(a, [x, y]b, 0) \rightsquigarrow' a^*$.
    - If $c = s(d)$ then $(R(a, [x, y]b, s(d)))^* \equiv b^*[d^*/x, R(a^*, [x, y]b^*, d^*)/y]$ and from I.H. $a \rightsquigarrow' a^*$, $b \rightsquigarrow' b^*$ and $d \rightsquigarrow' d^*$, use (5) and $R(a, [x, y]b, s(d)) \rightsquigarrow' b^*[d^*/x, R(a^*, [x, y]b^*, d^*)/y]$.
    - If c is neither 0 or $s(d)$ then $(R(a, [x, y]b, c))^* \equiv R(a^*, [x, y]b^*, c^*)$ from I.H. $a \rightsquigarrow' a^*$, $b \rightsquigarrow' b^*$ and $c \rightsquigarrow' c^*$, use (4) and $R(a, [x, y]b, c) \rightsquigarrow' R(a^*, [x, y]b^*, c^*)$.
- If (2) is the last rule, then $s(t) \rightsquigarrow' s(t')$, so $M \equiv s(t)$ and $M' \equiv s(t')$ with $t \rightsquigarrow' t'$. By I.H. $t' \rightsquigarrow' t^*$, apply (2) and we get that $s(t') \rightsquigarrow' s(t^*) \equiv s(t)^* \equiv M^*$.
- If (3) is the last rule, then $R(a, [x, y]b, 0) \rightsquigarrow' a'$ with $a \rightsquigarrow' a'$. By I.H. $a' \rightsquigarrow' a^*$, hence we have $R(a, [x, y]b, 0) \rightsquigarrow' a' \rightsquigarrow' a^* \equiv (R(a, [x, y]b, 0))^* \equiv M^*$.
- If (4) is the last rule, then $R(a, [x, y]b, c) \rightsquigarrow' R(a', [x, y]b', c')$. We do a case analysis on c:
  - If $c = 0$ then $M^* \equiv (R(a, [x, y]b, 0))^* \equiv a^*$. We have that $M' \equiv R(a', [x, y]b', 0)$ with $a \rightsquigarrow' a'$, $b \rightsquigarrow' b'$ and from (1) $0 \rightsquigarrow' 0$. From I.H. $a' \rightsquigarrow' a^*$, use (3) and we get that $R(a', [x, y]b', 0) \rightsquigarrow' a^* \equiv M^*$.
  - If $c = s(d)$ for some term d then $M^* \equiv (R(a, [x, y]b, s(d)))^* \equiv b^*[c^*/x, R(a^*, [x, y]b^*, d^*)/y]$. We have that $M' \equiv R(a', [x, y]b', s(d'))$ with $a \rightsquigarrow' a'$, $b \rightsquigarrow' b'$ and from (2) $s(d) \rightsquigarrow' s(d')$ with $d \rightsquigarrow' d'$. By I.H. $a' \rightsquigarrow' a^*$, $b' \rightsquigarrow' b^*$ and $d' \rightsquigarrow' d^*$, use (5) and we get that $R(a', [x, y]b', s(d')) \rightsquigarrow' b^*[d^*/x, R(a^*, [x, y]b^*, d^*)/y] \equiv M^*$.
  - If c is not 0 or $s(d)$, then we have $a \rightsquigarrow' a'$, $b \rightsquigarrow' b'$ and $c \rightsquigarrow' c'$. By I.H. $a' \rightsquigarrow' a^*$, $b' \rightsquigarrow' b^*$ and $c' \rightsquigarrow' c^*$. Apply (4) and we get that $R(a', [x, y]b', c') \rightsquigarrow' R(a^*, [x, y]b^*, c^*) \equiv M^*$.
- If (5) is the last rule, then $R(a, [x, y]b, s(c)) \rightsquigarrow' b'[c'/x, R(a', [x, y]b', c')/y]$ with $a \rightsquigarrow' a'$, $b \rightsquigarrow' b'$ and $c \rightsquigarrow' c'$. By I.H., $a' \rightsquigarrow' a^*$, $b' \rightsquigarrow' b^*$, and $c' \rightsquigarrow' c^*$. Applying (5), we get $R(a', [x, y]b', s(c')) \rightsquigarrow' b^*[c^*/x, R(a^*, [x, y]b^*, c^*)/y] \equiv (R(a, [x, y]b, s(c)))^* \equiv M^*$.

□

By using Lemma 3.6, we can prove that the diamond property holds for $\rightsquigarrow'$ by the following lemma.

Lemma 3.7 (The Diamond Property for $\rightsquigarrow'$ in $TT(\mathbb{N})$)**.** *If $M \rightsquigarrow' N$ and $M \rightsquigarrow' P$ then there exists Z such that $N \rightsquigarrow' Z$ and $P \rightsquigarrow' Z$.*

Proof. Take $Z \equiv M^*$ and by Lemma 3.6, the result follows. □

As we have proved the diamond property for $\rightsquigarrow'$ in Lemma 3.7, we are going to show that from the diamond property we can achieve the Church-Rosser property for $\twoheadrightarrow$.

Theorem 3.8 (Church-Rosser property for $\twoheadrightarrow$ in $TT(\mathbb{N})$)**.** *If $M \twoheadrightarrow N$ and $M \twoheadrightarrow P$, then there exists Z such that $N \twoheadrightarrow Z$ and $P \twoheadrightarrow Z$.*

Proof. By Lemma 3.7, $\rightsquigarrow'$ satisfies the diamond property. It follows from a result about Church-Rosser [2, Lemma 3.2.2 on p. 59] that the reflexive-transitive closure of a relation that satisfies the diamond property also satisfies the diamond property. By Lemma 3.4, we know that $\twoheadrightarrow$ is the reflexive-transitive closure of $\rightsquigarrow'$. Therefore, $\twoheadrightarrow$ also satisfies the diamond property, and so we have the Church-Rosser property for $\twoheadrightarrow$. □

We now present a useful corollary that will be instrumental in showing that two terms are convertible, which is helpful for establishing a conservative extension between type theories.

Corollary 3.9. *In $TT(\mathbb{N})$, $a \simeq b$ if and only if there exists a c such that $a \twoheadrightarrow c$ and $b \twoheadrightarrow c$.*

Proof. If $a \twoheadrightarrow c$ and $b \twoheadrightarrow c$ then we already have $a \twoheadrightarrow c \twoheadleftarrow b$, by definition of convertable relation, $a \simeq b$. An identical proof for the other direction can be found from a result about Church-Rosser [2, Theorem 3.1.13 on p. 54], and since we have Church-Rosser property for $\twoheadrightarrow$, we can use that theorem. □

3.2 Church-Rosser Property for $TT(\mathbb{N}, \times)$

For $TT(\mathbb{N}, \times)$, we also have the Church-Rosser Property. The proof is similar to the proof for Church-Rosser property for $TT(\mathbb{N})$ (Section 3.1). The difference is that we need to add more rules for the parallel one-step reduction. The Church-Rosser property cannot be directly inferred from $TT(\mathbb{N})$ because the parallel one-step reduction in $TT(\mathbb{N}, \times)$ (See Figure 10) has more rules compared to the parallel one-step reduction in $TT(\mathbb{N})$ (See Figure 9). The reason is that it introduces additional scenarios that need to be considered in our proofs for Church-Rosser in $TT(\mathbb{N}, \times)$. For example, when proving Lemma 3.12, we need to consider additional cases. We have included some extra cases to consider in Lemma 3.12 to illustrate the process. Similarly, if we would reduce an R-term in Lemma 3.12, then we would have the same scenario as in Lemma 3.6, where we need to perform case distinction on the term c. Because of these factors, the proofs becomes larger as we encounter greater variety of terms in $TT(\mathbb{N}, \times)$ compared to $TT(\mathbb{N})$. We provide in this section the necessary rules, definitions and theorems that needs to be proven in order to achieve the Church-Rosser property for $TT(\mathbb{N}, \times)$. The process of proving Church-Rosser for $TT(\mathbb{N}, \times)$ is similar as in $TT(\mathbb{N})$, but requires considering more cases. We leave some details for the reader to verify.

$$\frac{}{t \rightsquigarrow' t} \ (1) \qquad \frac{t \rightsquigarrow' t'}{s(t) \rightsquigarrow' s(t')} \ (2) \qquad \frac{a \rightsquigarrow' a'}{R(a, [x,y]b, 0) \rightsquigarrow' a'} \ (3)$$

$$\frac{a \rightsquigarrow' a' \qquad b \rightsquigarrow' b' \qquad c \rightsquigarrow' c'}{R(a, [x,y]b, c) \rightsquigarrow' R(a', [x,y]b', c')} \ (4)$$

$$\frac{a \rightsquigarrow' a' \qquad b \rightsquigarrow' b' \qquad c \rightsquigarrow' c'}{R(a, [x,y]b, s(c)) \rightsquigarrow' b'[c'/x, R(a', [x,y]b', c')/y]} \ (5)$$

$$\frac{a \rightsquigarrow' a'}{\pi_1(a) \rightsquigarrow' \pi_1(a')} \ (6) \qquad \frac{a \rightsquigarrow' a'}{\pi_2(a) \rightsquigarrow' \pi_2(a')} \ (7)$$

$$\frac{a \rightsquigarrow' a'}{\pi_1(\langle a, b\rangle) \rightsquigarrow' a'} \ (8) \qquad \frac{b \rightsquigarrow' b'}{\pi_2(\langle a, b\rangle) \rightsquigarrow' b'} \ (9)$$

$$\frac{a \rightsquigarrow' a' \qquad b \rightsquigarrow' b'}{\langle a, b\rangle \rightsquigarrow' \langle a', b'\rangle} \ (10)$$

Figure 10: Rules for parallel one-step reduction for $TT(\mathbb{N}, \times)$

Lemma 3.10. *$\twoheadrightarrow$ is the reflexive-transitive closure of $\rightsquigarrow'$*

Proof. The proof is similar to lemma 3.4, but we need to swap Lemma 3.1 and Lemma 3.2 for a corresponding version of the $\rightsquigarrow'$ relation for $TT(\mathbb{N}, \times)$. □

Definition 3.11. *In $TT(\mathbb{N}, \times)$, we write the maximal parallel one-step reduct of a term M as M^* inductively as follows:*

- $0^* := 0$
- $z^* := z$ *(for a variable z)*
- $(s(n))^* := s(n^*)$
- $(R(a, [x, y]b, c))^* := R(a^*, [x, y]b^*, c^*)$ *(If c is not 0 or $s(n)$)*
- $(R(a, [x, y]b, 0))^* := a^*$
- $(R(a, [x, y]b, s(c)))^* := b^*[c^*/x, R(a^*, [x, y]b^*, c^*)]$
- $(\pi_1(a))^* := \pi_1(a^*)$ *(For a that is not of the form $\langle a_1, a_2\rangle$)*
- $(\pi_2(a))^* := \pi_2(a^*)$ *(For a that is not of the form $\langle a_1, a_2\rangle$)*
- $(\pi_1(\langle a, b\rangle))^* := a^*$
- $(\pi_2(\langle a, b\rangle))^* := b^*$
- $(\langle a, b\rangle)^* := \langle a^*, b^*\rangle$

Lemma 3.12. *In $TT(\mathbb{N}, \times)$, whenever $M \rightsquigarrow' M'$ then $M' \rightsquigarrow' M^*$*

Proof. By induction over the reduction rules of $M \rightsquigarrow' M'$. We do three cases and leave the rest to the reader:

- If (6) is the last rule, then $\pi_1(a) \rightsquigarrow' \pi_1(a')$ with $a \rightsquigarrow' a'$. By I.H., $a' \rightsquigarrow' a^*$. Applying (6), we get $\pi_1(a') \rightsquigarrow' \pi_1(a^*) \equiv (\pi_1(a))^* \equiv M^*$.
- If (8) is the last rule, then $\pi_1(\langle a, b\rangle) \rightsquigarrow' a'$ with $a \rightsquigarrow' a'$. By I.H., $a' \rightsquigarrow' a^* \equiv (\pi_1(\langle a, b\rangle))^* \equiv M^*$.
- If (10) is the last rule, then $\langle a, b\rangle \rightsquigarrow' \langle a', b'\rangle$ with $a \rightsquigarrow' a'$ and $b \rightsquigarrow' b'$. By I.H., $a' \rightsquigarrow' a^*$ and $b' \rightsquigarrow' b^*$. Applying (10), we get $\langle a', b'\rangle \rightsquigarrow' \langle a^*, b^*\rangle \equiv (\langle a, b\rangle)^* \equiv M^*$.

□

Lemma 3.13 (The Diamond Property for $\rightsquigarrow'$ in $TT(\mathbb{N}, \times)$)**.** *If $M \rightsquigarrow' N$ and $M \rightsquigarrow' P$ then there exists Z such that $N \rightsquigarrow' Z$ and $P \rightsquigarrow' Z$.*

Proof. Take $Z \equiv M^*$ and by Lemma 3.12, the result follows.

□

Theorem 3.14 (Church-Rosser property for $\twoheadrightarrow$ in $TT(\mathbb{N}, \times)$)**.** *If $M \twoheadrightarrow N$ and $M \twoheadrightarrow P$ then there exists Z such that $N \twoheadrightarrow Z$ and $P \twoheadrightarrow Z$.*

Proof. Similar to Theorem 3.8, but use Lemma 3.10 and Lemma 3.13 instead □

Corollary 3.15. *In $TT(\mathbb{N}, \times)$, $a \simeq b$ if and only if there exists a c such that $a \twoheadrightarrow c$ and $b \twoheadrightarrow c$.*

Proof. Identical to Corollary 3.9. □

4 Translation and Conservative Extension

In this section, we utilize the encodings mentioned in Section 2, to translate from $TT(\mathbb{N}, \times)$ to $TT(\mathbb{N})$. By using the pairing function, we can represent pairs in $TT(\mathbb{N})$ and extract their individual elements, as we have proven it. This demonstrates similar behaviours between $TT(\mathbb{N}, \times)$ and $TT(\mathbb{N})$.

After defining the translations, we need to ensure that the judgements behave the same, which are requirements that we need to fulfill for the different notions of conservativity. To verify some notions completely, we need to include reductions, and this is the part where Section 3 will come in handy, as we have proven results about convertibility between terms in $TT(\mathbb{N})$ and $TT(\mathbb{N}, \times)$.

4.1 Defining structures of the translation from $TT(\mathbb{N}, \times)$ to $TT(\mathbb{N})$

We present here how a translation of a given term of $TT(\mathbb{N}, \times)$ is formed in $TT(\mathbb{N})$. As presented earlier, we use the encodings of a pair in $TT(\mathbb{N})$ which will represent the translated version of pairing in $TT(\mathbb{N}, \times)$.

Definition 4.1. *For any term a of $TT(\mathbb{N}, \times)$, define the term $[\![a]\!]$ of $TT(\mathbb{N})$ inductively as follows:*

- $[\![x]\!] := x$
- $[\![0]\!] := 0$
- $[\![s(n)]\!] := s([\![n]\!])$
- $[\![R(a, [x, y]b, n)]\!] := R([\![a]\!], [x, y][\![b]\!], [\![n]\!])$
- $[\![\langle a, b\rangle]\!] := pr([\![a]\!], [\![b]\!])$
- $[\![\pi_1(a)]\!] := p_1([\![a]\!])$
- $[\![\pi_2(a)]\!] := p_2([\![a]\!])$

Next we show that substitution is also preserved as expected, meaning that if we have $[\![M[N/x]]\!]$ then this term is equivalent to $[\![M]\!][[\![N]\!]/x]$.

Lemma 4.2. *For any terms M, N we have that $[\![M[N/x]]\!] \equiv [\![M]\!][[\![N]\!]/x]$.*

Proof. By induction on M. We do two base cases and one case for the induction step. The ones left out are similar:

- If $M \equiv x$ then $[\![x[N/x]]\!] \equiv [\![N]\!]$ and $[\![x]\!][[\![N]\!]/x] \equiv x[[\![N]\!]/x] \equiv [\![N]\!]$.
- If $M \equiv 0$ then $[\![0[N/x]]\!] \equiv [\![0]\!] \equiv 0 \equiv [\![0[[\![N]\!]/x]]\!] \equiv [\![[\![0]\!][[\![N]\!]/x]]\!]$.
- If $M \equiv \pi_1(a)$ then $[\![\pi_1(a)[N/x]]\!] \equiv [\![\pi_1(a[N/x])]\!] \equiv p_1([\![a[N/x]]\!]) \overset{I.H}{\equiv} p_1([\![a]\!][[\![N]\!]/x])$ and $[\![\pi_1(a)]\!][[\![N]\!]/x] \equiv p_1([\![a]\!])[[\![N]\!]/x] \equiv p_1([\![a]\!][[\![N]\!]/x])$.

□

4.2 Translating Judgements

This section provides two theorems for judgement translations from $TT(\mathbb{N}, \times)$ to $TT(\mathbb{N})$. All the notions of conservativity mentioned in Section 1.3, involves the preservation of derivable judgement between the two theories. This means that proving the preservation of derivable judgement is important, as it gives us that we can transfer theorems between type theories. Theorem 4.3 is trivial, since $TT(\mathbb{N}, \times)$ is an extension of $TT(\mathbb{N})$ then every judgement in $TT(\mathbb{N})$ is obviously a judgement in $TT(\mathbb{N}, \times)$, no translation here is needed. For Theorem 4.4 we need to do more work, and we prove the theorem by induction on the derivation.

Theorem 4.3. *Every derivable judgement of $TT(\mathbb{N})$ is a derivable judgement of $TT(\mathbb{N}, \times)$*

Proof. Trivial. □

Theorem 4.4. *If $\mathcal{J}$ is a derivable judgement of $TT(\mathbb{N}, \times)$ then $[\![\mathcal{J}]\!]$ is a derivable judgement of $TT(\mathbb{N})$.*

Proof. By induction on the derivation of $\mathcal{J}$.

We do a case distinction on the last rule of the derivation, some parts of the derivations are omitted for the reader to prove:

- If the last rule is a *var* we translate it as:

$$\frac{}{\Gamma, x : A, \Delta \vdash x : A}\, Var \qquad\qquad \frac{}{[\![\Gamma]\!], [\![x]\!] : \mathbb{N}\,, \Delta \vdash [\![x]\!] : \mathbb{N}}\, Var$$

- If the last rule is a *Zero* we translate it as:

$$\frac{}{\Gamma \vdash 0 : \mathbb{N}}\, Zero \qquad\qquad \frac{}{[\![\Gamma]\!] \vdash [\![0]\!] : \mathbb{N}}\, Zero$$

- If the last rule is a *Succ* we translate it as:

$$\frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash s(n) : \mathbb{N}}\, Succ \qquad\qquad \frac{\begin{array}{c}\vdots \\ I.H. \\ \vdots \\ [\![\Gamma]\!] \vdash [\![n]\!] : \mathbb{N}\end{array}}{[\![\Gamma]\!] \vdash [\![s(n)]\!] : \mathbb{N}}\, Succ$$

- If the last rule is a π_1 we translate it as:

$$\frac{\Gamma \vdash p : A \times B}{\Gamma \vdash \pi_1(p) : A}\, \pi_1$$

$$\dfrac{\dfrac{}{[\![\Gamma]\!] \vdash 0 : \mathbb{N}}\, Zero \qquad \dfrac{\dfrac{}{[\![\Gamma]\!], y : \mathbb{N} \vdash y : \mathbb{N}}\, Var \qquad \dfrac{}{[\![\Gamma]\!], x : \mathbb{N} \vdash x : \mathbb{N}}\, Var \qquad \dfrac{\vdots}{[\![\Gamma]\!], x : \mathbb{N} \vdash \overline{2}^x | n : \mathbb{N}}}{[\![\Gamma]\!], x : \mathbb{N}\,, y : \mathbb{N} \vdash If(\overline{2}^x | n, x, y) : \mathbb{N}}\, Rec \qquad [\![\Gamma]\!] \vdash [\![p]\!] : \mathbb{N}}{[\![\Gamma]\!] \vdash p_1([\![p]\!]) : \mathbb{N}}\, Rec$$

- π_2 is similar as for π_1:

- If the last rule is a *Pair* we translate it as:

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\Gamma \vdash \langle a, b \rangle : A \times B}\, Pair$$

$$\dfrac{\dfrac{}{[\![\Gamma]\!] \vdash 0 : \mathbb{N}}\, Zero \qquad \dfrac{\dfrac{}{[\![\Gamma]\!], y : \mathbb{N} \vdash y : \mathbb{N}}\, Var \qquad \dfrac{\dfrac{}{[\![\Gamma]\!], y' : \mathbb{N} \vdash y' : \mathbb{N}}\, Var}{[\![\Gamma]\!], y' : \mathbb{N} \vdash s(y') : \mathbb{N}}\, Succ \qquad \dfrac{\vdots}{[\![\Gamma]\!] \vdash 2^a : \mathbb{N}}}{[\![\Gamma]\!], y : \mathbb{N} \vdash y + 2^a : \mathbb{N}}\, Rec \qquad \dfrac{\vdots}{[\![\Gamma]\!] \vdash b + b : \mathbb{N}}}{[\![\Gamma]\!] \vdash s(b + b) : \mathbb{N}}\, Succ}{[\![\Gamma]\!] \vdash pr(a, b) : \mathbb{N}}\, Rec$$

- If the last rule is a *Rec* we translate it as:

$$\frac{\Gamma \vdash a : A \qquad \Gamma, x : \mathbb{N}, y : A \vdash b : A \qquad \Gamma \vdash n : \mathbb{N}}{\Gamma \vdash R(a, [x, y]b, n) : A}\ Rec$$

$$\frac{\begin{array}{ccc} \vdots & \vdots & \vdots \\ I.H. & I.H. & I.H. \\ \vdots & \vdots & \vdots \\ [\![\Gamma]\!] \vdash [\![a]\!] : \mathbb{N} & [\![\Gamma]\!], x : \mathbb{N}, y : \mathbb{N} \vdash [\![b]\!] : \mathbb{N} & [\![\Gamma]\!] \vdash [\![n]\!] : \mathbb{N} \end{array}}{[\![\Gamma]\!] \vdash R([\![a]\!], [x, y][\![b]\!], [\![n]\!]) : \mathbb{N}}\ Rec$$

□

4.3 Translating Reductions

In this section, we focus on proving properties related to convertibility between terms. We consider terms where all the free variables are substituted with canonical terms. To remind the reader, a canonical term is one that cannot be further reduced and contains no free variables inside it. By substituting all the free variables with canonical terms, we simplify the process of convertibility between terms, as we eliminate the presence of free variables. Free variables can complicate the proof of conservativity, as we will discuss in the next section for strong conservativity. We will demonstrate that due to the existance of free variables, $TT(\mathbb{N})$ and $TT(\mathbb{N}, \times)$ are not strongly conservative.

First, we begin by proving some properties about reductions that should be trivial for a translated term:

Theorem 4.5. *Suppose in $TT(\mathbb{N}, \times)$:*

- $\Gamma \vdash M : A$
- $\Gamma \vdash N : A$

then

1. *if $M \rightsquigarrow N$ then $[\![M]\!] \twoheadrightarrow [\![N]\!]$*
2. *if $M \twoheadrightarrow N$ then $[\![M]\!] \twoheadrightarrow [\![N]\!]$*
3. *if $M \simeq N$ then $[\![M]\!] \simeq [\![N]\!]$*

Proof. We prove each property:

1. Proof by induction on the reduction of $M \rightsquigarrow N$.
   - If $R(a, [x, y]b, 0) \rightsquigarrow a$ then $[\![R(a, [x, y]b, 0)]\!] \equiv R([\![a]\!], [x, y][\![b]\!], 0) \rightsquigarrow [\![a]\!] \equiv [\![N]\!]$.
   - If $R(a, [x, y]b, s(n)) \rightsquigarrow b[n/x, R(a, [x, y]b, n)/y]$ then $[\![R(a, [x, y]b, s(n))]\!] \equiv R([\![a]\!], [x, y][\![b]\!], s([\![n]\!])) \rightsquigarrow [\![b]\!][[\![n]\!]/x, R([\![a]\!], [x, y][\![b]\!], [\![n]\!])/y] \equiv [\![b]\!][[\![n]\!]/x, [\![R(a, [x, y]b, n)]\!]/y] \overset{Lemma\ 4.2}{\equiv} [\![b[n/x, R(a, [x, y]b, n)/y]]\!]$.
   - If $\pi_1(\langle a, b\rangle) \rightsquigarrow a$ then $[\![\pi_1(\langle a, b\rangle)]\!] \equiv p_1([\![\langle a, b\rangle]\!]) \equiv p_1(pr([\![a]\!], [\![b]\!])) \overset{Theorem\ 2.20}{\twoheadrightarrow} [\![a]\!]$.
   - For $\pi_2(\langle a, b\rangle) \rightsquigarrow b$ it is similar as in $\pi_1(\langle a, b\rangle) \rightsquigarrow a$.
   - If some $a \rightsquigarrow b$ is the reduction inside a term M.
     Consider $M[a/x] \rightsquigarrow M[b/x]$ then $[\![M[a/x]]\!] \overset{Lemma\ 4.2}{\equiv} [\![M]\!][[\![a]\!]/x] \overset{I.H.}{\twoheadrightarrow} [\![M]\!][[\![b]\!]/x] \overset{Lemma\ 4.2}{\equiv} [\![M[b/x]]\!]$.
2. Suppose $M \twoheadrightarrow N$ then we have a sequence $M \rightsquigarrow M_0 \rightsquigarrow M_1 \rightsquigarrow ... \rightsquigarrow M_n \rightsquigarrow N$. From (1) we have $[\![M]\!] \twoheadrightarrow [\![M_0]\!] \twoheadrightarrow [\![M_1]\!] \twoheadrightarrow ... \twoheadrightarrow [\![M_n]\!] \twoheadrightarrow [\![N]\!]$ and thus we have $[\![M]\!] \twoheadrightarrow [\![N]\!]$.

3. Suppose that we have $M \simeq N$ then we have that $M \twoheadrightarrow Z$ and $N \twoheadrightarrow Z$ from Corollary 3.15. From (2) we get that $[\![M]\!] \twoheadrightarrow [\![Z]\!]$ and $[\![N]\!] \twoheadrightarrow [\![Z]\!]$, use Corollary 3.15 and we get that $[\![M]\!] \simeq [\![N]\!]$.

□

Theorem 4.7 is what we need in order to prove canonical conservativity. This theorem states that when we have a term of a specific type, with free variables inside the term, if we substitute all the free variables with canonical terms, then the resulting term is convertible to its translation. To prove Theorem 4.7, we require an additional theorem, which states that if we substitute all the free variables inside a term with canonical terms, then the resulting term reduces to a canonical term.

Theorem 4.6. *Suppose that in $TT(\mathbb{N}, \times)$, if $x_1 : A_1, ..., x_n : A_n \vdash b : B$ and the terms $t_1, ..., t_n$ are canonical to the types $A_1, ..., A_n$, respectively. Then $b[t_1/x_1, ..., t_n/x_n]$ reduces to a canonical term of type B.*

Proof. By induction on the derivation of $x_1 : A_1, ..., x_n : A_n \vdash b : B$.

- If $x_1 : A_1, ..., x_n : A_n \vdash 0 : \mathbb{N}$ then 0 is already in its canonical form, which is a numeral.
- If $x_1 : A_1, ..., x_n : A_n \vdash z : B$ then let $z \equiv x_i$ and $B = A_i$ for some $0 < i \leq n$. Take the canonical term t_i of type A_i and the term $x_i[t_i/x_i] \equiv t_i$, which is a canonical term of type $A_i = B$.
- If $x_1 : A_1, ..., x_n : A_n \vdash s(a) : \mathbb{N}$ with $x_1 : A_1, ..., x_n : A_n \vdash a : \mathbb{N}$ we get from I.H. that the term $a[t_1/x_1, ..., t_n/x_n]$ reduces to a canonical term m of type $\mathbb{N}$, which is a numeral from the definition of canonical terms. Then we have the term $s(a)[t_1/x_1, ..., t_n/x_n] \equiv s(a[t_1/x_1, ..., t_n/x_n]) \twoheadrightarrow s(m)$, which is a numeral and a canonical term of type $\mathbb{N}$.
- If $x_1 : A_1, ..., x_n : A_n \vdash R(a, [x, y]b, c) : B$ with $x_1 : A_1, ..., x_n : A_n \vdash a : B$, $x_1 : A_1, ..., x_n : A_n, x : \mathbb{N}, y : B \vdash b : B$ $x_1 : A_1, ..., x_n : A_n \vdash c : \mathbb{N}$. From the I.H. the terms $a[t_1/x_1, ..., t_n/x_n]$ and $c[t_1/x_1, ..., t_n/x_n]$ reduces to the canonical terms t_a and t_c of types B and $\mathbb{N}$, respectively. Therefore, the term $R(a, [x, y]b, c)[t_1/x_1, ..., t_n/x_n] \equiv R(a[t_1/x_1, ..., t_n/x_n], [x, y]b[t_1/x_1, ..., t_n/x_n], c[t_1/x_1, ..., t_n/x_n]) \twoheadrightarrow R(t_a, [x, y]b[t_1/x_1, ..., t_n/x_n], t_c)$. t_c is a canonical term of type $\mathbb{N}$, from the definition of a canonical term, it is a numeral. We do a case analysis on t_c:
  - If $t_c = 0$ then $R(t_a, [x, y]b[t_1/x_1, ..., t_n/x_n], 0) \rightsquigarrow t_a$ and t_a is a canonical term of type B.
  - If $t_c = s(m)$. Notice that m is a numeral since t_c is a numeral.
    Then the term $R(t_a, [x, y]b[t_1/x_1, ..., t_n/x_n], s(m)) \rightsquigarrow$
    $b[m/x, R(t_a, [x, y]b[t_1/x_1, ..., t_n/x_n], m)/y, t_1/x_1, ..., t_n/x_n]$. We need to do another induction on m to show that the term $b[m/x, R(t_a, [x, y]b[t_1/x_1, ..., t_n/x_n], m)/y, t_1/x_1, ..., t_n/x_n]$ reduces to a canonical term of type B.
    - If $m = 0$ then $b[0/x, R(t_a, [x, y]b[t_1/x_1, ..., t_n/x_n], 0)/y, t_1/x_1, ..., t_n/x_n] \rightsquigarrow$
      $b[0/x, t_a/y, t_1/x_1, ..., t_n/x_n]$ where we substitute all the variables to its canonical form and from the outer induction this reduces to a canonical term of type B.
    - If $m = s(m')$ then $b[s(m')/x, R(t_a, [x, y]b[t_1/x_1, ..., t_n/x_n], s(m'))/y, t_1/x_1, ..., t_n/x_n] \rightsquigarrow$
      $b[s(m')/x, b[m'/x, R(t_a, [x, y]b[t_1/x_1, ..., t_n/x_n], m')/y]/y, t_1/x_1, ..., t_n/x_n]$. From the inner I.H. $b[m'/x, R(t_a, [x, y]b[t_1/x_1, ..., t_n/x_n], m')/y]$ reduces to a canonical term t_b of type B and therefore we get that $b[s(m')/x, b[m'/x, R(t_a, [x, y]b[t_1/x_1, ..., t_n/x_n], m')/y]/y, t_1/x_1, ..., t_n/x_n] \twoheadrightarrow$
      $b[s(m')/x, t_b/y, t_1/x_1, ..., t_n/x_n]$. From the outer induction the term $b[s(m')/x, t_b/y, t_1/x_1, ..., t_n/x_n]$ reduces to a canonical term of type B.
- If $x_1 : A_1, ..., x_n : A_n \vdash \pi_1(p) : B$ with $x_1 : A_1, ..., x_n : A_n \vdash p : B \times C$ for some type C. From the I.H. the term $p[t_1/x_1, ..., t_n/x_n]$ reduces to a canonical term of type $B \times C$. From the definition of a canonical term, the canonical of $p[t_1/x_1, ..., t_n/x_n]$ is $\langle u_1, u_2 \rangle$ where u_1 is canonical of type B and u_2 is canonical of type C. Then the term $\pi_1(p)[t_1/x_1, ..., t_n/x_n] \equiv \pi_1(p[t_1/x_1, ..., t_n/x_n]) \twoheadrightarrow \pi_1(\langle u_1, u_2 \rangle) \rightsquigarrow u_1$, and u_1 is a canonical term of type B.

- $x_1 : A_1, ..., x_n : A_n \vdash \pi_2(p) : B$ is similar as in $x_1 : A_1, ..., x_n : A_n \vdash \pi_1(p) : B$.

□

Theorem 4.7. *In $TT(\mathbb{N}, \times)$, if $x_1 : A_1, ..., x_n : A_n \vdash a : \mathbb{N}$ and the terms $t_1, ..., t_n$ are canonical to the types $A_1, ..., A_n$, respectively. Then $a[t_1/x_1, ..., t_n/x_n] \simeq [\![a[t_1/x_1, ..., t_n/x_n]]\!]$ in $TT(\mathbb{N}, \times)$.*

Proof. Suppose that $x_1 : A_1, ..., x_n : A_n \vdash a : \mathbb{N}$ and the terms $t_1, ..., t_n$ are canonical to the types $A_1, ..., A_n$, respectively. By Theorem 4.6, $a[t_1/x_1, ..., t_n/x_n]$ reduces to a canonical term m of type $\mathbb{N}$, which means that m is a numeral. We know that $a[t_1/x_1, ..., t_n/x_n] \twoheadrightarrow m$ and $m \twoheadrightarrow m$, and by Corollary 3.15, $a[t_1/x_1, ..., t_n/x_n] \simeq m$ and from Theorem 4.5, $[\![a[t_1/x_1, ..., t_n/x_n]]\!] \simeq [\![m]\!]$. We know that m is a numeral and by Definition 4.1, it follows immediately that $m \equiv [\![m]\!]$, which gives us that $a[t_1/x_1, ..., t_n/x_n] \simeq m \equiv [\![m]\!] \simeq [\![a[t_1/x_1, ..., t_n/x_n]]\!]$.
□

4.4 Conservative Relation between $TT(\mathbb{N}, \times)$ and $TT(\mathbb{N})$

In this section, we investigate how strong the conservative relation is between $TT(\mathbb{N})$ and $TT(\mathbb{N}, \times)$. The first thought would be that since we are able to encode the pairing function inside $TT(\mathbb{N})$, then $TT(\mathbb{N}, \times)$ should be strongly conservative over $TT(\mathbb{N})$ (Definition 1.8). However, $TT(\mathbb{N}, \times)$ is not strongly conservative over $TT(\mathbb{N})$ and the reason is that it fails when it comes to free variables. Consider the Fibonacci Sequence [19, p. 158]:

- $fib(0) = 0$
- $fib(1) = 1$
- $fib(n) = fib(n-1) + f(n-2)$

In $TT(\mathbb{N}, \times)$ we can define fib as well. Let $fib := \pi_1(R(\langle 0, 1\rangle, [x, y]\langle \pi_2(y), \pi_1(y)+\pi_2(y)\rangle, n))$, when calculating the fifth term of the fibonacci sequence we do it recursively by:

$0th$: $\pi_1(R(\langle 0, 1\rangle, [x, y]\langle \pi_2(y), \pi_1(y) + \pi_2(y)\rangle, 0)) \rightsquigarrow \pi_1(\langle 0, 1\rangle) \rightsquigarrow 0$

$1st$: $\pi_1(R(\langle 0, 1\rangle, [x, y]\langle \pi_2(y), \pi_1(y) + \pi_2(y)\rangle, 1)) \twoheadrightarrow \pi_1(\langle 1, 1\rangle) \rightsquigarrow 1$

$2nd$: $\pi_1(R(\langle 0, 1\rangle, [x, y]\langle \pi_2(y), \pi_1(y) + \pi_2(y)\rangle, 2)) \twoheadrightarrow \pi_1(\langle 1, 2\rangle) \rightsquigarrow 1$

$3rd$: $\pi_1(R(\langle 0, 1\rangle, [x, y]\langle \pi_2(y), \pi_1(y) + \pi_2(y)\rangle, 3)) \twoheadrightarrow \pi_1(\langle 2, 3\rangle) \rightsquigarrow 2$

$4th$: $\pi_1(R(\langle 0, 1\rangle, [x, y]\langle \pi_2(y), \pi_1(y) + \pi_2(y)\rangle, 4)) \twoheadrightarrow \pi_1(\langle 3, 5\rangle) \rightsquigarrow 3$

$5th$: $\pi_1(R(\langle 0, 1\rangle, [x, y]\langle \pi_2(y), \pi_1(y) + \pi_2(y)\rangle, 5)) \twoheadrightarrow \pi_1(\langle 5, 8\rangle) \rightsquigarrow 5$

The property the term fib has is that it memorizes its previous two sequences and adds them together to calculate the next sequence. However, the term we have defined $n : \mathbb{N} \vdash fib : \mathbb{N}$ is not convertible to its translation $n : \mathbb{N} \vdash [\![fib]\!] : \mathbb{N}$. First notice that fib and $[\![fib]\!]$ are not the same term, then what is left is that we need to find a term that both reduces to in order to show that they are convertible, by following Corollary 3.15. The term $n : \mathbb{N} \vdash fib : \mathbb{N}$ can not be reduced any further, because the free variable n is blocking the computation and fib becomes a stuck term [13]. A stuck term is a term that is not in its canonical form and can not be reduced any further from the given reduction rules of a type theory. Therefore, we can not reduce fib any further. We can not reduce $[\![fib]\!]$ to fib either, since fib is of the form $\pi_1(...)$ and $[\![fib]\!]$ translates it to $p_1(...)$, which excludes all the $\pi_1(...)$ forms, so $[\![fib]\!] \not\twoheadrightarrow fib$. Then there exists no common reduct between fib and $[\![fib]\!]$, which means that they are not convertible by Corollary 3.15. More generally, fib is not convertible to any term than itself, which gives us the following lemma:

Lemma 4.8. *There is no s in $TT(\mathbb{N})$ such that $\Gamma \vdash s : \mathbb{N}$ and $s \simeq fib$ in $TT(\mathbb{N}, \times)$.*

Proof. Take any s such that $\Gamma \vdash s : \mathbb{N}$ in $TT(\mathbb{N})$. Since s is a term of $TT(\mathbb{N})$, there is no term of the form $\pi_1(p)$ for some term p inside s. Therefore, s cannot be transformed into a term of the form $\pi_1(p)$, implying that $s \not\equiv fib$ and $s \not\twoheadrightarrow fib$. Examining the reduction rules of $TT(\mathbb{N}, \times)$, we observe that fib is a normal form, meaning there is no a such that $fib \rightsquigarrow a$. This indicates that fib can only reduce to itself with zero reduction steps, i.e., $fib \twoheadrightarrow fib$ and nothing else. Since we only have $fib \twoheadrightarrow fib$ for fib and $s \not\twoheadrightarrow fib$, then there is no term z such that both $fib \twoheadrightarrow z$ and $s \twoheadrightarrow z$. By corollary 3.15, we conclude that $s \not\simeq fib$. □

Now we can use fib as a counterexample to demonstrate that $TT(\mathbb{N}, \times)$ is not strongly conservative over $TT(\mathbb{N})$.

Theorem 4.9. *$TT(\mathbb{N}, \times)$ is not strongly conservative over $TT(\mathbb{N})$.*

Proof. Take the Fibonacci term $n : \mathbb{N} \vdash fib : \mathbb{N}$ in $TT(\mathbb{N}, \times)$, and by Lemma 4.8, there is no s in $TT(\mathbb{N})$ such that $\Gamma \vdash s : \mathbb{N}$ and $s \simeq fib$ in $TT(\mathbb{N}, \times)$. Which means that $TT(\mathbb{N}, \times)$ is not strongly conservative over $TT(\mathbb{N})$. □

$TT(\mathbb{N})$ and $TT(\mathbb{N}, \times)$ are not strongly conservative as we have defined in Definition 1.8. They are still conservative though on a lower degree. The issue we have is the free variable, which introduces stuck terms. We can take care with the issue of the stuck terms with the properties we have proved about reductions involving canonical terms.

Theorem 4.10. *$TT(\mathbb{N}, \times)$ is canonically conservative over $TT(\mathbb{N})$.*

Proof. The only type we have in $TT(\mathbb{N})$ is $\mathbb{N}$. Suppose we have $x_1 : \mathbb{N}, ..., x_n : \mathbb{N} \vdash t : \mathbb{N}$ in $TT(\mathbb{N}, \times)$ for any t and that $t_1, ..., t_n$ are canonical to the type $\mathbb{N}$, i.e. $t_1, ..., t_n$ are numerals.
By Theorem 4.3 we get that $x_1 : \mathbb{N}, ..., x_n : \mathbb{N} \vdash [\![t]\!] : \mathbb{N}$ in $TT(\mathbb{N})$ and by Theorem 4.7, $t[t_1/x_1, ..., t_n/x_n] \simeq [\![t[t_1/x_1, ..., t_n/x_n]]\!]$ in $TT(\mathbb{N}, \times)$. □

Corollary 4.11. *$TT(\mathbb{N}, \times)$ is closed conservative over $TT(\mathbb{N})$ and $TT(\mathbb{N}, \times)$ is propositionally conservative over $TT(\mathbb{N})$.*

Proof. By Theorem 4.10, $TT(\mathbb{N}, \times)$ is canonically conservative over $TT(\mathbb{N})$. By Theorem 1.14 and Theorem 1.15, they are closed and propositionally conservative as well. □

4.5 Type Theory of Natural Numbers and Primitive Recursive Functions

In this section, we will see that functions that are denoted by a term in $TT(\mathbb{N})$ are the primitive recursive functions. When encodings were made in $TT(\mathbb{N})$, we encoded only functions that are primitive recursive. And because we have showed that $TT(\mathbb{N}, \times)$ is canonically conservative over $TT(\mathbb{N})$, the functions that are denoted by the terms in $TT(\mathbb{N}, \times)$ are also primitive recursive. This demonstrates the results of $TT(\mathbb{N}, \times)$ being canonically conservative over $TT(\mathbb{N})$, which has impact on future research on canonical conservativity. First we need to define what it means for a function to be denoted by a term.

Definition 4.12. *The function denoted by $x_1 : \mathbb{N}, ..., x : n : \mathbb{N} \vdash t : \mathbb{N}$ is f_t where $f_t(r_1, ..., r_n) = s$ if and only if $t[\overline{r_1}/x_1, ..., \overline{r_n}/x_n] \twoheadrightarrow \bar{s}$.*

Next, we use the knowledge of the canonically conservative relation between $TT(\mathbb{N})$ and $TT(\mathbb{N}, \times)$ and show that a given term that denotes one function in $TT(\mathbb{N}, \times)$ also denotes the same function in $TT(\mathbb{N})$.

Theorem 4.13. *If $x_1 : \mathbb{N}, ..., x_n : \mathbb{N} \vdash t : \mathbb{N}$ in $TT(\mathbb{N}, \times)$ then it denotes the same function as $x_1 : \mathbb{N}, ..., x_n : \mathbb{N} \vdash [\![t]\!] : \mathbb{N}$ in $TT(\mathbb{N})$.*

Proof. Let f_t be the function that is denoted by $x_1 : \mathbb{N}, ..., x_n : \mathbb{N} \vdash t : \mathbb{N}$ in $TT(\mathbb{N}, \times)$. By definition, $t[\overline{r_1}/x_1, ..., \overline{r_n}/x_n] \twoheadrightarrow f_t(r_1, ..., r_n)$, and since $\overline{r_1}, ..., \overline{r_n}$ are numerals we know by Theorem 4.7, that $t[\bar{r_1}/x_1, ..., \bar{r_n}/x_n] \simeq [\![t[\bar{r_1}/x_1, ..., \bar{r_n}/x_n]]\!] \equiv [\![t]\!][\bar{r_1}/x_1, ..., \bar{r_n}/x_n]$. We know that $\bar{s}$ is a numeral, so it is also canonical of type $\mathbb{N}$, which means it can not be reduced any further. From $t[\overline{r_1}/x_1, ..., \overline{r_n}/x_n] \twoheadrightarrow$

$\bar{s}$, $t[\bar{r_1}/x_1, ..., \bar{r_n}/x_n] \simeq [\![t]\!][\bar{r_1}/x_1, ..., \bar{r_n}/x_n]$ and Corollary 3.15, We know that $t[\bar{r_1}/x_1, ..., \bar{r_n}/x_n] \twoheadrightarrow Z$, $[\![t]\!][\bar{r_1}/x_1, ..., \bar{r_n}/x_n] \twoheadrightarrow Z$ and $Z \twoheadrightarrow \bar{s}$ which gives us that $[\![t]\!][\bar{r_1}/x_1, ..., \bar{r_n}/x_n] \twoheadrightarrow \bar{s}$. Hence, $f_{[\![t]\!]}(r_1, ..., r_n) = f_t(r_1, ..., r_n)$. □

Finally, we prove that all the terms in $TT(\mathbb{N})$ denote exactly the primitive recursive function. Then we use Theorem 4.13, in order to show that all the terms in $TT(\mathbb{N}, \times)$ denote exactly the primitive recursive function as well. Recall also that we can do this since $TT(\mathbb{N}, \times)$ is a canonically conservative extension over $TT(\mathbb{N})$.

Theorem 4.14. *The functions denoted by the terms of $TT(\mathbb{N})$ are exactly the primitive recursive functions.*

Proof. First we prove that for every term $x_1 : \mathbb{N}, .., x_n : \mathbb{N} \vdash t : \mathbb{N}$, such that f_t is denoted by t, then f_t is primitive recursive. We do induction on the derivation of $x_1 : \mathbb{N}, .., x_n : \mathbb{N} \vdash t : \mathbb{N}$.

- If f_t is denoted by $x_1 : \mathbb{N}, .., x_n : \mathbb{N} \vdash 0 : \mathbb{N}$ then f_t is constant 0.
- If f_t is denoted by $x_1 : \mathbb{N}, .., x_n : \mathbb{N} \vdash x_i : \mathbb{N}$ for $0 < i \leq n$ then f_t is the projection function P_i.
- If f_t is denoted by $x_1 : \mathbb{N}, .., x_n : \mathbb{N} \vdash s(a) : \mathbb{N}$ then f_t is the function $succ \circ f_a$, and by I.H. f_a is primitive recursive.
- If f_t is denoted by $x_1 : \mathbb{N}, .., x_n : \mathbb{N} \vdash R(a, [x, y]b, c) : \mathbb{N}$ then define $g : \mathbb{N}^{n+1} \to \mathbb{N}$ by
  - $g(r_1, ..., r_n, 0) = f_a(r_1, ..., r_n)$
  - $g(r_1, ..., r_n, m + 1) = f_b(r_1, ..., r_n, m, g(r_1, ..., r_n, m))$

  From the induction hypothesis, $f_a(r_1, ..., r_n)$, $f_b(r_1, ..., r_n, m, g(r_1, ..., r_n, m))$ and $f_c(r_1, ..., r_n)$ are primitive recursive.
  - If $f_c(r_1, ..., r_n) = 0$ then $R(a, [x, y]b, c)[\overline{r_1}/x_1, ..., \overline{r_n}/x_n] \equiv$
    $R(a[\overline{r_1}/x_1, ..., \overline{r_n}/x_n], [x, y]b[\overline{r_1}/x_1, ..., \overline{r_n}/x_n], c[\overline{r_1}/x_1, ..., \overline{r_n}/x_n]) \twoheadrightarrow$
    $R(a[\overline{r_1}/x_1, ..., \overline{r_n}/x_n], [x, y]b[\overline{r_1}/x_1, ..., \overline{r_n}/x_n], 0) \rightsquigarrow$
    $a[\overline{r_1}/x_1, ..., \overline{r_n}/x_n] \overset{I.H.}{\twoheadrightarrow} \overline{f_a(r_1, ..., r_n)}$, and $f_a(r_1, ..., r_n) = g(r_1, ..., r_n, 0)$ is primitive recursive.
  - If $f_c(r_1, ..., r_n) = m + 1$ for $m > 0$ then $R(a, [x, y]b, c)[\overline{r_1}/x_1, ..., \overline{r_n}/x_n] \twoheadrightarrow$
    $R(a, [x, y]b, s(\overline{m}))[\overline{r_1}/x_1, ..., \overline{r_n}/x_n] \equiv$
    $R(a[\overline{r_1}/x_1, ..., \overline{r_n}/x_n], [x, y]b[\overline{r_1}/x_1, ..., \overline{r_n}/x_n], s(\overline{m})[\overline{r_1}/x_1, ..., \overline{r_n}/x_n]) \twoheadrightarrow$
    $R(a[\overline{r_1}/x_1, ..., \overline{r_n}/x_n], [x, y]b[\overline{r_1}/x_1, ..., \overline{r_n}/x_n], s(\overline{m})) \rightsquigarrow$
    $b[m/x, R(a, [x, y]b, c)[\overline{r_1}/x_1, ..., \overline{r_n}/x_n]/y, \overline{r_1}/x_1, ..., \overline{r_n}/x_n] \overset{I.H.}{\twoheadrightarrow} \overline{f_b(r_1, ..., r_n, m, g(r_1, ..., r_n, m))}$, and
    $f_b(r_1, ..., r_n, m, g(r_1, ..., r_n, m)) = g(r_1, ..., r_n, m + 1)$ is primitive recursive.

Now we want to show that for every primitive recursive function, there is a term that denotes that function.

- The constant 0 is denoted by $x_1 : \mathbb{N}, .., x_n : \mathbb{N} \vdash 0 : \mathbb{N}$.
- The projection function P_i is denoted by $x_1 : \mathbb{N}, .., x_n : \mathbb{N} \vdash x_i : \mathbb{N}$.
- The successor function is denoted by $x_1 : \mathbb{N}, .., x_n : \mathbb{N} \vdash s(a) : \mathbb{N}$.
- The primitive recursive function (number 5 of Definition 2.1) is denoted by $x_1 : \mathbb{N}, .., x_n : \mathbb{N} \vdash R(a, [x, y]b, c) : \mathbb{N}$.

We need to show that the composition

$$h(x_1, ..., x_n) = f(g_1(x_1, ..., x_n), ..., g_k(x_1, ..., x_n)),$$

which is primitive recursive, is also denoted by a term. Let f be denoted by $y_1 : \mathbb{N}, ..., y_k : \mathbb{N} \vdash t : \mathbb{N}$ while for $0 < i \leq k$, let g_i be denoted by $x_1 : \mathbb{N}, ..., x_n : \mathbb{N} \vdash u_i : \mathbb{N}$. We need to show that h is denoted by the term $x_1 : \mathbb{N}, ..., x_n : \mathbb{N} \vdash t[u_1/y_1, ..., u_k/y_k] : \mathbb{N}$. When substituting the numerals $\overline{r_1}, ..., \overline{r_n}$ we get

$$t[u_1/y_1, ..., u_k/y_k][\overline{r_1}/x_1, ..., \overline{r_n}/x_n] \equiv t[u_1[\overline{r_1}/x_1, ..., \overline{r_n}/x_n]/y_1, ..., u_k[\overline{r_1}/x_1, ..., \overline{r_n}/x_n]/y_k],$$

and since for each u_i denotes g_i, we get that

$$t[u_1[\overline{r_1}/x_1, ..., \overline{r_n}/x_n]/y_1, ..., u_k[\overline{r_1}/x_1, ..., \overline{r_n}/x_n]/y_k] \twoheadrightarrow t[\overline{g_1(r_1, ..., r_n)}/y_1, ..., \overline{g_k(r_1, ..., r_n)}/y_k].$$

Finally, since t denotes f we get that

$$t[\overline{g_1(r_1, ..., r_n)}/y_1, ..., \overline{g_k(r_1, ..., r_n)}/y_k] \twoheadrightarrow \overline{f(g_1(r_1, ..., r_n), ..., g_k(r_1, ..., r_n))} \equiv \overline{h(r_1, ..., r_n)}.$$

□

Corollary 4.15. *The functions denoted by the terms of $TT(\mathbb{N}, \times)$ are exactly the primitive recursive functions.*

Proof. We prove each direction:

- Take any term $x_1 : \mathbb{N}, .., x_n : \mathbb{N} \vdash t : \mathbb{N}$, such that f_t is denoted by t, by Theorem 4.13 we have that f_t is denoted by both $x_1 : \mathbb{N}, ..., x : n : \mathbb{N} \vdash t : \mathbb{N}$ in $TT(\mathbb{N}, \times)$ and $x_1 : \mathbb{N}, ..., x : n : \mathbb{N} \vdash [\![t]\!] : \mathbb{N}$ in $TT(\mathbb{N})$. By Theorem 4.14, f_t is primitive recursive.
- Suppose that f_t is primitive recursive, then from Theorem 4.14 f_t is denoted by the term $x_1 : \mathbb{N}, ..., x : n : \mathbb{N} \vdash t : \mathbb{N}$ in $TT(\mathbb{N})$. Then it is trivial that f_t is denoted by the term $x_1 : \mathbb{N}, ..., x : n : \mathbb{N} \vdash t : \mathbb{N}$ in $TT(\mathbb{N}, \times)$ too.

□

5 Discussion

5.1 Conclusion

The purpose of this thesis was to investigate the conservative relations between type theories. We examined the type theories of natural numbers, $TT(\mathbb{N})$ and $TT(\mathbb{N}, \times)$, and established that $TT(\mathbb{N}, \times)$ is canonically conservative over $TT(\mathbb{N})$. This result implies that $TT(\mathbb{N}, \times)$ is closed conservative and propositionally conservative over $TT(\mathbb{N})$ as well. Because of the similarities between $TT(\mathbb{N})$ and $TT(\mathbb{N}, \times)$, one might initially expect them to be strongly conservative, but the presence of free variables led to stuck terms. This violated the requirement that there should exist a term in $TT(\mathbb{N})$ that is convertible to a corresponding term in $TT(\mathbb{N}, \times)$. This, in turn, led to $TT(\mathbb{N}, \times)$ not being strongly conservative over $TT(\mathbb{N})$. Hence, strong conservativity was not applicable in the conservative relation between $TT(\mathbb{N})$ and $TT(\mathbb{N}, \times)$. However, it still deserves further research, as it might prove useful in other type theories beyond $TT(\mathbb{N})$ and $TT(\mathbb{N}, \times)$.

On the other hand, canonical conservativity is open for further investigation. For instance, there may be some additional details to consider in the definition of canonical conservativity, in order to create a more general definition for conservativity between type theories. It is currently unclear whether canonical conservativity serves as a general definition of a conservative extension, because as we have observed so far, we only know that $TT(\mathbb{N}, \times)$ is canonically conservative over $TT(\mathbb{N})$, but not for other type theories that share similar relations. We would need to expand our research between different type theories, beyond $TT(\mathbb{N})$ and $TT(\mathbb{N}, \times)$, in order to verify if canonical conservativity is applicable to them. As a guide for the investigation of conservative relations, the implications in Figure 4 could be beneficial, since it is better to investigate more than one notion of conservativity.

If a theory is closed conservative over its sub-theory, then it implies propositional conservativity when function types are included in the extended theory. This is because we can express all the free variables as

terms that are bound within lambda terms in the extended theory. However, it remains an open question whether closed conservativity implies propositional conservativity without the use of function types.

Despite canonical conservativity being weaker than strong conservativity, due to the involvement of canonical terms, it demonstrated that the functions denoted by a term in $TT(\mathbb{N}, \times)$ are exactly the primitive recursive functions, from the theorems that we needed to prove canonical conservativity. Without these results, corollary 4.15 would require additional cases, for example, proving that an ordered pair denotes a primitive recursive function. The major point to be made is that from $TT(\mathbb{N}, \times)$ being canonically conservative over $TT(\mathbb{N})$ it gave us that the terms of $TT(\mathbb{N}, \times)$ and $TT(\mathbb{N})$, denote the same functions. This can be valuable as it allows us to define new functions in the extended theory that are also definable in the sub-theory. Working with an extended theory that has additional rules can make the theorem proving process easier for us, as we can use more resources.

If we only tried to prove that $TT(\mathbb{N}, \times)$ is propositionally conservative over $TT(\mathbb{N},)$, then it would not be obvious to prove that all the functions denoted by a term in $TT(\mathbb{N}, \times)$ are exactly the primitive recursive functions, since we have no insight about convertibility between $TT(\mathbb{N})$ and $TT(\mathbb{N}, \times)$. We would have to do a proof induction on Corollary 4.15, as we did in Theorem 4.14, but with more cases.

With canonical conservativity, we enhance our understanding of the terms across the type theories, and since terms represent proofs in type theory, we have a deeper understanding of proofs, as well as programs. Another major point to be made, fulfilling the requirement of canonical conservativity, may grant us highly valuable properties when it comes to transferring theorems and understanding the relationship between two type theories.

By seeking for a general definition of a conservative extension, we strive to enhance our understanding of the foundation of mathematics, as there are type theories that serves as a foundation of mathematics, like Martin-Löf type theory. If a type theory T_1 that serves as a foundation of mathematics has an extended theory T_2 that is canonically conservative over T_1, then we may be able to prove new results in T_2 that is therefore provable in T_1 as well. While proving conservativity, Figure 4 should serve as a guide, as we may get stronger notions of conservativity between the theories, than canonical conservativity.

5.1.1 Contributions and Insights Gained

One significant insight that we have gained from canonical conservativity is that we impose specific conditions for the extension of a theory. For instance, in the presence of a canonically conservative relationship between two type theories, we are limited to terms that denotes functions such as $f(t)$, where t is in its canonical form. For example, consider the proof of Theorem 4.13. We relied on the proof of Theorem 4.7, which in turn proved canonical conservativity between $TT(\mathbb{N})$ and $TT(\mathbb{N}, \times)$. Because of these considerations, we defined what it means for a term to denote a function more precisely (see Definition 4.12). We defined it in such a way, that it substituted all the free variables with numerals, i.e., canonical terms. We did so because $TT(\mathbb{N}, \times)$ is a canonically conservative extension over $TT(\mathbb{N})$, and this helped us to prove Theorem 4.13, which in turn proved Corollary 4.15. A strong conservative relation between two type theories, could imply the existence of additional properties between the theories that could go beyond than simply be able to define the same functions between two theories.

We know that the terms in $TT(\mathbb{N})$ and $TT(\mathbb{N}, \times)$ can express the primitive recursive functions, and the relationship between them is that they are canonically conservative. Primitive recursive functions are considered to be significantly basic, then one might suggest that canonical conservativity is a relatively weak notion. However, canonical conservativity grants us that the theories defines the same functions, which is a valuable property, which in that sense, it is a relatively strong notion. For instance, if one theory can express more than just the primitive recursive functions, while another theory can only express the primitive recursive functions, then these two theories should not be canonically conservative. We cannot conclude that canonical conservativity can be considered a weak definition, because we have only investigated canonical conservativity on two simple type theories, namely $TT(\mathbb{N})$ and $TT(\mathbb{N}, \times)$. If we were about to investigate Gödel's System T [1], and try to prove canonical conservativity for another theory that extends System T, we could then verify if the terms for each theory denote the exact same functions. If that is the case, then we can transfer more functions than just the primitive recursive functions, since System T can express more

than $TT(\mathbb{N})$, for example the Ackermann function. Therefore, our contribution lies on further research on canonical conservativity, to explore its applications and implications in mathematics and computer science.

It is unclear how these definitions of conservative extensions relates back to non-type theories through the propositions as types correspondence. This is because our notions of conservativity are built upon the standard formulation and are extended with convertibility as an additional condition. We want to establish stronger notions of conservativity within type theory, in order to get further insight on our proofs or programs. If one were to establish a conservative extension between two non-type theories and translate it to their type-theoretic equivalence, then investigating conservative extension through these type theories might grant us some further insight from a type-theoretic perspective. For instance, in System T, we have strong connections to arithmetic, and conservative extension has been studied in [1, p. 18] for Peano arithmetic and Heyting arithmetic for some particular sentences. Therefore, it would be valuable to explore if we can find a theory that is canonically conservative over System T. This investigation could yield some interesting results in the extended theory, which would also hold for System T and potentially have implications for arithmetic. And for the same reason as for arithmetic, one may study other foundations of mathematics through their type-theoretic equivalence, in order to gain further insights of conservativity of that perspective.

5.2 Further Work

This thesis opens up for further investigations when studying conservative extensions. We propose four further investigations, where one serves as a more practical approach towards conservative extension. The others are similar investigations as in this thesis but with different type theories and extensions of previous ones. The practical approach is providing the design of a new proof assistant, with conservativity in considerations. While the more theoretical aspects for further work are:

- Type Theories of General recursion, where we investigate general recursive functional programs from their type-theoretic equivalence. We propose this to have a closer relationship to functional programming languages. Having a definition of a conservative extension can be valuable when extending programming languages, so we do not break the existing programming language.
- Extending $TT(\mathbb{N})$, as it could be valuable for further investigation on similar extensions as between $TT(\mathbb{N})$ and $TT(\mathbb{N}, \times)$. We might expect similar results, but there might arise new notions of conservativity through the investigations.
- Type Theories of Set Theory: instead of working with the natural number as a type, it could be worth investigating a different kind of type theory. Canonically conservative and the different notions of conservativity deserves to be put into test for other type theories, instead of those that process natural numbers as a type, and with additional type constructors.

Conservative Relations between Type Theories of General Recursion

$TT(\mathbb{N})$ is one way to translate primitive recursive function into type theory. There is a related topic written by Ana Bove about translating general recursive functional programs into their type-theoretic equivalents [4]. Although the translations are not between type theories, they involve the transformation of functional programs into their type-theoretic equivalents. Even if it does not discuss conservative extensions between type theories, it could still be relevant to this thesis, since it deals with general recursive functional programs. While this thesis is dealing with primitive recursive functions into their type theoretic equivalences.

One could extend a type theory of General recursion and investigate the notion of conservativity between the two of them. Questions to investigate are: are they conservative to the same notions as $TT(\mathbb{N})$ and $TT(\mathbb{N}, \times)$? Do we need another definition of what a conservative extension means and if so, is the new definition of a conservative extension true between $TT(\mathbb{N})$ and $TT(\mathbb{N}, \times)$ as well?

Extending Type Theory of Natural Numbers

One could extend $TT(\mathbb{N}, \times)$ even further to a type theory of natural numbers with pairing and lists, called $TT(\mathbb{N}, \times, [])$. In $TT(\mathbb{N}, \times, [])$ we add a new type constructor that is defining a list, so the types would be:

$$A, B ::= \mathbb{N} \mid A \times B \mid [A]$$

Then the terms for $TT(\mathbb{N}, \times, [])$ are:

$$a, b, c ::= 0 \mid z \mid s(a) \mid R(a, [x, y]b, c) \mid \langle a, b \rangle \mid \pi_1(a) \mid \pi_2(a) \mid [c_1, ..., c_n]$$

We could then use Gödel numbering, which Gödel developed for the proof in his incompleteness theorem [10], which we use in order to encode the list terms and thereafter try to prove conservative relation. We would expect them to be canonically conservative, as we can implement lists by using primitive recursive functions, but there may be other notions of conservativity that would arise during the investigation.

We could extend $TT(\mathbb{N}, \times, [])$ even further and add formulas as a type. In Martin-Löf type theory we have propositions as types, which is the same here, but with formulas as well. The terms would be a proof of a formula then as well. We could then investigate how Gödel numbering can be used in the translation between $TT(\mathbb{N}, \times, [])$ and the type theory of natural numbers with formulas, since Gödel used Gödel numbering in order to encode formulas. Similar with $TT(\mathbb{N}, \times, [])$, we would expect them to be canonically conservative as its strongest notion. Even if we expect them to be canonically conservative, it is worth investigating why that is the case. We could be able to detect early when a type theory is canonically conservative over another. And investigating more simple type theories may grant us further knowledge on why some type theories are canonically conservative.

If we would add function types to $TT(\mathbb{N})$, which corresponds to Gödel's System T [1], say that we call it $TT(\mathbb{N}, \rightarrow)$, then we can express the Ackermann function in the theory. For that case, we can express more that the primitive recursive functions, and we know that $TT(\mathbb{N}, \rightarrow)$ is not canonically conservative over $TT(\mathbb{N})$. Hence, if we were to extend $TT(\mathbb{N}, \rightarrow)$ by introducing additional rules that should be able to define the same functions as in $TT(\mathbb{N}, \rightarrow)$, the resulting theory should be canonically conservative over $TT(\mathbb{N}, \rightarrow)$. By doing this, we can transfer more theorems between these two theories and define more functions.

Type Theories of Set Theory: A Study of Conservative Extension

Instead of investigating type theories of natural numbers, one might consider to investigate type theories of Set Theory instead. For instance, in set theory, an ordered pair $\langle a, b \rangle$ can be defined as $\{\{a\}, \{a, b\}\}$. Additionally, functions can also be defined in set theory [9]. From this, we can have translations between the following type theories:

- A theory with one type 'set'.
- A theory with a type 'set' and a type constructor $\times$ (Cartesian product).
- A theory with a type 'set', type constructor $\times$ and a type constructor $\rightarrow$ (functions).

From those theories, we can use Figure 4, for guidance to investigate the conservative extensions between the type theories of set theory. And by studying a type-theoretic equivalence of set theory, we can gain insights about how set theory could be extended.

Applications in Proof Assistants

Type theory is a great tool to be used as a programming language, mathematical system, and also a logic for reasoning about correctness [15]. One particular application of type theory is in proof assistants. In proof assistants, a user can construct proofs in order to verify theorems in its logic. For instance, in Coq [5], the type theory called the Calculus of Inductive Constructions is implemented in this framework, in order to ensure correctness in our proofs.

When it comes to conservativity, our notions of conservativity can be valuable in the development of the design of a new proof assistant or as new features for existing proof assistants. For example, if we create a proof assistant that incorporates a condition for determining whether two type theories within the system are canonically conservative or one of the other notions of conservative extension, then we could add features that would make it easier for the user's theorem proving process. For instance, if it is the case that the two theories are found to be canonically conservative, we know that every function that is definable in the extended theory is definable in the original theory. And from propositional conservativity, we know that for all theorems in the original theory, if the theorem is provable in the extended theory then it is provable in the original theory. This enables the possibility for transferring theorems from the extended theory to the original theory, and use the extended theory in order to prove theorems. For example, suppose that we have a proof assistant where we can define two new theories T_1 and T_2. We prove that T_2 is canonically conservative over T_1, which give us the information that both T_1 and T_2 can define the same functions. We know that canonical conservativity implies propositional conservativity, then if we know that a theorem P, that originates as a type from T_1, is provable in T_2 then P is provable i T_1. We can now work with T_2 instead of T_1, because all the functions that we define in T_2 is definable in T_1. With such a condition in place, users can work with T_2 to prove theorems, by knowing that these theorems can also be provable in T_1. This new design would make it easier for the theorem proving process in proof assistants, by allowing users to create additional rules for a type theory, without compromising the original theory.

Utilizing the strength of canonical conservativity within a proof assistant can be highly advantageous for theorem proving. It would provide a reliable framework for transferring theorems between type theories that are conservative, and would be beneficial for ensuring correctness for our programs. And by incorporating a stronger notion of conservativity in a proof assistant, we may be able to design the new proof assistant for more features.

To initiate for further investigation, we can start by examining the type theory that Coq is based on. The Calculus of Inductive Constructions (CIC) [5], which is an extension of The Calculus of Constructions(CoC), where CoC was developed by Thierry Coquand [7]. We begin by extending CIC, just like we did for $TT(\mathbb{N})$ with $TT(\mathbb{N}, \times)$, analyze the conservative relations between them, and look for similarities with the notions of conservativity proposed in this thesis. If we can establish a new definition of conservativity between CIC and its extension, then this definition can be integrated in the new design of a new proof assistant that have this notion of conservativity as a condition. Furthermore, as we explore conservative extension within proof assistants, we may encounter scenarios that would suggest further improvements in our notions of conservativity. As practical applications often reveals unexpected outcomes.

References

[1] Jeremy Avigad and Solomon Feferman. Gödel's functional ("dialectica") interpretation. In Sam Buss, editor, *Handbook of Proof Theory*, pages 337–405. Elsevier, 1998.

[2] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Studies in Logic and the Foundations of Mathematics. College Publications, 2012.

[3] G.S. Boolos, J.P. Burgess, and R.C. Jeffrey. *Computability and Logic*. Cambridge University Press, 2007.

[4] Ana Bove. *General Recursion in Type Theory*. PhD thesis, Chalmers University of Technology, Göteborg, Sweden, 2002.

[5] Adam Chlipala. An introduction to programming and proving with dependent types in coq. *Journal of Formalized Reasoning*, 3(2):1–93, Jan. 2010.

[6] Alonzo Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2):56–68, 1940.

[7] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2):95–120, 1988.

[8] Jean-Yves Girard. *Proofs and types*. Cambridge University Press, Cambridge, 1989.

[9] D.C. Goldrei. *Classic Set Theory: For Guided Independent Study*. Chapman & Hall mathematics. Taylor & Francis, 1996.

[10] Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme. *Monatshefte für Mathematik und Physik*, 38(1):173–198, 1931.

[11] Martin Hofmann. Conservativity of equality reflection over intensional type theory. In *Selected Papers from the International Workshop on Types for Proofs and Programs*, TYPES '95, page 153–164, Berlin, Heidelberg, 1995. Springer-Verlag.

[12] J. Lambek and P. J. Scott. *Introduction to Higher Order Categorical Logic*. Cambridge University Press, USA, 1986.

[13] Daniel R. Licata and Robert Harper. Canonicity for 2-dimensional type theory. *SIGPLAN Not.*, 47(1):337–348, jan 2012.

[14] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.

[15] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*. Clarendon Press, USA, 1990.

[16] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[17] António Porto and Armando Matos. Ackermann and the superpowers. *ACM SIGACT News*, 12:90–95, 09 1980.

[18] Jr. Rogers, Hartley. *Theory of recursive functions and effective computability*. 1957, Massachusetts.

[19] K. Rosen. *Discrete Mathematics and Its Applications*. McGraw-Hill Education, 2011.

[20] Peter Selinger. Lecture notes on the lambda calculus. *CoRR*, abs/0804.3434, 2008.

[21] Peter Smith. *An Introduction to Formal Logic*. Cambridge Introductions to Philosophy. Cambridge University Press, 2 edition, 2020.

[22] Martin Sulzmann, Manuel Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System F with type equality coercions. In *ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI'07)*, pages 53–66. ACM, January 2007.

[23] Simon Thompson. *Type Theory and Functional Programming.* Addison Wesley Longman Publishing Co., Inc., USA, 1991.

[24] A. S. Troelstra and H. Schwichtenberg. *Basic Proof Theory.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2 edition, 2000.

[25] T. Xue. Definitional extension in type theory. In *19th International Conference on Types for Proofs and Programs (TYPES 2013)*, volume 26, pages 251–269, 07 2014.

A Implementation of $TT(\mathbb{N})$ in Haskell

Here is an implementation of $TT(\mathbb{N})$ in Haskell, which was used to test the encodings for the basic functions and pairing in $TT(\mathbb{N})$. We used this implementation for the demonstration of using the pairing function in practise, which is illustrated in fig. 5.

```
data Nat = Zero | S Nat
    deriving (Show, Eq)

-- one :: Nat
one = S Zero

-- two :: Nat
two = S one

-- three :: Nat
three = S two

-- four :: Nat
four = S three

-- Reduction Rules for TTNat:

rec :: Nat -> (Nat -> Nat -> Nat) -> Nat -> Nat
rec a b Zero = a
rec a b (S n) = b n (rec a b n)

-- Encodings

-- Addition
add :: Nat -> Nat -> Nat
add a b = rec a (\x y -> S(y)) b

-- Multiplication
mul :: Nat -> Nat -> Nat
mul a b = rec Zero (\x y -> add y a) b

-- Exponential
expN :: Nat -> Nat -> Nat
expN a b = rec one (\x y -> (mul a y) ) b

-- Predecessor
predN :: Nat -> Nat
predN n = rec Zero (\x y -> x) n

-- Truncated subtraction
monus :: Nat -> Nat -> Nat
monus a b = rec a (\x y -> predN y) b

-- Signum functions

-- Signum
```

```
sig :: Nat -> Nat
sig a = rec Zero (\x y -> one) a

-- Reverse Signum
sigNeg :: Nat -> Nat
sigNeg a = rec one (\x y -> Zero) a

-- Truth values
t = one
f = Zero

-- Logical Connective

-- Disjunction
orN :: Nat -> Nat -> Nat
orN a b = rec (sig a) (\x y -> sig b) b

-- Conjunction
andN :: Nat -> Nat -> Nat
andN a b = sigNeg $ orN (sigNeg a) (sigNeg b)

-- Implication
cond :: Nat -> Nat -> Nat
cond a b = orN (sigNeg a) (sig b)

-- Predicates

-- Equality
eq :: Nat -> Nat -> Nat
eq a b = monus one ( add (sig (monus a b) ) (sig (monus b a) ) )

-- Not equal
neq :: Nat -> Nat -> Nat
neq a b = sigNeg (eq a b)

-- If-then-else
ifTE :: Nat -> Nat -> Nat -> Nat
ifTE a b c = rec c (\x y -> b) a

-- Less than
lessThan :: Nat -> Nat -> Nat
lessThan a b = sig (monus b a)

-- Less than or equal
leq :: Nat -> Nat -> Nat
leq a b = orN (lessThan a b) (eq a b)

-- Is even
isEven :: Nat -> Nat
isEven n = rec one (\_ y -> sigNeg y ) n

-- Bounded Existential
```

```
exists :: (Nat -> Nat) -> Nat -> Nat
exists p n = rec (p Zero) (\x y -> ifTE (p (S x )) t y) n

-- Bounded universal
forAll :: (Nat -> Nat) -> Nat -> Nat
forAll p n = rec (p Zero) (\x y -> ifTE (p (S x )) y Zero) n

-- Divides
divides :: Nat -> Nat -> Nat
divides m n = exists (\x -> eq n (mul m x)) n

-- Pairing
pr :: Nat -> Nat -> Nat
pr a b = mul (expN two a) (S(add b b))

-- Projection 1:
p1 :: Nat -> Nat
p1 n = rec Zero (\x y -> ifTE (divides (expN two x) n) x y) n

-- Projection 2:
p2 :: Nat -> Nat
p2 n = rec Zero (\x y -> ifTE (divides (S (add x x) ) n) x y) n

-- Convert to integer
i :: Nat -> Int
i Zero = 0
i (S n) = 1 + i n

-- Convert to numeral
n :: Int -> Nat
n 0 = Zero
n i = S $ n $ pred i
```