UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE MATEMÁTICA

On the Star Combinatory Calculus: formalization and applications to classical logic

Lia Malato

Mestrado em Matemática

Dissertação orientada por:
Professor Doutor Fernando Ferreira
Professor Doutor Alcides Fonseca

2024

"Logic is the foundation of the certainty of all the knowledge we acquire."

– Leonhard Euler

Acknowledgements

It would be foolish of me to attempt to mention everyone who has contributed to this dissertation. However, it would also be foolish of me not to try to lay down the names of those who, whether knowing it or not, have profoundly shaped who I am. To quote José Ortega y Gasset in his work *Meditaciones del Quijote* from 1914:

I am I and my circumstance.

This dissertation is the result of countless interactions I had throughout my life. And this is why I will do my very best to mention those who have impacted my academic journey in a way that often not even words are enough to describe.

Some teachers might have forgotten me by now, but I have not forgotten them. I have to thank the teachers and professors who have actively made me a better person and a better thinker. Ermelinda Carvalho and Carlos Raminhos, I thank you for having shaped my younger self. Because of you I was able to become who I am today. Pedro Luís, Eva Marques, Manuela Pereira, Anabela Tomé and Prof. Dr. Cristina Luís, each one of you has taught me critical thinking skills that are carried along in every thought that goes through my mind. Laurence Scassellati and Claudia Losito you both kept me sane throughout my first university years by so lovingly sharing your artistic teachings with me. I thank you for all the trust you have placed in me during these years.

Although I was already convinced that mathematics was the path I wanted to follow, some professors made me rediscover why this is the case by gaving me a new perspective on how to think about various mathematical topics. Thank you Dr. Simon Campese, Prof. Dr. Norbert Poncin and, more recently, Prof. Dr. Pedro Resende and Prof. Dr. Kevin Buzzard. Moreover, I thank Prof. Dr. Ana Moura for her feedback, Prof. Dr. Jorge Buescu for making me truly believe I could finish this dissertation and my supervisors Prof. Dr. Fernando Ferreira and Prof. Dr. Alcides Fonseca for all their dedicated time. I would also like to express my profound gratitude to the research unit LASIGE from the Department of Informatics of the University of Lisbon that made it possible to develop the present dissertation as a bridge between the Mathematics and Informatics Departments of the University of Lisbon.

Finally, a special token of gratitude to the three most important people in my life. I thank Graça da Silva for the love she gave me while I was growing up. And last, but certainly not least, I would like to thank Francisco Albuquerque Picado and Isabel Leite for their never-ending support in every aspect of my life.

These acknowledgments are a written recognition to those who have made it possible for me to be able to look back at every single step of the writing journey with happiness and fulfillment. This work is dedicated to all those I have mentioned and to all the mathematicians who give their best to turn mathematics more approachable and enjoyable.

Abstract

We begin by developing a new typed combinatory calculus called *Star Combinary Calculus* which involves both the well-known finite types and the more recent star type. With this calculus at hand we will construct a functional interpretation that translates a typed extension of classical first-order logic into itself. After having introduced the necessary context, we will prove the soundness of this interpretation together with other relevant results. We will then state and prove a new generalization of Herbrand's Theorem that involves a bounded form of the Axiom of Choice, an axiom that only unveils itself in the theorem when it is stated in this newer context.

Once the Star Combinatory Calculus is fully discussed, we lay our attention onto proof assistants, more in particular onto the proof assistant Lean. We will give a general overview of this proof assistant to then, as a final goal, provide an original formalization of some of the results associated with the Star Combinatorial Calculus in Lean.

Keywords: Star Combinatory Calculus • functional interpretations • first-order logic and higher-order logic in all finite types • Herbrand's Theorem • proof assistant Lean

Resumo

O objetivo da presente dissertação é definir uma interpretação funcional para a lógica clássica em tipos finitos e, com isso, dar uma nova visão ao Teorema de Herbrand num âmbito mais geral.

Começamos por desenvolver um novo cálculo combinatório tipado chamado *Cálculo de Combinadores Estrela*, que envolve tanto os bem-conhecidos tipos finitos (o tipo base e o tipo função ou seta) como o mais recente *tipo estrela*. Enquanto o tipo seta representa funcionais, o tipo estrela representa conjuntos finitos e não-vazios. Baseando-nos numa linguagem de lógica clássica de primeira ordem com pelo menos uma constante, vamos desenvolver uma extensão tipada desta mesma para o contexto de lógica clássica de ordem superior e de tipos finitos. Neste sentido, cada objeto desta nova linguagem terá associado um tipo finito. Esta nova linguagem vai herdar as constantes da linguagem da lógica de primeira ordem, às quais acrescentamos novas constantes exclusivas da nova linguagem. Estas novas constantes consistem nos chamados combinadores e nas chamadas constantes estrela. Por outro lado, a nova linguagem terá igualmente símbolos primitivos adicionais como o quantificador universal limitado, o símbolo da igualdade e o símbolo do pertence. Através da existência de quantificadores limitados e ilimitados, definimos a classe de *fórmulas base* como sendo as fórmulas sem quantificadores ilimitados. Esta classe de fórmulas surgirá em múltiplos resultados.

Uma vez a síntaxe da linguagem definida, apresentamos uma teoria adequada ao estudo do Cálculo de Combinadores Estrela, correspondendo esta a uma extensão tipada da lógica clássica de primeira ordem. Usamos um cálculo completo para a lógica clássica de primeira ordem desenvolvido por Shoenfield, tal como axiomas que regem os combinadores, a igualdade, as constantes estrela e os quantificadores limitados. É de referir que a maioria dos axiomas da axiomática desta teoria são fechos universais de fórmulas base. Esta característica tornar-se-á útil ao longo da dissertação. De seguida, apresentamos uma versão tipada do Axioma da Escolha chamado *Axioma da Escolha Limitado*. Embora fora da axiomática original, este axioma terá um papel fundamental no contexto da presente dissertação.

Com a sintaxe e a axiomática apresentadas, estudamos o *Teorema da Completude Combinatorial* que nos permite representar certas funções através de termos da linguagem. Com estes termos, é-nos possível introduzir a notação lambda como símbolo definido que, por sua vez, nos dá acesso a lambda abstrações. Em poucas palavras, o Teorema da Completude Combinatorial permite definir termos a partir dos combinadores da linguagem, termos estes que neste contexto desempenham o papel do operador usual da substituição.

A partir deste estudo, continuamos a desenvolver o Cálculo de Combinadores Estrela analisando as suas conversões de termos. Apresentamos as definições necessárias para podermos, entre outros, discutir resultados relacionados com as propriedades de *confluência* e *normalização* deste cálculo. Após concluirmos que cada termo do Cálculo de Combinadores Estrela pode ser reduzido a uma forma normal e que esta forma normal é única, estudamos certas características associadas à estrutura de termos normais. Para tal, definimos o conceito de termos *set-like* com o objetivo de apresentarmos uma estrutura geral para termos normais e fechados.

Com os fundamentos do Cálculo de Combinadores Estrela definidos e estudados, passamos a debruçar-nos sobre interpretações funcionais. Tendo por base a interpretação funcional de Shoenfield, originalmente definida no âmbito da aritmética, passamos a estudar uma versão herbrandizada da interpretação funcional de Shoenfield, desta vez no âmbito da lógica clássica em tipos finitos o que permite traduzir a linguagem do Cálculo de Combinadores Estrela em si mesma. A interpretação funcional é dita *herbrandizada* porque as testemunhas das interpretações de fórmulas são acumuladas em conjuntos finitos.

Após a apresentação do contexto necessário, focamo-nos em resultados associados a esta interpretação, resultados estes que desvendam o papel desempenhado pelo Axioma da Escolha Limitado no contexto da lógica clássica de ordem superior em todos os tipos finitos com o tipo estrela. Começamos pelo *Teorema da Correção*. Este teorema central torna possível a extração de informação sob forma de termos, dado o caráter construtivista da sua demonstração. É de notar que a construção destes termos faz referência constante ao Teorema da Completude Combinatorial no sentido em que usamos lambda abstrações para os definir.

A demonstração do Teorema da Correção requer um outro resultado: o *Teorema da Monotonia*. Este último é um resultado que afirma que, tendo a interpretação de uma fórmula, a matriz da interpretação é válida para a variável existencial da interpretação, mas também para qualquer outra que a contenha. O Teorema da Monotonia está estritamente relacionado com o caráter herbrandizado da interpretação funcional em questão.

Juntamente com o Teorema da Correção, torna-se pertinente estudarmos os chamados *princípios característicos* da nossa interpretação funcional de Shoenfield. Estes são os princípios que, embora figurem na hipótese do Teorema da Correção, desaparecem na conclusão. Neste sentido, demonstramos que a versão herbrandizada da interpretação funcional de Shoenfield tem apenas um princípio característico, nomeadamente o Axioma da Escolha Limitado. Depois de termos demonstrado o Teorema da Correção e discutido o princípio característico da interpretação funcional em causa, estudamos em que condições uma fórmula é equivalente à sua tradução. O *Teorema da Caracterização* proporciona a resposta: uma fórmula e a sua interpretação são sempre equivalentes na teoria associada ao Cálculo de Combinadores Estrela quando esta é reforçada com o Axioma da Escolha Limitado.

Com base em tudo o que foi exposto anteriormente, foi possível então desenvolver um primeiro contributo original da presente dissertação. O contributo em questão prende-se com uma aplicação do Cálculo de Combinadores Estrela, nomeadamente uma nova generalização do bem-conhecido *Teorema de Herbrand*. Após enunciarmos o teorema para lógica clássica de primeira ordem, apresentamos uma série de resultados que permitem generalizar pela primeira vez o teorema em questão no âmbito da nossa extensão da lógica clássica. O enunciado generalizado envolve o Axioma da Escolha Limitado, axioma este que apenas fica visível quando o teorema é enunciado em ordem superior. Após termos demonstrado a versão generalizada do Teorema de Herbrand, apresentamos igualmente uma nova demonstração baseada num argumento semântico, de modo a recuperar o resultado original a partir do resultado generalizado.

Com a parte matemática da dissertação concluída, fazemos incidir a nossa atenção no segundo contributo original da presente dissertação, designadamente a formalização de determinados resultados e noções associados ao Cálculo de Combinadores Estrela no assistente de prova Lean. Começamos por descrever o que é um assistente de prova, juntamente com vantagens e desvantagens do respetivo uso no âmbito da investigação matemática. De seguida, fazemos uma breve introdução ao conceito de *Proposições como Tipos* (do inglês *Propositions as Types*), a base de funcionamento das provas matemáticas em Lean, e apresentamos as táticas (do inglês *tactics*) mais importantes para a leitura e a compreensão do código da formalização.

Após termos fornecido uma visão panorâmica do assistente de prova Lean, passamos a descrever partes da nossa formalização original. Discutimos a formalização de conceitos básicos como os tipos finitos, os termos e as fórmulas da linguagem, para depois apresentarmos a formalização do enunciado e da demonstração do Teorema da Completude Combinatorial. De seguida, prosseguimos com a apresentação da formalização da versão herbrandizada da interpretação

funcional de Shoenfield. Concluímos a segunda parte da dissertação mencionando as principais diferenças entre os enunciados matemáticos e os enunciados em Lean e explicamos como o uso de Lean como assistente de prova permitiu detetar pequenas gralhas na escrita do texto matemático. Como nota final, fazemos o ponto da situação no que diz respeito ao uso de assistentes de prova pela comunidade matemática e mencionamos a importância destes no futuro desenvolvimento desta área.

Palavras-chave: Cálculo de Combinadores Estrela • interpretações funcionais • lógica de primeira ordem e lógica de ordem superior em todos os tipos finitos • Teorema de Herbrand • assistente de prova Lean

Table of contents

List of figures

List of tables

List of symbols

Axioms, theories and languages

AxE_1, AxE_2	axioms of equality
AxU	axiom of the bounded universal quantifiers
AxC_1, AxC_2	axioms of the combinators
AxP_1, AxP_2	primary axioms of the star constants
AxS_1 - AxS_4	secondary axioms of the star constants
bAC^ω_*	Bounded Axiom of Choice (for base formulas)
PL	predicate logic, first-order logic
PL^ω_*	Star Combinatory Predicate Logic with finite types including the star type
$\mathscr{L}$	first-order logic language with at least one constant symbol, without equality
$\mathscr{L}^\omega_*$	Star Combinatory Language (typed language based on the language $\mathscr{L}$)

Symbols

σ, τ, ρ	finite types
a, b, c, d, e, f, g	variables
u, v, w, x, y, z	variables
Φ, Ψ	variables
p, q, r, s, t	terms
A, B, C	formulas
A_{qf}	formula A is quantifier-free
A_{at}	formula A is atomic
A_b	formula A is a base formula
A^{SH}	Shoenfield's functional interpretation of formula A
A_{SH}	matrix or lower SH-formula of formula A

Syntactic aspects

$\neg$	negation (primitive)
$\vee$	disjunction (primitive)
$\wedge$	conjunction (defined)
$\rightarrow$	implication (defined)
$\leftrightarrow$	equivalence (defined)
$\forall$	universal quantifier (primitive)
$\forall x \in t$	bounded universal quantifier (primitive)
$\exists$	existential quantifier (defined)

$\exists x \in t$	bounded existential quantifier (defined)
$=_\sigma$	equality between terms of type σ (primitive)
$\in_\sigma$	membership of type σ (primitive)
$\subseteq_\sigma$	inclusion of type σ (defined)
$\Sigma_{\sigma,\tau,\rho}$	sigma combinator of type $(\sigma \to (\tau \to \rho)) \to ((\sigma \to \tau) \to (\sigma \to \rho))$
$\Pi_{\sigma,\tau}$	pi combinator of type $\sigma \to (\tau \to \sigma)$
$\mathfrak{s}_\sigma$	singleton of type σ
$\cup_\sigma$	binary union of type $\sigma^* \to (\sigma^* \to \sigma^*)$
$\bigcup_{\sigma,\tau}$	indexed union of type $\sigma^* \to ((\sigma \to \tau^*) \to \tau^*)$
$=$	syntactic equality between terms
$\equiv$	syntactic equality between formulas
$:=$	definitional syntactic equality between terms
$:\equiv$	definitional syntactic equality between formulas
$\rightsquigarrow$	conversion symbol
$\succ_1$	one-step term reduction
$\succ$	transitive closure of $\succ_1$
$\succeq$	term reduction
$\xrightarrow{1}$	directed graph representation of one-step term reductions
$\xrightarrow{\#}$	directed graph representation of term reductions

Types

ω	finite types
$*$	star type
G	ground type
$\to$	function type

Tuple notation

$\underline{t}$	tuple of terms $t_1, ..., t_n$
$\underline{\sigma}$	tuple of types $\sigma_1, ..., \sigma_n$
$\forall \underline{x}$	notation for $\forall x_1$ ... $\forall x_n$ where $\underline{x}$ is the tuple of variables $x_1, ..., x_n$
$\forall \underline{x} \in \underline{t}$	notation for $\forall x_1 \in t_1$ ... $\forall x_n \in t_n$ where $\underline{x}$ is the tuple of variables $x_1, ..., x_n$ and $\underline{t}$ is the tuple of terms $t_1, ..., t_n$ in which $\underline{x}$ does not occur
$\exists \underline{x}$	notation for $\exists x_1$ ... $\exists x_n$ where $\underline{x}$ is the tuple of variables $x_1, ..., x_n$
$\exists \underline{x} \in \underline{t}$	notation for $\exists x_1 \in t_1$ ... $\exists x_n \in t_n$ where $\underline{x}$ is the tuple of variables $x_1, ..., x_n$ and $\underline{t}$ is the tuple of terms $t_1, ..., t_n$ in which $\underline{x}$ does not occur
$\underline{t} =_\sigma \underline{q}$	notation for $t_i =_\sigma q_i$ for all $1 \leq i \leq n$, where $\underline{t}$ is the tuple of terms $t_1, ..., t_n$ and $\underline{q}$ is the tuple of terms $q_1, ..., q_n$

$\underline{t} \in_\sigma \underline{q}$	notation for $t_i \in_\sigma q_i$ for all $1 \leq i \leq n$, where $\underline{t}$ is the tuple of terms $t_1, ..., t_n$ and $\underline{q}$ is the tuple of terms $q_1, ..., q_n$

Other symbols

SH	herbrandized version of Shoenfield's functional interpretation
tq	application of the term t to the term q
$t(x)$	term t in which the variable x may occur
$A(t)$	formula A in which the term t may occur
$t[q/x]$	substitution of the variable x by the term q in the term t
$A[q/x]$	substitution of the variable x by the term q in the formula A
$\lambda x.t$	lambda abstraction of the term t relative to the variable x
$\Gamma \vdash A$	formula A is derivable from the set of formulas Γ
$\frac{A}{B}$	inference rule where the conclusion B is inferred from the premise A
$A \Rightarrow B$	other notation for an inference rule where the conclusion B is inferred from the premise A

Introduction

What does it mean to *compute*?

When thinking of computations, our minds might wander off to calculations or algorithms. But what exactly does this concept entail from a mathematical point of view? For many centuries, mathematicians strove to find a definition for this concept but without reaching a satisfying conclusion. That was until the 20th century.

It all started with Alonzo Church... and Kurt Gödel and Alan Turing. Just like Philip Wadler remarked in his paper *Propositions as Types* [40]:

> “Like buses: you wait two thousand years for a definition of ‘effectively calculable’, and then three come along at once. The three were lambda calculus, published 1936 by Alonzo Church, recursive functions, proposed by Gödel at lectures in Princeton in 1934 and published in 1936 by Stephen Kleene, and Turing machines, published in 1937 by Alan Turing.”

The 20th century was the Holy Grail on the quest to define the notion of *computability*. It not only made it possible to formalize this notion, but it built a now everlasting bridge between mathematics and computer science. Our goal with the present dissertation is to follow these footsteps and to further strengthen this bridge. On the side of mathematics we will study the *Star Combinatory Calculus* and its applications to classical logic, where we will see a new and generalized formulation of Herbrand's Theorem through the use of a hebrandized version of Shoenfield's functional interpretation. On the side of computer science, we will formalize certain concepts associated with the Star Combinatory Calculus in a proof assistant called *Lean*.

To understand the upcoming sections, we have to talk about Kurt Gödel and how he shattered David Hilbert's dreams in the 20th century. And to understand what these dreams were, we need to go back to the beginning of the 20th century, when Bertrand Russell published his famous *Russell's Paradox* in 1901. Like many others, Hilbert became concerned about the possibility of having such paradoxes in the seemingly reliable field of mathematics. As a response, in the 1920's, Hilbert created his program called *Hilbert's Program* with the goal of turning mathematics into a fundamentally solid field where contradictions would no longer arise. Hilbert's program had three main focuses: *formalizing* mathematics using precise formal systems, proving its *consistency* and all of this by using the so-called *finitistic* reasoning, that is, avoiding concepts such as infinity and making consistency proofs as simple as possible. The program created momentum inside the mathematical community, but everything changed when Gödel published his *Incompleteness Theorems* in 1931. Suddenly, it was no longer possible to prove the consistency of all of mathematics. Hilbert's agenda had failed.

In reaction to this, and to rectify the damage Gödel had caused to Hilbert's Program, Gödel started investigating under which conditions it would still be possible to prove the consistency of certain parts of mathematics, even if it had to be done in a more limited context. Gödel's solution was his *Dialectica functional interpretation*, published in 1958 in a journal also called Dialectica (the name of the functional intepretation was inherited by the name of the journal where it was first published). Such an interpretation made it possible to reformulate statements from one theory to another and to reduce the consistency of the more complex theory to the consistency of the less complex one. Gödel's goal was to prove the consistency of Heyting Arithmetic HA, the intuitionistic counterpart to Peano Arithmetic PA, by reducing it to the consistency of another theory T he then developed. In this new theory T, which in very simple terms can be viewed as a quantifier-free version of HA, Gödel continued Hilbert's trend of relying on *finitary* reasoning by proposing an extension of it in his article *On a hitherto* *unutilized extension of the finitary standpoint* [25] (English translation of the original German article [26]). He focused on introducing concepts related to computability and constructiveness, two important notions when dealing with intuitionistic theories. With this, Gödel was able to reduce the consistency of HA to the consistency of T. But in the end, he went even further: he reduced the consistency of PA to the consistency of HA through the use of his so-called *double negation interpretation* and, together with his Dialectica interpretation that reduced the consistency of HA to the consistency of T, he was even able to reduce the consistency of PA to the consistency of T.

Based on Gödel's work, mathematicians have created numerous functional interpretations for other theories. Through these developments, it is now known that functional interpretations have a significant amount of applications related to computational constructs and proof mining [2, 10]. In the present dissertation, we will explore a new application of functional interpretations: we will see and prove how the well-known Herbrand's Theorem for classical logic is a direct consequence of the Soundness Theorem of a herbrandized version of Shoenfield's functional interpretation.

There are two other major consequences of the works of Gödel, Church and Turing that are worth discussing in the context of the present dissertation. One was the development of the concept *Programs as Proofs* and the other was the creation of *proof assistants.*

Programs as Proofs represents a direct link between computational programs and mathematical proofs. In simple terms, it states that constructing a program is essentially the same as proving a mathematical proposition. Hence, it becomes possible to verify the correctness of an algorithm by proving that certain statements are true. Reversely, it becomes possible to prove results through the use of programs. This paradigm is the basis of how proofs are done in a proof assistant and will be described in later sections.

Proof assistants can be described as software to construct and verify mathematical proofs on a computer. By introducing mathematical definitions and statements into the software using specific commands, the software is then able to assist the user with a proof. When this is successful, it is said that the definitions and results have been *formalized* in the proof assistant. It is important to notice that nowadays, a proof assistant is not yet capable of constructing a whole proof on its own, it merely checks the correctness of every step of a proof and each step is given by the user. A proof assistant still heavily relies on human input before being able to assist with the proof. Well-known proof assistants include Agda and Coq. However, we will use the more recent proof assistant Lean and, just like Hilbert had wished, we will formalize relevant definitions and results of the dissertation.

Structure of the dissertation

We will now describe the structure and main points of the dissertation. Since the content of the dissertation is extensive, we provide a thorough discussion on its structure. The present dissertation has two main focuses. First, that of describing the Star Combinatory Calculus (Sections 1 to 3). Second, that of formalizing some of the main aspects associated with this new calculus in the proof assistant Lean (Section 4). The dissertation is organized as follows:

Section 1. The first section focuses on laying the ground layers of the rest of the document. To do so, we begin by describing the language on which the rest of the document is based. Initially, we consider a first-order language $\mathscr{L}$ to which we add new features to define the extended new language $\mathscr{L}^\omega_*$. One of said features is that the language $\mathscr{L}^\omega_*$ will be a *typed* language. In other words, a language of *finite types*. This means that to every term in $\mathscr{L}_*^\omega$ we will associate an information about it and this information will be called a *finite type*. We will introduce the usual finite types together with the more recent *star type*. This process is described in Section 1.1.

After having defined the finite types, we continue to describe the Star Combinatory Calculus in Section 1.2. We describe the syntax of $\mathscr{L}_*^\omega$ in Section 1.2.1 to then define a corresponding typed theory PL_*^ω in which we will consistently work in throughout the document. To do so, we introduce a set of axiom schemas and inference rules in Section 1.2.2. We present a suitable and complete logical calculus for the axioms of classical logic due to Shoenfield to which we add other relevant axioms to deal with the new aspects of $\mathscr{L}_*^\omega$. We then discuss a typed version of the well-known Axiom of Choice in Section 1.2.3. This axiom will be denoted bAC_*^ω (for *Bounded Axiom of Choice*) and, even though it is not part of the axiomatic of PL_*^ω, it will be of crucial importance in later sections.

From here on, we discuss important structural results about PL_*^ω. In Section 1.2.4, we discuss how to obtain lambda abstractions in PL_*^ω from a result called *Combinatorial Completeness*. In Section 1.2.4, we see how PL_*^ω can be viewed as a so called *term rewriting system* when we add certain rules to the syntax and axiomatic of PL_*^ω. Then, in Section 1.2.5, we reference and discuss results such as the *Strong Normalization Theorem* and the *Church-Rosser Theorem* for the Star Combinatory Calculus. Finally, in Section 1.2.6 we provide a characterization of closed and normal terms as a consequence of the previous results by means of set-theoretical notation.

Section 2. This section is dedicated to interpreting the theory PL_*^ω into itself. This means that we will define a map from the formulas of PL_*^ω to formulas of the same theory in such a way that from a formula A we obtain a new formula A^{SH} of the form $\forall x \exists y\ A_{SH}(x, y)$, where x and y are possibly empty tuples of variables and A_{SH} is a formula where unbounded quantifiers such as $\forall$ and $\exists$ do not appear. Such a map is an example of a functional interpretation which we will define in Section 2.2 and call a *herbrandized* Shoenfield functional interpretation. We will see that this interpretation is herbrandized in the sense that it will be possible to accumulate witnesses of the existential quantifications $\exists y$ into finite and non-empty sets.

Section 3. With the fundamental definitions at hand, the purpose of this section is to state and prove strong results that are natural consequences of the herbrandized Shoenfield functional interpretation. In Section 3.1 we will prove the *Soundness Theorem*. This theorem states that if a formula A is provable in PL_*^ω strengthened with bAC_*^ω, then there exist terms that witness the existential quantifiers of the interpretation of A. As will be seen, it will be possible to accumulate these witnesses into finite and non-empty sets. To prove the Soundness Theorem, we will state and prove the *Monotonicity Lemma*. This lemma states that if a formula holds for a certain term, then it will also hold for any bigger term that contains it.

From here on, we will unveil the role of the Bounded Axiom of Choice bAC_*^ω. In Section 3.2, we will focus on the *Characterization Theorem*. In short, this theorem answers the question:

What is the relationship between a formula and its interpretation?

and gives the condition under which a formula is always equivalent to its interpretation. In Section 3.2.2 we discuss that bAC_*^ω is the only axiom we need to add to the axiomatic of PL_*^ω for a formula to always be equivalent to its interpretation. We will also see that, although bAC_*^ω is in the premises of the Soundness Theorem, the axiom vanishes in the conclusion of the theorem. Such a principle is called a *characteristic principle*.

Finally, Section 3.2.3 delves into the relationship between the Soundness Theorem and Herbrand's Theorem from first-order logic. We will state and prove an original generalized version of Herbrand's Theorem in PL^{ω}_{*} and show how to retrieve the original theorem in first-order logic through a new proof. Surprisingly enough and contrary to the first-order logic version, the generalized version will give a special role to $\mathsf{bAC}^{\omega}_{*}$.

This concludes the first part of the dissertation, the one dedicated to the mathematical study of the Star Combinatory Calculus and the herbrandized Shoenfield functional interpretation.

Section 4. The last section constitutes the second part of the dissertation, which is dedicated to the proof assistant Lean. In this section, we will give a brief introduction to Lean and explain how this proof assistant allows us to formalize certain aspects associated with the Star Combinatory Calculus and the herbrandized Shoenfield functional interpretation. We will then provide and explain some relevant snippets of the code which are original contributions. Moreover, as is common practice when formalizing mathematics in Lean, the whole code will be publicly available on Github and can be viewed and downloaded through the link in Section 4.

Final considerations

Having explained the structure of the dissertation, we now make some final considerations. First, being a dissertation that unites mathematics and computer science, it was of utmost importance to create a text that made it possible for enthusiasts of both worlds to be able to read and understand it. Hence, contrary to what is found in most articles and works on more advanced topics on mathematical logic, we chose to include as many examples and explanations as needed to make the reading for both target audiences as smooth and comprehensible as possible. Second, to make a clear distinction between examples, remarks and the rest of the text, we chose to end examples and remarks with the symbol $\boxtimes$.

We conclude the introduction with a quote from Scottish mathematician Eric Temple Bell's work *Queen of the Sciences* (1931):

"*Obvious* is the most dangerous word in mathematics."

And it is precisely following this quote that the present dissertation aims to be as self-contained as possible, with a sufficient amount of examples and explanations, to send the reader off to a pleasant and insightful adventure through the Star Combinatory Calculus, the herbrandized Shoenfield functional interpretation and their formalization in Lean.

1 Finite types and the Star Combinatory Calculus

The present dissertation focuses on the *Star Combinatory Calculus* together with its finite types and on the consequences of associating a functional interpretation to its underlying language and theory. Throughout this first section we will define the Star Combinatory Calculus by going through the corresponding syntax and axiomatic to then study important results that hold for this calculus. To achieve this goal it will be necessary to first discuss the notion of *finite types*, since the overall discussion on the Star Combinatory Calculus is also a discussion on types.

1.1 Finite types

When thinking about the numbers $1, 2, 3$, most people would think of the natural numbers. This thought could be represented as $1 \in \mathbb{N}, 2 \in \mathbb{N}, 3 \in \mathbb{N}$ or even $\{1, 2, 3\} \subseteq \mathbb{N}$. Were we to think of the integers instead of the naturals, then we could, for example, write $2 \in \mathbb{Z}$. In both cases we chose to associate an information to the numbers: in this case, the set they belong to. The goal is to associate to each term of a language a certain information, and this information is called a *type*. A language where all terms have an associated type is called a *typed language* and a theory associated to a typed language is called a *typed theory*.

To formalize the above example, consider the language of arithmetic which we will denote by $\mathscr{L}_{\text{nat}}$. This language consists of one constant 0, two binary relation symbols $=$ and $<$, one unary function symbol *succ* (representing the successor of a number) and two binary function symbols $+$ and $\cdot$. If we want to turn the language $\mathscr{L}_{\text{nat}}$ into a typed language, then, in particular, every natural number will be attributed a type. This type, the type of the natural numbers, will be denoted by N and the typed language will be denoted by $\mathscr{L}^{\omega}_{\text{nat}}$. With this is mind, while in set theory we would write $2 \in \mathbb{N}$, in a typed theory we would write $2 : N$ or 2^N, as to indicate that *the term* 2 *has type* N. Similarly, while in the standard set-theoretic interpretation we would write $succ : \mathbb{N} \to \mathbb{N}$ for the successor function symbol, in $\mathscr{L}^{\omega}_{\text{nat}}$ the successor function symbol would be typed as $succ : N \to N$.

Interestingly enough, while for the function symbols we have $+, \cdot : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, in the present dissertation their corresponding types will not be given by $N \times N \to N$, but rather by $N \to (N \to N)$. The reason why $+$ and $\cdot$ have this type and what the types of the relational symbols $=$ and $<$ are, will be explained in Section 1.2. As can be seen, the types of *succ*, $+$ and $\cdot$ were all constructed from N and the arrow $\to$. In fact, every type in $\mathscr{L}^{\omega}_{\text{nat}}$ is constructed from them. A type such as N is therefore called a *ground type* and $\to$ is often called a *type constructor*.

Although the language of arithmetic is useful to illustrate these concepts, it is not the language this dissertation focuses on. Instead, we will shift our attention to first-order logic. In this subsection we will introduce new types for this different framework and in Section 1.2 we will define a new language $\mathscr{L}^{\omega}_{*}$ based on a first-order language $\mathscr{L}$. While for $\mathscr{L}_{\text{nat}}$ types were constructed based on N and $\to$, for this new language $\mathscr{L}^{\omega}_{*}$ we will instead require a ground type denoted G and not one, but two type constructors $\to$ and $*$.

Definition 1.1. The *finite types* are defined inductively as follows:

a) The *ground type* G is a type.
b) If σ and τ are types, then the *function type* $\sigma \to \tau$ is a type.
c) If σ is a type, then the *star type* σ^* is a type.

Some authors also use the notation $\tau\sigma$ or $\tau(\sigma)$ for the type $\sigma \to \tau$ (notice that these notations invert the order of the types). For simplicity, we will refer to finite types as *types*. In short, finiteness here refers to the idea that every type must be of finite length. A type such as $G \to (G \to G)^* \to G \to ...$ would not be admissible.

The intended meaning of these types is as follows. If σ and τ are types, then the function type $\sigma \to \tau$ represents the type of functions which takes arguments of type σ and returns elements of type τ. The star type σ^* represents the type of finite and non-empty subsets of elements of type σ.

Example 1.1. Consider again the typed language $\mathscr{L}^\omega_{\text{nat}}$ of arithmetics. If $\mathscr{L}^\omega_{\text{nat}}$ also had the star type from Definition 1.1, then the type of a subset such as $\{1,2,3\}$ would be N^* (as $1 : N$, $2 : N$, $3 : N$ and $\{1,2,3\}$ is a non-empty and finite subset of elements of N). More interesting examples are $\{\{1,2\},\{3\}\} : (N^*)^*$ and $\{+,\cdot\} : (N \to (N \to N))^*$. ⊠

We now consider examples of types in the more general framework of Definition 1.1.

Example 1.2. *(Examples of finite types)*

1. G^* is a type. It represents the type of finite non-empty subsets of elements of type G.
2. $G \to G$ is a type. It represents the type of functions which takes arguments of type G and returns elements of type G.
3. $G \to (G \to G)$ and $(G \to G) \to G$ are both (different) types. The first represents the type of functions which takes arguments of type G and returns functions of type $G \to G$. The latter is the type of functions which take functions of type $G \to G$ as arguments and returns elements of type G.

Let's now consider more elaborated types.

4. $(G \to G) \to (G \to (G \to G))$ is a type.
5. If σ and τ are types, then $\sigma \to ((\sigma^* \to \tau) \to \tau)$ and $(\sigma^* \to \tau)^*$ are both types.

For example, the type $(\sigma^* \to \tau)^*$ represents the type of the finite non-empty subsets of elements of type $\sigma^* \to \tau$, which in turn is the type of the functions that take arguments of the type of finite subsets of elements of type σ and returns elements of type τ. ⊠

As can be seen when writing longer types, the extensive use of brackets is not very convenient. Thus, we adopt the following convention used by most authors.

Notation 1.1. *(Type notation)* Types are written *associated to the right.* Hence, if $\sigma_1, ..., \sigma_n$ are types, instead of writing $\sigma_1 \to (\sigma_2 \to (... \to \sigma_n)...)$, we will write $\sigma_1 \to \sigma_2 \to ... \to \sigma_n$. Using this convention, the types in items 3., 4. and 5. in the above example can be written, respectively, as follows:

3'. The type $G \to (G \to G)$ will be written $G \to G \to G$ while the type $(G \to G) \to G$ remains unchanged.
4'. The type $(G \to G) \to (G \to (G \to G))$ is written $(G \to G) \to G \to G \to G$.
5'. The type $\sigma \to ((\sigma^* \to \tau) \to \tau)$ is written $\sigma \to (\sigma^* \to \tau) \to \tau$ while $(\sigma^* \to \tau)^*$ remains unchanged. ⊠

Next, we consider a notation that will be useful when considering tuples of types such as $\sigma_1, ..., \sigma_n$, where σ_i is a type for $i \in \{1, ..., n\}$. These are called *type tuples* and are often abbreviated by a type with an underscore such as $\underline{\sigma}$.

Notation 1.2. Let $k, n \in \mathbb{N}_{\geq 0}$. Let $\underline{\sigma} = \sigma_1, ..., \sigma_n$ and $\underline{\tau} = \tau_1, ..., \tau_k$ be type tuples. Then $\underline{\sigma} \to \underline{\tau}$ denotes the type tuple $\sigma_1 \to ... \to \sigma_n \to \tau_1, ..., \sigma_1 \to ... \to \sigma_n \to \tau_k$. For $n = 0$, $\underline{\sigma} \to \underline{\tau}$ is $\underline{\tau}$ and, for $k = 0$, $\underline{\sigma} \to \underline{\tau}$ is empty. ⊠

Example 1.3. Let $\underline{\sigma} = G^*, G \to G$ and $\underline{\tau} = G, (G \to G)^*$ be type tuples. Then $\underline{\sigma} \to \underline{\tau}$ denotes the type tuple $G^* \to (G \to G) \to G, G^* \to (G \to G) \to (G \to G)^*$. ⊠

After having considered the previous examples, a question we could ask at this stage is whether or not it is possible to write down a general form for types. Luckily enough, the answer is affirmative.

Remark 1.1. Let $\sigma_1, ..., \sigma_n$ be types, where $n \in \mathbb{N}_{\geq 0}$. From Definition 1.1, we can conclude that types are *always* of the form
$$\sigma_1 \to \cdots \to \sigma_n \to \rho, \tag{1.1}$$
where ρ is either the ground type G or a star type (i.e. σ^* for some type σ).

For instance, the type $(\sigma^* \to \tau)^*$ from Example 1.2 corresponds to the case where ρ is a star type and $n = 0$. Form (1.1) is a direct consequence of Definition 1.1 and can be seen by induction on the complexity of the type. ⊠

There is one last aspect to mention about types, namely the concept of *currying*. In order to discuss it, we first make the following remark.

Remark 1.2. *(Different "types" of types)* It was not a coincidence that we opened the present section on finite types with an example related to the natural numbers. In our definition of types (Definition 1.1), we defined types based on a ground type and two type constructors, namely $\to$ and $*$. However, this is not the original definition of finite types. Originally, finite types did not even include star types, only a ground type that represented the natural numbers and the type constructor $\to$ (as was the case with the introductory example).

As will be seen in Section 2, finite types were also used in 1958 by Kurt Gödel in his article *Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes* [26], translated to English as *On a hitherto unutilized extension of the finitary standpoint* [25], which we mentioned in the introduction. Gödel's definition of finite types was stated as detailed below:

> "Here the notion 'computable function of type t' is defined as follows:
>
> 1. the computable functions of type 0 are the natural numbers;
> 2. if the notions 'computable function of type t_0', 'computable function of type t_1', ..., 'computable function of type t_k' (with $k > 1$) have already been defined, then a computable function of type $(t_0, t_1, ..., t_k)$ is defined as an operation, always performable (and constructively recognizable as such), that to every k-tuple of computable functions of types $t_1, ..., t_k$ assigns a computable function of type t_0 (...).
>
> The functions occurring in this hierarchy are called 'computable functions of finite type over the natural numbers.'"

This definition was one of the ground stones to extend Hilbert's finitary reasoning. While Hilbert only dealt with concrete and computable functions from the naturals to the naturals, Gödel defined them to act on naturals, on functions, on functions of functions, etc.

Even though the finite types Gödel mentioned in his definition were intended for a different framework (the framework of arithmetic), they are the basis for the finite types from Definition 1.1. As can be seen by Gödel's definition, the symbol 0 represented the ground type whose intended meaning was to represent the set of the natural numbers (what we chose to denote N when talking about $\mathscr{L}^{\omega}_{\mathrm{nat}}$ at the beginning of the present section). Although challenging to notice at first, Gödel's type constructor was already $\to$. This can be seen by first recalling that some authors write $\tau\sigma$ for the type $\sigma \to \tau$ and by noticing that the type $(t_0, t_1, ..., t_k)$ from clause 2. above is the type $t_k \to ... \to t_1 \to t_0$. Clause 2. then states that if we apply an object of type $t_k \to ... \to t_1 \to t_0$ to an object of type $t_k \to ... \to t_1$, we obtain an object of type t_0.

Despite the fact that finite types were initially restricted solely to natural numbers and had only one type constructor, further theoretical developments made authors create new type constructors based on other sets. A type constructor that has often been included in the definition of types is the type constructor $\times$ to create the so called *product type* $\sigma \times \tau$ for types σ, τ. The intended meaning of the product type $\sigma \times \tau$ is to represent the cartesian product where the first coordinate is of type σ and the second coordinate is of type τ. Contrary to the product type, which has been in use for a long time, the star type is fairly recent and was first introduced for classical logic in 2017 by Fernando Ferreira and Gilda Ferreira in their article *A herbrandized functional interpretation of classical first-order logic* [21]. $\boxtimes$

As mentioned in the previous remark, finite types can be defined based on a ground type and on type constructors such as $\to$, $\times$ and $*$. Nevertheless, many authors choose not to include the type constructor $\times$ because of a process called *currying*.

Remark 1.3. (*Currying*) In simple terms, currying avoids the use of the product type $\times$ when the function type $\to$ is provided. It represents the process of transforming a type such as $\sigma \times \tau \to \rho$ into the type $\sigma \to \tau \to \rho$. This is why some authors choose not to include the product type and therefore usually restrict their definition of finite types to a ground type and the type constructor $\to$.

To illustrate this process, let R, S, T be sets. It is a common notation to write R^T to indicate the set of functions from T to R. From set-theory, it is widely-known that $R^{S\times T}$ is isomorphic to $(R^T)^S$, that is, it is possible to define a bijection sending a function $f : S \times T \to R$ to the function $f' : S \to (T \to R)$. This correspondence between $R^{S\times T}$ and $(R^T)^S$ is a particular case of currying.

In the framework of this dissertation, currying states that there exists a correspondence between the types $\sigma \times \tau \to \rho$ and $\sigma \to \tau \to \rho$. This is the reason why the types of $+$ and $\cdot$ are indeed $N \to N \to N$ as stated at the beginning of the present section. $\boxtimes$

In the upcoming section, we will follow Gödel's footsteps. Once Gödel had introduced his notion of finite types through the notion of computable functions, he used them to construct a typed theory, nowadays known as *Gödel's system* T. We will now also construct a typed theory but based on our definition of finite types (Definition 1.1). This theory will be denoted by PL^ω_* and will be described in the next section.

1.2 The Star Combinatory Calculus

The main focus of this section will be to define the *Star Combinatory Calculus* together with various results that hold in it. First, we will define a typed theory denoted by PL^ω_*. The theory PL^ω_* will consist of the language $\mathscr{L}^\omega_*$ covered in Section 1.2.1 and the formal deduction system to go with it, covered in Section 1.2.2.

Notice that the fact that PL^ω_* is a typed theory is illustrated by its own notation. First, the notation PL stands for predicate logic, as to indicate that the language $\mathscr{L}^\omega_*$ of PL^ω_* will be based on a first-order language $\mathscr{L}$. Furthermore, the notation ω stands for finite types and $*$ stands for the star type. We explicitly use the notation $*$ in PL^ω_* because the usual notion of finite types does not contain star types (as these were only recently introduced in 2017 [21]).

1.2.1 The syntax

To define the language $\mathscr{L}^\omega_*$ of PL^ω_*, we fix a language $\mathscr{L}$ of first-order logic (without equality) with at least one constant symbol. The primitive logical symbols for $\mathscr{L}$ are the *disjunction* $\vee$, the *negation* $\neg$ and the *universal quantifier* $\forall$. Recall that these form a complete system of logical symbols for classical logic and, as such, the other symbols, namely the *conjunction* $\wedge$, the *implication* $\rightarrow$, the *equivalence* $\leftrightarrow$ and the *existential quantifier* $\exists$, are the usual (classical) abbreviations based on the primitive symbols

$$A \wedge B :\equiv \neg(\neg A \vee \neg B), \quad A \rightarrow B :\equiv \neg A \vee B, \quad \exists x A :\equiv \neg \forall x \neg A. \tag{1.2}$$

Notice that although they are written with the same symbol, the implication symbol $\rightarrow$ for formulas and the arrow symbol $\rightarrow$ for types do not represent the same notion. However, this is the most widespread notation for both symbols.

The new language $\mathscr{L}^\omega_*$, based on $\mathscr{L}$, will be viewed not only as an extension of $\mathscr{L}$, but also as a *typed* extension of $\mathscr{L}$. Hence, a term t in $\mathscr{L}$ will also be a term in $\mathscr{L}^\omega_*$ and can, for example, be of type $G \rightarrow G$ when seen as a term in $\mathscr{L}^\omega_*$. To define $\mathscr{L}^\omega_*$, we begin by defining its syntax, that is, its constants, terms and formulas.

CONSTANTS

The constants of $\mathscr{L}^\omega_*$ consist of the constants of $\mathscr{L}$ (which will be called *$\mathscr{L}$-constants*), as well as constants that are not in $\mathscr{L}$, namely the *logical constants* (also called *combinators*) and the *star constants* which are constants related to the new star type. As it is expected, each one of these constants will receive an appropriate type.

Definition 1.2. The *constants* of $\mathscr{L}^\omega_*$ consist of the *$\mathscr{L}$-constants*, the *logical constants* and the *star constants* described as follows:

a) The *$\mathscr{L}$-constants*:
   i. For each constant k of $\mathscr{L}$, there is a constant in $\mathscr{L}^\omega_*$ of type G which is also denoted k.
   ii. For each function symbol f of $\mathscr{L}$, there is a constant in $\mathscr{L}^\omega_*$ of type $G \rightarrow \cdots \rightarrow G \rightarrow G$ (the number of arrows corresponds to the arity of f) which is also denoted f.

b) The *logical constants* or *combinators*:[1]
   i. A combinator $\Pi_{\sigma,\tau}$ of type $\sigma \rightarrow \tau \rightarrow \sigma$ for each pair of types σ, τ.
   ii. A combinator $\Sigma_{\sigma,\tau,\rho}$ of type $(\sigma \rightarrow \tau \rightarrow \rho) \rightarrow (\sigma \rightarrow \tau) \rightarrow \sigma \rightarrow \rho$, for each triple of types σ, τ, ρ.

c) The *star constants*:
   i. *Singletons*: a constant $\mathfrak{s}_\sigma$ of type $\sigma \rightarrow \sigma^*$ for each type σ.
   ii. *Unions*: a constant $\cup_\sigma$ of type $\sigma^* \rightarrow (\sigma^* \rightarrow \sigma^*)$ for each type σ.
   iii. *Indexed unions*: a constant $\bigcup_{\sigma,\tau}$ of type $\sigma^* \rightarrow (\sigma \rightarrow \tau^*) \rightarrow \tau^*$ for each pair of types σ, τ.

As previously mentioned, we will write $t : \sigma$ or t^σ to indicate that an object t has type σ. Both notations will be used interchangeably. The names of the star constants will become clear in Remark 1.6.

As the convention of associating types to the right (Notation 1.1) may be rather bewildering at first, especially for longer types, we make the types for both the logical and the star constants of the above definition explicit. Without this convention, the types of the combinators are

$$\begin{aligned} \Pi_{\sigma,\tau} :&\quad \sigma \rightarrow (\tau \rightarrow \sigma), \\ \Sigma_{\sigma,\tau,\rho} :&\quad (\sigma \rightarrow (\tau \rightarrow \rho)) \rightarrow ((\sigma \rightarrow \tau) \rightarrow (\sigma \rightarrow \rho)), \end{aligned}$$

[1] Some authors choose to use the letters K and S for the combinators Π and Σ, respectively.

while the types of the star constants are

$$\begin{aligned} \mathfrak{s}_\sigma :\ & \sigma \to \sigma^*, \\ \cup_\sigma :\ & \sigma^* \to (\sigma^* \to \sigma^*), \\ \bigcup_{\sigma,\tau} :\ & \sigma^* \to ((\sigma \to \tau^*) \to \tau^*). \end{aligned}$$

It is also important to understand that there are not *only* two combinators and three star constants in $\mathscr{L}^\omega_*$. In order to understand this subtle remark, consider the singleton star constant as an example. As the definition explicitly states, there is a constant $\mathfrak{s}_\sigma$ *for each type* σ. Hence, there are as many singleton star constants as there are types. Analogously, there are as many Π combinators as there are pairs of types. A similar reasoning applies to the combinator Σ and to the other star constants.

Remark 1.4. Another aspect to take into account is that Definition 1.2 did not mention how relational symbols from $\mathscr{L}$ were to be typed in $\mathscr{L}^\omega_*$. It was seen that both constants and function symbols of $\mathscr{L}$ were associated to constants with a corresponding type in $\mathscr{L}^\omega_*$. For example, for a binary function symbol f in $\mathscr{L}$, there exists a constant f of type $G \to G \to G$ in $\mathscr{L}^\omega_*$. As will be seen, relational symbols coming from $\mathscr{L}$ are treated differently. More on this once we have introduced the atomic formulas of $\mathscr{L}^\omega_*$ (*cf.* Remark 1.9). ⊠

TERMS

We now direct our attention to the terms in $\mathscr{L}^\omega_*$. These will consist of the terms coming from $\mathscr{L}$, as well as of other terms which do not come from $\mathscr{L}$.

Definition 1.3. The *terms of* $\mathscr{L}^\omega_*$ are defined inductively as follows:

a) All the constants of $\mathscr{L}^\omega_*$ (Definition 1.2) are terms.
b) Variables are terms and, for each type σ, there are countably many variables of type σ denoted by $x^\sigma, y^\sigma, z^\sigma, \ldots$
c) If $t^{\sigma\to\tau}$ and q^σ are terms, then the *application* $app(t,q)$ *of* t *to* q is a term of type τ. In order to improve readability, we will write tq instead of $app(t,q)$.

Before providing examples of terms, we consider two important remarks to help with the general understanding of term applications, in particular of term applications involving the star constants.

Remark 1.5. *(Type restriction on term applications)* Notice that not all terms can be applied to one another. In fact, the third clause of Definition 1.3 forms a restriction on which terms can be applied to one another. Consider the following example. Let σ and τ be types and let t^τ and q^σ be terms. It is in general not possible to apply t to q. For tq to be a valid application, the type of t must be a function type which starts with the type of q. The type τ can, for example, be $\sigma \to \sigma$ or $\sigma \to \tau^* \to \sigma$ (among other possibilities).

Moreover, notice that the application of terms is neither associative nor commutative. Not only would the use of these properties result in different terms, but the applications would not be possible, as the types would not be compatible anymore. ⊠

Remark 1.6. *(Intended meaning of the star constants)* Now that the terms have been defined, we can discuss the intended meaning behind the star constants. The intended meaning of the singleton $\mathfrak{s}_\sigma$ is to map a term $t : \sigma$ to the singleton set $\{t\}$ whose only element is t itself. The intended meaning of the star constant $\cup_\sigma$ is to map terms t^{σ^*} and q^{σ^*} to their union $t \cup q$. Notice that since the union is used between sets, t and q must be of the same *star* type. To conclude, the intended meaning of the star constant $\bigcup_{\sigma,\tau}$ is to map terms t^{σ^*} and $f^{\sigma\to\tau^*}$ to the indexed union $\bigcup_{w\in t} fw$.

Formal notation	Set-theoretic notation
$\mathfrak{s}t$	$\{t\}$
$\cup tq$	$t \cup q$
$\bigcup tf$	$\bigcup_{w\in t} fw$

Table 1: Set-theoretic meaning of applications involving star constants

Notice the change in the order of application in $\bigcup tf$ and $\bigcup_{w\in t} fw$. ⊠

Bearing the last two remarks in mind, we now consider some examples of term applications with their corresponding types.

Example 1.4. Let τ, σ and ρ be types. Consider the following typed terms in $\mathscr{L}^\omega_*$ given by x^σ, $y^{\sigma\to\tau^*}$, $t^{\sigma\to\tau}$, $p^{(\sigma\to\tau)\to\tau\to\rho}$ and $q^{\sigma\to\tau\to\rho}$. The following table provides examples of applications between typed terms on the left together with explanations about their types on the right.

Typed term	Explanation
1. $tx : \tau$	$t : \sigma \to \tau$ and $x : \sigma$
2. $(pt)(tx) : \rho$	$pt : \tau \to \rho$ and $tx : \tau$
3. $\Pi_{\sigma,\tau}x : \tau \to \sigma$	$\Pi_{\sigma,\tau} : \sigma \to \tau \to \sigma$ and $x : \sigma$
4. $(\Sigma_{\sigma,\tau,\rho}q)t : \sigma \to \rho$	$\Sigma_{\sigma,\tau,\rho} : (\sigma \to \tau \to \rho) \to (\sigma \to \tau) \to \sigma \to \rho$ $\Sigma_{\sigma,\tau,\rho}q : (\sigma \to \tau) \to \sigma \to \rho$ $(\Sigma_{\sigma,\rho,\tau}q)t : \sigma \to \rho$
5. $((\Sigma_{\sigma,\tau,\sigma}\Pi_{\sigma,\tau})t)x : \sigma$	$\Sigma_{\sigma,\tau,\sigma} : (\sigma \to \tau \to \sigma) \to (\sigma \to \tau) \to \sigma \to \sigma$ $\Sigma_{\sigma,\tau,\sigma}\Pi_{\sigma,\tau} : (\sigma \to \tau) \to \sigma \to \sigma$ $(\Sigma_{\sigma,\tau,\sigma}\Pi_{\tau,\sigma})t : \sigma \to \sigma$ $((\Sigma_{\sigma,\tau,\sigma}\Pi_{\tau,\sigma})t)x : \sigma$
6. $\cup_\tau((\bigcup_{\sigma,\tau}(\mathfrak{s}_\sigma x))y) : \tau^* \to \tau^*$	$\mathfrak{s}_\sigma x : \sigma^*$ $\bigcup_{\sigma,\tau} : \sigma^* \to ((\sigma \to \tau^*) \to \tau^*)$ $\bigcup_{\sigma,\tau}(\mathfrak{s}_\sigma x) : (\sigma \to \tau^*) \to \tau^*$ $(\bigcup_{\sigma,\tau}(\mathfrak{s}_\sigma x))y : \tau^*$ $\cup_\tau((\bigcup_{\sigma,\tau}(\mathfrak{s}_\sigma x))y) : \tau^* \to \tau^*$

Table 2: Examples of terms in $\mathscr{L}^\omega_*$ with their corresponding types

⊠

As can be seen from this example, the extensive use of brackets for term applications results in a cumbersome notation. To adress this issue, we introduce the following notation.

Notation 1.3. *(Term application notation)* Opposite to the convention for writing types (which were written associated to the right), applications of terms are written *associated to the left.* Hence, if $t_1, ..., t_n$ are terms, instead of writing $((t_1t_2)\cdots)t_n$ we write $t_1t_2\cdots t_n$. Using this notation, the applications 2., 4., 5. and 6. from the above example, will be written, respectively, as $pt(tx)$, $\Sigma_{\sigma,\tau,\rho}qt$, $\Sigma_{\sigma,\tau,\sigma}\Pi_{\sigma,\tau}tx$ and $\cup_\tau(\bigcup_{\sigma,\tau}(\mathfrak{s}_\sigma x)y)$. ⊠

In Example 1.4, we considered terms with more general types. When terms are specifically of type G, they are important enough to be given a specific name.

Definition 1.4. A term of type G is said to be a *ground term*. In particular, a constant of type G is said to be a *ground constant*.

Ground terms include the terms in $\mathscr{L}$ (when seen as terms in $\mathscr{L}_*^\omega$), but are not restricted to them. In fact, it is possible to form ground terms which are exclusive to $\mathscr{L}_*^\omega$. The term $\Pi_{G,\sigma}t^G q^\sigma$ is an example of such a ground term. It is a ground term since $\Pi_{G,\sigma}t : \sigma \to G$ and $\Pi_{G,\sigma}tq : G$, but because it includes the constant $\Pi_{G,\sigma}$ (a constant exclusive to $\mathscr{L}_*^\omega$), the term $\Pi_{G,\sigma}tq$ is also exclusive to $\mathscr{L}_*^\omega$.

At this point, it is worth mentioning that just like there are tuples of types (as seen in Notation 1.4), there are also tuples of terms in $\mathscr{L}_*^\omega$. As done with types, we use an underline to indicate that a term is a tuple. For instance, the notation $\underline{t}$ indicates a *term tuple* such as $t_1, ..., t_n$.

Notation 1.4. *(Term tuples and corresponding types)* Let $k, n \in \mathbb{N}_{\geq 0}$. Let $\underline{t}$ be a term tuple $t_1, ..., t_k$ of type $\underline{\sigma} \to \underline{\tau}$ and let $\underline{q}$ be a term tuple $q_1, ..., q_n$ of type $\underline{\sigma}$. Then the application $\underline{tq}$ denotes the term tuple $t_1q_1...q_n, ..., t_kq_1...q_n$ of type tuple $\underline{\tau}$. ⊠

Example 1.5. Let ρ be a type. Let $\underline{\sigma}$ be the type tuple $G^*, \rho \to \rho^*, \rho$ and let $\underline{\tau}$ be the type tuple G, ρ. Let $\underline{t}$ be the term tuple given by t_1, t_2 of type tuple $\underline{\sigma} \to \underline{\tau}$. The term tuple $\underline{t}$ can then be written as

$$t_1^{G^* \to (\rho \to \rho^*) \to \rho \to G}, t_2^{G^* \to (\rho \to \rho^*) \to \rho \to \rho}.$$

Let $\underline{q}$ be the term tuple q_1, q_2, q_3 of tuple type $\underline{\sigma}$, which can be written as

$$q_1^{G^*}, q_2^{\rho \to \rho^*}, q_3^{\rho}.$$

The application $\underline{tq}$ is then of type $\underline{\tau}$ and is given by $(t_1q_1q_2q_3)^G, (t_2q_1q_2q_3)^\rho$. ⊠

We will now move on to define the formulas of $\mathscr{L}_*^\omega$.

FORMULAS

In order to define the formulas in $\mathscr{L}_*^\omega$, we first need to see if the primitive symbols in $\mathscr{L}_*^\omega$ are the same as in $\mathscr{L}$. The primitive symbols for $\mathscr{L}$ were $\vee$, $\neg$ and $\forall$. Alas, these are no longer sufficient to describe the language $\mathscr{L}_*^\omega$, as we now need to take types into account. This is why we need to extend the primitive symbols and introduce new ones, one for each type.

Definition 1.5. The *primitive symbols* of the language $\mathscr{L}_*^\omega$ consist of

a) the primitive logical connectives $\vee$ (the *disjunction*), $\neg$ (the *negation*) and, for each type σ, an *unbounded universal quantifier* $\forall$,
b) a binary *equality symbol* $=_\sigma$ for each type σ,
c) a binary *membership symbol* $\in_\sigma$ for each type σ,
d) a *bounded universal quantifier* $\forall x^\sigma \in_\sigma t^{\sigma^*}$ for each type σ, where x is a variable and t is a term in which x does not occur.

It should be emphasized that since there is an unbounded universal quantifier $\forall$ *for each type*, a more precise notation would be to write $\forall_\sigma$ for each type σ. However, this is not a widespread notation.

Remark 1.7. In the quantification $\forall x_1, ..., x_n$, the variables x_1 to x_n do not need to be of the same type. Indeed, since $\forall x_1, ..., x_n$ is an abbreviation for $\forall x_1 ... \forall x_n$, we can have $\forall x_1^{\sigma_1} ... \forall x_n^{\sigma_n}$ where the types σ_1,...,σ_n may all be distinct from one another. ⊠

Next, we define the building blocks for the formulas of $\mathscr{L}_*^\omega$, the *atomic formulas*. These are generated from already constructed terms by means of the relational symbols in $\mathscr{L}$, as well as the equality and membership symbols from the previous definition.

Definition 1.6. The *atomic formulas* of $\mathscr{L}_*^\omega$ consist of expressions of the form

a) $R(t_1, ..., t_n)$, where R is an n-ary relational symbol in $\mathscr{L}$ and $t_1, ..., t_n$ are ground terms in $\mathscr{L}_*^\omega$,
b) $t =_\sigma q$, where σ is a type and t^σ and q^σ are terms in $\mathscr{L}_*^\omega$ (expressions of this form are often called *equations of type* σ),
c) $t \in_\sigma q$, where σ is a type and t^σ and q^{σ^*} are terms in $\mathscr{L}_*^\omega$.

If the types are clear, we can omit the types in the equality and membership symbols and write $t = q$ and $t \in q$ instead of $t =_\sigma q$ and $t \in_\sigma q$. The same applies to the bounded universal quantifier, where we may write $\forall x \in t$ instead of $\forall x^\sigma \in_\sigma t^{\sigma^*}$. Moreover, if we wish to emphasize that a formula A in $\mathscr{L}_*^\omega$ is an atomic formula, we will use the index *at* and write A_{at}.

As a quick mention, while the equality symbol is only used between terms of the same type, this is not the case with the membership symbol. In $t \in_\sigma q$, the term q does not have the same type as t, but, instead, its corresponding star type. For example, if σ, τ are types and t_1^σ, t_2^σ, p^{σ^*}, r^τ are terms, then while $t_1 \in_\sigma p$ makes sense, in general neither $t_1 \in_\sigma t_2$ nor $t_1 \in_\sigma r$ do.(2)

We present two remarks with respect to atomic formulas and the types involved in them.

Remark 1.8. The atomic formulas of $\mathscr{L}_*^\omega$ include, in particular, the atomic formulas of the original language $\mathscr{L}$ when seen as objects in $\mathscr{L}_*^\omega$. In fact, atomic formulas in $\mathscr{L}$ are expressions of the form $R(t'_1, ..., t'_n)$, where R is an n-ary relational symbol and $t'_1, ..., t'_n$ are first-order terms in $\mathscr{L}$. But these first-order terms are automatically ground terms in $\mathscr{L}_*^\omega$. ⊠

Remark 1.9. When introducing the constants in $\mathscr{L}_*^\omega$, we briefly mentioned that relational symbols coming from $\mathscr{L}$ are not typed (*cf.* Remark 1.4). Now, in Definition 1.6, this becomes apparent. Although the terms $t_1, ..., t_n$ used as arguments for a relational symbol R coming from $\mathscr{L}$ will always be ground terms, the relational symbol R itself is not typed. The other relational symbols which are exclusive to $\mathscr{L}_*^\omega$ such as $\in_\sigma$ and $=_\sigma$ are typed as can be seen from their definition. ⊠

We proceed with the definition of the *formulas* in $\mathscr{L}_*^\omega$.

Definition 1.7. The *formulas* of $\mathscr{L}_*^\omega$ consist of

a) the atomic formulas of $\mathscr{L}_*^\omega$,
b) the expressions obtained from the atomic formulas by means of the propositional connectives $\neg$ and $\vee$ and the bounded and unbounded universal quantifiers.

As usual, in order to generate formulas using the defined propositional connectives such as implication $\rightarrow$, conjunction $\wedge$ and existential unbounded quantifiers $\exists$, we rely on the usual classical abbreviations 1.2 which can be canonically extended to formulas in $\mathscr{L}_*^\omega$.

(2) The expression $t_1 \in_\sigma r$ would only hold if τ were σ^*.

One of these abbreviations defines the unbounded existential quantifier $\exists$ in terms of the unbounded universal quantifier $\forall$. In line with this, one could question whether there exists a bounded existential quantifier that can be defined from the bounded universal quantifier $\forall x \in t$.

Definition 1.8. Let A be a formula, x a variable and t a term in which x does not occur, all in $\mathscr{L}^\omega_*$. We define the *bounded existential quantifier* $\exists x^\sigma \in_\sigma t^{\sigma^*}$ as

$$\exists x^\sigma \in_\sigma t^{\sigma^*} \ A(x) :\equiv \neg \forall x^\sigma \in_\sigma t^{\sigma^*} \ \neg A(x).$$

Definition 1.9. The *bounded quantifiers* consist of the bounded universal quantifier $\forall x \in t$ and the bounded existential quantifier $\exists x \in t$. The *unbounded quantifiers* consist of their unbounded analogues $\forall$ and $\exists$.

The next definition assigns a name to an important class of formulas. Similarly to the well-known class of quantifier-free formulas which consists of the formulas where no quantifiers occur, we now define a class of formulas related to the new bounded quantifiers: the class of *base formulas*. This class can be viewed as an extension of the class of quantifier-free formulas.

Definition 1.10. The class of *base formulas* is the smallest class of formulas that include the atomic formulas and which is closed under the boolean connectives $\neg$ and $\vee$ and the bounded quantifiers. If a formula A is a base formula, we will use an index b and write A_b.

Note that while a quantifier-free formula does not allow for the occurrence of *any* kind of quantifiers (both bounded and unbounded), in a base formula *bounded* quantifiers may appear. Consequently, any quantifier-free formula is a base formula, but not the other way round.

Example 1.6. Let A be a formula and B_b a base formula, both in $\mathscr{L}^\omega_*$. Let σ and τ be types and let t^{σ^*} and q^{τ^*} be terms in which variables x^σ and y^τ do not occur, respectively.

1. The formula $\forall x \in_\sigma t \ \neg B_b(x)$ is a base formula, while $\forall x \in_\sigma t \ \exists y \ \neg B_b(x, y)$ is not.
2. The formula $\forall x \in t \ \exists y \in q \ [A(x) \vee B_b(y)]$ is a base formula as long as A is base. ⊠

Remark 1.10. The class of base formulas is also closed under $\wedge$, $\rightarrow$ and $\leftrightarrow$. This can be seen by referring to the classical abbreviations for the defined symbols and to the definition of base formulas. ⊠

1.2.2 The axiomatic

In this subsection, we provide the axioms required to work with $\mathscr{L}^\omega_*$ and which axiomatize the typed theory PL^ω_* we will be working in. Here is a brief overview of the axioms essential for our future discussion (especially relevant for the proof of the *Soundness Theorem*, discussed in Section 3.1). The axioms consist of

- the axioms of classical logic (we will use a suitable logical calculus due to Shoenfield),
- the axioms of equality,
- the axioms of the bounded universal quantifiers $\forall x \in t$,
- the axioms of the combinators Π and Σ,
- and the axioms related to the star type $*$, which will be classified as
  - primary axioms of the star constants, and
  - secondary axioms of the star constants.

We begin by describing the axioms and rules for classical logic in the form of a calculus due to American logician Joseph R. Shoenfield. This calculus is complete for first-order logic and was first presented in Shoenfield's notable work *Mathematical Logic* from 1967 ([21] apud [36]).

Axiom 1.1. *Shoenfield's Calculus for classical logic* consists of:

a) Two axiom schemas:
   i. *Excluded Middle*: $\neg A \vee A$,
   ii. *Substitution*: $\forall x A(x) \to A(t)$.

b) Five rules:
   i. *Expansion*: $\dfrac{A}{B \vee A}$,
   ii. *Contraction*: $\dfrac{A \vee A}{A}$,
   iii. *Associativity*: $\dfrac{A \vee (B \vee C)}{(A \vee B) \vee C}$,
   iv. *Cut*: $\dfrac{A \vee B \quad \neg A \vee C}{B \vee C}$,
   v. $\forall$*-introduction*: $\dfrac{A(x) \vee B}{\forall x A(x) \vee B}$, provided that x does not occur free in B.

For the next axioms, it is important to take into consideration that all of the axioms involve types. Therefore, there exists an axiom schema *for each type*. In what follows, let σ, τ, ρ be types.

Axiom 1.2. Let A_b be a base formula and let x^σ, y^σ be variables. The *axioms of equality* consist of the universal closure of

$$x =_\sigma x, \tag{AxE$_1$}$$

$$x =_\sigma y \ \wedge \ A_b \ \to \ A'_b, \tag{AxE$_2$}$$

where A'_b is obtained from A_b by replacing some of the free variable occurrences x by y, as long as y is free for the said occurrence of x.

Remark 1.11. It is worth noting that although the axioms of equality are stated in terms of base formulas, the more general statements for arbitrary formulas follow automatically. ⊠

As expected, it is possible to conclude the symmetry and transitivity of the equality between variables.

Proposition 1.1. Let $x^\sigma, y^\sigma, z^\sigma$ be variables. Then the following statements apply:

a) *Symmetry:* $\mathsf{PL}^\omega_* \vdash \forall x^\sigma \forall y^\sigma \ [\ x =_\sigma y \ \to \ y =_\sigma x\ ]$,
b) *Transitivity:* $\mathsf{PL}^\omega_* \vdash \forall x^\sigma \forall y^\sigma \forall z^\sigma \ [\ x =_\sigma y \ \wedge \ y =_\sigma z \ \to \ x =_\sigma z\ ]$.

Proof. Let $x^\sigma, y^\sigma, z^\sigma$ be variables. For both proofs, notice that equalities between variables of the same type are base formulas so that it is possible to use instances of Axiom AxE_2. For the symmetry property, since $x =_\sigma y \wedge x =_\sigma x \to y =_\sigma x$ is an instance of AxE_2 (where A_b is given by $x =_\sigma x$ and A'_b by $y =_\sigma x$) and since $x =_\sigma x$ corresponds to Axiom AxE_1, it follows that $x =_\sigma y \to y =_\sigma x$. For the transitivity property, since we have already proven the symmetry property, we can simply use $y =_\sigma x \wedge y =_\sigma z \to x =_\sigma z$ which is also an instance of AxE_2 (where A_b is given by $y =_\sigma z$ and A'_b by $x =_\sigma z$) to conclude the proof. □

We now present the axiom that governs the bounded universal quantifier.

Axiom 1.3. Let A be a formula, x^σ a variable and t^{σ^*} a term in which x does not occur. The *axiom of the bounded universal quantifier* $\forall x \in_\sigma t$ is given by the universal closure of

$$\forall x \in_\sigma t\ A(x) \ \leftrightarrow \ \forall x\ (x \in_\sigma t \ \to \ A(x)). \tag{AxU}$$

Remark 1.12. If the corresponding bounded existential quantifier symbol $\exists x \in_\sigma t$ had been taken to be a primitive symbol (instead of a defined symbol as we chose to), then its axiom would be $\exists x \in t\ A_b(x) \ \leftrightarrow\ \exists x\ (x \in t \wedge A_b(x))$. However, we prefer to take the bounded existential quantifier as a defined symbol, in order to keep the analogy with the unbounded quantifiers, where $\forall$ is primitive and $\exists$ is defined. ⊠

Axiom 1.4. The *axioms of the combinators* Π and Σ are

$$\forall x^\sigma \forall y^\tau\ [\ \Pi_{\sigma,\tau} xy =_\sigma x\ ], \tag{AxC_1}$$

$$\forall x^{\rho\to\sigma\to\tau} \forall y^{\rho\to\sigma} \forall z^\rho\ [\ \Sigma_{\rho,\sigma,\tau} xyz =_\tau xz(yz)\ ]. \tag{AxC_2}$$

With regard to the combinators and their axioms, we can make the following considerations.

Remark 1.13.

1. An easy way to remember the axioms for the combinators is to shed the following light upon them. The combinator Π can be viewed as a *cancellator* with respect to the second variable applied to it. In fact, in AxC_1 the second variable y is cancelled under the action of Π. On the other hand, the combinator Σ can be viewed as a *duplicator* since in AxC_2 the variable z is duplicated under the action of Σ.

2. By the way the axioms of the combinators are formulated, one could think that each combinator has a certain arity just as relational and functional symbols have. For example, Axiom AxC_1 could give the impression that the Π combinator has an arity of 2, as there are two terms involved in the axiom (x and y above). From this, it would make sense to use the notation $\Pi(x, y)$ instead of Πxy. However, this is not the correct way to look at it since, in the context of this dissertation, there are no cartesian products. In AxC_1, the left-hand side Πxy is the result of two consecutive applications: first applying Π to x and then applying Πx to y. As a consequence, formally we should opt to write $(\Pi x)y$ or Πxy instead of $\Pi(x, y)$ (even though some authors choose to use this latter notation in a more informal fashion). ⊠

The last axioms we need to formalize the Star Combinatory Calculus are the axioms related to the star constants. These are divided into *primary* and *secondary* axioms.

Axiom 1.5. The *primary axioms of the star constants* are

$$\forall x^\sigma \forall y^{\sigma\to\tau^*}\ [\ \textstyle\bigcup_{\sigma,\tau}(\mathfrak{s}_\sigma x)y =_{\tau^*} yx\ ], \tag{AxP_1}$$

$$\forall x^{\sigma^*} \forall y^{\sigma^*} \forall z^{\sigma\to\tau^*}\ [\ \textstyle\bigcup_{\sigma,\tau}(\cup_\sigma xy)z =_{\tau^*} \cup_\tau(\bigcup_{\sigma,\tau} xz)(\bigcup_{\sigma,\tau} yz)\ ]. \tag{AxP_2}$$

In Remark 1.6, we discussed the intended meaning of the star constants and used set-theoretical notation to convey meaning to the term applications which involved these constants. Bearing this in mind, it is now easy to see that the primary axioms for the star constants correspond to set-theoretical equalities. In fact, if we go back to using the set-theoretical notation we can see that Axiom AxP_1 corresponds to the set-theoretical equality

$$\bigcup_{w\in\{x\}} yw = yx$$

while Axiom AxP_2 corresponds to the set-theoretical equality

$$\bigcup_{w\in x\cup y} zw = \left(\bigcup_{w\in x} zw\right) \cup \left(\bigcup_{w\in y} zw\right).$$

Axiom 1.6. The *secondary axioms of the star constants* are given by

$$\forall x^\sigma \forall y^\sigma \; [\; x \in_\sigma \mathfrak{s}_\sigma y \;\leftrightarrow\; x =_\sigma y \;], \tag{AxS_1}$$

$$\forall x^\sigma \forall a^{\sigma^*} \forall b^{\sigma^*} \; [\; x \in_\sigma \cup_\sigma ab \;\leftrightarrow\; x \in_\sigma a \vee x \in_\sigma b \;], \tag{AxS_2}$$

$$\forall a^{\sigma^*} \forall f^{\sigma \to \tau^*} \forall b^\tau \; [\; b \in_\tau \textstyle\bigcup_{\sigma,\tau} af \;\leftrightarrow\; \exists x \in_\sigma a \; (b \in_\tau fx) \;], \tag{AxS_3}$$

$$\forall x^{\sigma^*} \exists y^\sigma \; [\; y \in_\sigma x \;]. \tag{AxS_4}$$

Now that the axioms have been introduced, we conclude this subsection with some remarks about their structure.

Remark 1.14. *(Axioms as universal closure of base formulas)* Most axioms in this subsection are universal closures of base formulas. In fact, the axioms for equality (Axioms 1.2), the axioms for the combinators (Axioms 1.4), and the primary and secondary axioms for the star constants (Axioms 1.5 and 1.6) are all universal closures of base formulas. The main reason for choosing to write the axioms in terms of base formulas will become clear in later sections. As a matter of fact, once we've introduced Shoenfield's functional interpretation this feature will turn out to be very useful.

Hence, we could have chosen to write that the axioms were universal closures of base formulas and just state the corresponding base formulas. For example, axioms AxS_1 and AxS_3 would then be stated as being the universal closures of the following base formulas

$$x \in_\sigma \mathfrak{s}_\sigma y \;\leftrightarrow\; x =_\sigma y, \tag{$\mathsf{AxS}_1{}'$}$$

$$b \in_\tau \bigcup_{\sigma,\tau} af \;\leftrightarrow\; \exists x \in_\sigma a \; (b \in_\tau fx). \tag{$\mathsf{AxS}_3{}'$}$$

However, in order to keep a clear overview of the types of each term, we chose to explicitly write out the quantifiers in the axioms. ⊠

Furthering this remark, notice that the base formulas which followed the universal quantifiers in the above axioms were mostly written between square brackets. This leads to the next notation which will be extensively used in Sections 2 and 3.

Notation 1.5. Base formulas will often be written between square brackets. Although this notation is not compulsory, it will make the structure of some more complex formulas easier to understand. ⊠

We will come back to this remark and notation in the proof of the Soundness Theorem (Theorem 3.1) in Section 3.

Although every axiom is of importance, Axiom AxS_4 deserves special attention. When introducing the notion of finite types, we mentioned that the intended meaning of the star type was to represent finite and non-empty objects. Axiom AxS_4 now ensures the non-emptiness from an axiomatic point of view by stating that any variable of star type is, in fact, non-empty. We make two observations to complement this explanation.

Remark 1.15. *(Remarks on Axiom AxS_4)*

1. Not only is Axiom AxS_4 important as it translates the intended meaning of the star type, but it is also necessary in order to express certain formulas. For example, only with said axiom is it possible to have certain equivalences such as

$$\exists x \in t \; (A(x) \vee B) \;\leftrightarrow\; \exists x \in t \; A(x) \vee B.$$

2. When examining Axiom AxS_4, one could think that it is not the universal closure of a base formula. However, it is and this can be seen by rewriting the axiom using the usual logical equivalences. As a matter of fact, Axiom AxS_4 could have been stated equivalently (yet less elegantly) as $\forall x \exists y \in x\ (y \in x)$ to show off its structure. ⊠

1.2.3 The Bounded Axiom of Choice

For the upcoming results we need to mention a logical principle, namely the *Bounded Axiom of Choice*. As the name indicates, this principle is related to the usual set-theoretical Axiom of Choice. Following an approach similar to [23], our focus now is to state these two axioms and briefly explore the connection between them. Although there are various formulations for the usual set-theoretical Axiom of Choice, the formulation which is of interest to us states:

Any collection of non-empty sets has a choice function. (AC')

This formulation can be expressed as

$$\forall x \exists y\ A(x,y) \to \exists f \forall x\ A(x, fx). \tag{AC}$$

To see why, let $(S_x)_{x \in X}$ denote a collection of non-empty sets and let X be an index-set. The formulation AC' states that there exists a function $f(x)$ (called *choice function*) which verifies $\forall x\ (f(x) \in S_x)$. By letting $A(x,y)$ denote the condition $y \in S_x$, the formulation AC' can be expressed as AC, where the antecedent $\forall x \exists y\ A(x,y)$ denotes that all S_x are non-empty and the consequent $\exists f \forall x\ A(x, fx)$ expresses the existence of a choice function f.

Up to now, our brief discussion around the Axiom of Choice did not involve types. However, if we wish to express AC in PL^ω_* we must type it. By doing so, we obtain a first typed formulation of the Axiom of Choice.

Axiom 1.7. *(Axiom of Choice without the star type)* The *Axiom of Choice without the star type* is given by

$$\forall x^\sigma \exists y^\tau\ A(x,y) \to \exists f^{\sigma \to \tau} \forall x^\sigma\ A(x, fx), \tag{AC^ω}$$

where A is a formula.

This typed version of the Axiom of Choice is used in many works where finite types are used, but not the star type (refer, for example, to [23], where the author presents the Axiom of Choice for Heyting Arithmetic in all finite types, denoted HA^ω_0).

In the context of this dissertation we will use a version of the Axiom of Choice called the *Bounded Axiom of Choice* which we will denote by bAC^ω_* (Axiom 1.8). There are two main differences between AC^ω and bAC^ω_*. First, bAC^ω_* is restricted to base formulas (which explains the b in bAC^ω_*). Second, not only does bAC^ω_* use the star type (which explains the $*$ in bAC^ω_*) but, more importantly, it does not explicitly state the existence of a witness f for y in $A(x,y)$. Instead, it states the existence of a finite and non-empty set in which we can find a witness. More on this after the axiom itself.

Axiom 1.8. *(Bounded Axiom of Choice)* The *Bounded Axiom of Choice* is given by

$$\forall \underline{x}^{\underline{\sigma}} \exists \underline{y}^{\underline{\tau}}\ A_b(\underline{x}, \underline{y}) \to \exists f^{\underline{\sigma} \to \underline{\tau}^*} \forall \underline{x}^{\underline{\sigma}} \exists \underline{y} \in_{\tau} \underline{f}\underline{x}\ A_b(\underline{x}, \underline{y}), \tag{bAC^ω_*}$$

where A_b is a base formula of $\mathscr{L}^\omega_*$.

For technical reasons that will become apparent in Section 3, the axiom has to be stated for tuples of variables and tuples of types. Nevertheless, in order to compare the different versions of the Axiom of Choice, we now continue our discussion considering only individual variables.

The Axioms AC^ω and bAC^ω_* express slightly different ideas. The Axiom of Choice AC^ω states that there is a witness for y in $A(x,y)$ and the witness is explicitly given by fx. The Bounded Axiom of Choice bAC^ω_* states that there is a finite non-empty set (represented by fx) of possible witnesses that includes an actual witness for y in $A_b(x,y)$. Hence, while in AC^ω the application fx designates a witness, in bAC^ω_* the application fx designates a finite non-empty set of possible witnesses.

That fx has a different meaning in both axioms also becomes clear when looking at the types of the terms in them. First, notice that the type of the choice functions f in AC^ω and in bAC^ω_* are not the same.

1. In AC^ω, the variable f has type $\sigma \to \tau$ and, therefore, the term fx has type τ, which is the same type as y.
2. In bAC^ω_* the term f has type $\sigma \to \tau^*$ and the term fx has type τ^*. As a consequence, since τ^* is a star type, the term fx in bAC^ω_* can be represented by a finite and non-empty set $\{y_1^\tau, ..., y_n^\tau\}^{\tau^*}$, where each of the y_i, with $1 \leq i \leq n$, is a potential witness for y in $A_b(x,y)$. In bAC^ω_* the witness is chosen from this set and is no longer explicitly given as in AC^ω.

Hence, it is as if elevating the types in bAC^ω_* prevented the explicit definition of the witness as it was possible in AC^ω. It is, in fact, only possible to state that there is a finite and non-empty set where the witness is chosen from.

There are some further remarks to consider about the Bounded Axiom of Choice.

Remark 1.16.

1. As the choice function f in bAC^ω_* has type $\sigma \to \tau^*$, it is no longer a first-order variable. The Axiom bAC^ω_* can therefore not be expressed in first-order logic.
2. The Bounded Axiom of Choice is not the universal closure of a base formula (this will be of interest in the upcoming section). ⊠

The Bounded Axiom of Choice will assume a significant role in the upcoming sections. It will be further discussed after defining a herbrandized version of Shoenfield's functional interpretation in Section 2.2.

Now that the foundational guidelines of PL^ω_* have been established, next we will discuss the first result of this dissertation, a result related to the representation of terms in PL^ω_*.

1.2.4 The Combinatorial Completeness Theorem

Consider the function that takes any natural number and adds 2 to it. This function is usually written $f(x) = x + 2$. As can be seen, this notation can only be used if the function one wants to represent has a name such as f above. Although it is common to attribute names to functions, sometimes this is not necessary. A widespread nameless way to represent f would be to write it as $x \mapsto x + 2$. Another nameless notation that is equally widespread represents the above function as $\lambda x.(x + 2)$, as to indicate an anonymous function whose input is x and whose output is $x + 2$. This notation is usually referred to as *λ-abstraction* (*lambda abstraction*). This notation also makes it possible to evaluate functions. Usually, if we were to evaluate the function f at 5, we would write $f(5) = 5 + 2 = 7$. The same can now be expressed as $(\lambda x.(x + 2))5 = 5 + 2 = 7$, to indicate that x is replaced by 5 in $x + 2$. This form of expressing function evaluation is often referred to as *β-reduction* or *β-conversion* (*beta reduction/conversion*). From this example, it can be seen that this notation involving the λ-operator is useful to both represent and discuss functions.

The next result is about functions and how functions can be represented in PL^ω_* by means of terms. This result, called the *Combinatorial Completeness Theorem*, states that given a term $t(x)$, where t is a term that may have x as a variable, it is possible to construct a term q that represents (and acts like) the function $x \mapsto t(x)$ in PL^ω_*. The Combinatorial Completeness Theorem will also reveal the reason why it was necessary to define the combinators Σ and Π in the first place.

Theorem 1.1. (Combinatorial Completeness) *Let x^σ be a variable. For each term $t^\tau(x)$ in $\mathscr{L}^\omega_*$, there exists a term $q^{\sigma\to\tau}$ in $\mathscr{L}^\omega_*$ whose variables are those of t except x such that*

$$\mathsf{PL}^\omega_* \vdash qs =_\tau t[{}^s\!/\!_x]$$

for any term s^σ in $\mathscr{L}^\omega_*$.

Recall that $t[{}^s\!/\!_x]$ denotes the substitution operation where x is replaced by s in the term t. The substitution operation $t[{}^s\!/\!_x]$ of replacing x by s in t is, as usual, defined by induction on the complexity of t.

The theorem therefore states that there always exists a term q in $\mathscr{L}^\omega_*$ that does the same as the substitution operator, i.e. qs replaces x by s in $t(x)$. Naturally, the terms t and s must be of the same type in order for the substitution to take place.

Notation 1.6. Since the term q therefore acts like the function $x \mapsto t(x)$, we will use the notation $\lambda x.t$ for q. With this notation, the theorem states that

$$\mathsf{PL}^\omega_* \vdash (\lambda x.t)s =_\tau t[{}^s\!/\!_x]. \qquad \boxtimes$$

Remark 1.17. Notice that in the Star Combinatory Calculus, $\lambda x.t$ is not a primitive symbol of the language, it is simply a notation. This contrasts with other contexts where $\lambda x.t$ is seen as a primitive symbol (refer to Appendix A for a brief digression on this topic). Nevertheless and as mentioned above, the notation $\lambda x.t$ is called λ-*abstraction*, while the passage from $(\lambda x.t)s$ to $t[{}^s\!/\!_x]$ is often referred to as β-*reduction* or β-*conversion*. Since, in the present dissertation, $(\lambda x.t)s$ is still the application of two terms, the convention remains to associate the terms to the left (see Notation 1.3). $\boxtimes$

To make these notions transparent, consider again the introductory example. Just like in the example where $\lambda x.(x+2)$ represented the function $x \mapsto x+2$, in the theorem the term q represents the function $x \mapsto t(x)$, which explains why $\lambda x.t$ is an adequate notation for q. Moreover, just like $(\lambda x.(x+2))5$ replaces x by 5 in $x+2$, the application qs, *i.e.* $(\lambda x.t(x))s$, replaces x by s in $t(x)$. Continuing with this example, consider how the theorem would be applied to this function.

Example 1.7. Consider the term t in $\mathscr{L}^\omega_{\text{nat}}$ given by $x+2$, where x is a variable of type N. The Combinatorial Completeness Theorem guarantees that there is a term $q : N \to N$ (with the same variables as t except for x) that performs the same task as the function $x \mapsto x+2$. For example, if we consider the term $succ0$ (where, as before, $succ$ is the usual successor function), then performing the application $q(succ0)$ or performing the substitution $t[{}^{succ0}\!/\!_x] = succ0 + 2 = 1 + 2 = 3$ would lead to the same result. $\boxtimes$

Before proving the theorem, notice how the λ-abstraction $\lambda x.t$ is to be looked upon when the variable x and the term t are tuples.

Notation 1.7. *(λ-abstraction for tuples)* Let $\underline{x}$ be the tuple of variables $x_1, ..., x_k$ in $\mathscr{L}^{\omega}_{*}$.

1. If t is a term in PL^{ω}_{*}, then we write $\lambda\underline{x}.t$ or $\lambda x_1, ..., x_k.t$ to denote the term $\lambda x_1.(\lambda x_2.(...(\lambda x_k.t)...))$.
2. If $\underline{t}$ is the tuple of terms given by $t_1, ..., t_n$ in PL^{ω}_{*}, then we write $\lambda\underline{x}.\underline{t}$ to denote the tuple of terms $\lambda\underline{x}.t_1, ..., \lambda\underline{x}.t_n$. ⊠

For the proof of the theorem, it will be necessary to construct a term q for each possible term t in PL^{ω}_{*}. The definition of q will therefore depend on the complexity of t and q will be represented through the use of λ-abstractions. The following definition for q will be suitable for the proof.

Definition 1.11. Let x be a variable and let t, t_1, t_2 be terms. The term q is defined as follows:

a) If t is the variable x, then $\lambda x.x := \Sigma\Pi\Pi$.
b) If x does not occur in t, then $\lambda x.t := \Pi t$.
c) In any other case, define $\lambda x.(t_1 t_2) := \Sigma(\lambda x.t_1)(\lambda x.t_2)$.

Example 1.8. If t is the constant Σ, then we would use case b) and $q = \lambda x.\Sigma = \Pi\Sigma$. In the case where t is a variable different from x, we would also use case b). ⊠

In any of the three cases of the definition, the term q is defined using combinators. Now, the name of the theorem becomes clear. The *Combinatorial* Completeness Theorem relies on the two classes of *combinators* Σ and Π and these classes are enough to build a term q for *any* term t of $\mathscr{L}^{\omega}_{*}$. The combinators make it possible to define λ-abstractions in PL^{ω}_{*}. From here on, every time the λ-notation is used, we are inherently and unavoidably invoking the Combinatorial Completeness Theorem.

Now that the term q has been constructed in Definition 1.11, in order to prove the Combinatorial Completeness Theorem we need to show that the theorem holds for each one of the clauses of the definition. For example, for the last clause, we need to verify that given a term s, we have

$$qs = (\lambda x.(t_1 t_2))s \overset{\text{def.}}{=} \Sigma(\lambda x.t_1)(\lambda x.t_2)s \overset{\text{to show}}{=} (t_1 t_2)[{}^{s}/_{x}].$$

Proof (Combinatorial Completeness). Let σ, τ and ρ be types. Let $t^{\tau}(x^{\sigma})$ be a term and x a variable, both in $\mathscr{L}^{\omega}_{*}$. We need to show the existence of a term $q^{\sigma\to\tau}$ for each of the cases of Definition 1.11. We will prove the theorem by induction on the complexity of q. The proof relies on the combinator axioms AxC_1 and AxC_2 (Axioms 1.4).

Case 1. We need to show that $\lambda x.x := \Sigma\Pi\Pi$ is an appropriate definition for q in the case where the term t is the variable x. This will be done by showing $(\lambda x.x)s = x[{}^{s}/_{x}]$. Since replacing x by s in x is simply s, we have $x[{}^{s}/_{x}] = s$ and, therefore,

$$qs = (\lambda x.x)s \overset{\text{def.}}{=} \Sigma\Pi\Pi s \overset{\mathsf{AxC}_2}{=} \Pi s(\Pi s) \overset{\mathsf{AxC}_1}{=} s = x[{}^{s}/_{x}].$$

Case 2. We need to show that $\lambda x.t := \Pi t$ is an appropriate definition for q in the case where x does not occur in t by showing $(\lambda x.t)s = t[{}^{s}/_{x}]$. First notice that $t[{}^{s}/_{x}] = t$ (since x does not occur in t, no substitution takes place in t). Therefore, we have

$$qs = (\lambda x.t)s \overset{\text{def.}}{=} (\Pi t)s \overset{\mathsf{AxC}_1}{=} t = t[{}^{s}/_{x}].$$

Case 3. Lastly, we need to show that $\lambda x.(t_1 t_2) := \Sigma(\lambda x.t_1)(\lambda x.t_2)$ is an appropriate definition for q in the remaining cases by showing $(\lambda x.(t_1 t_2))s = (t_1 t_2)[{}^{s}/_{x}]$. By induction hypothesis (I.H.), there already exist terms $\lambda x.t_1$ and $\lambda x.t_2$ (respectively for terms t_1 and t_2) such that $(\lambda x.t_1)s = t_1[s/x]$ and $(\lambda x.t_2)s = t_2[s/x]$, respectively. Therefore,

$$qs = (\lambda x.(t_1 t_2))s \overset{\text{def.}}{=} \Sigma(\lambda x.t_1)(\lambda x.t_2)s \overset{\mathsf{AxC_2}}{=} (\lambda x.t_1)s((\lambda x.t_2)s) \overset{\text{I.H.}}{=} t_1[s/x]t_2[s/x] = (t_1 t_2)[s/x],$$

where the last equality holds, since replacing x by s in both t_1 and t_2 and then applying t_1 to t_2 is the same as replacing x by s in the application $t_1 t_2$. Both approaches result in the same term. $\square$

With λ-abstractions at hand, we can now make an important observation that will prove to be useful later on.

Remark 1.18. *(Every type is inhabited by at least one term)* For every type in $\mathscr{L}_*^\omega$, there exists at least one term of that type. To show this, we will define a term for every possible type, whether it is the ground type, the arrow type or the star type. When we introduced the language $\mathscr{L}$ at the beginning of this section, we assumed it had at least one constant symbol. Let k be such a constant symbol in $\mathscr{L}$. From the existence of k, it is now possible to systematically define a closed term in $\mathscr{L}_*^\omega$, also denoted k, for any given type as follows

$$\begin{aligned} k^G &:= k, \\ k^{\sigma \to \tau} &:= \lambda x^\sigma . k^\tau, \\ k^{\sigma^*} &:= \mathfrak{s} k^\sigma, \end{aligned}$$

where σ and τ are arbitrary types. As can be seen, we have associated to a constant symbol k of $\mathscr{L}$ a typed term in $\mathscr{L}_*^\omega$ and this for every possible type. We will come back to this remark in the proof of the Soundness Theorem in Section 3.1. $\boxtimes$

We finish this subsection with a comment on what will be discussed next. A key aspect of the present subsection was the Combinatorial Completeness Theorem and, in particular, the notion of β-conversion. In the upcoming subsection, we will formalize the notion of *conversion* to a more general context and study important results related to the Star Combinatory Calculus.

1.2.5 Conversions and the Church-Rosser and Strong Normalization Theorems

The Star Combinatory Calculus enjoys two interesting properties: the *Church-Rosser* and the *strong normalization* properties. In order to understand and discuss these results, we need some preliminary notions, one of them being another syntactic aspect of the Star Combinatory Calculus, namely its *conversions*.

Conversions can, very informally, be seen as *one-sided* equalities. It is a way of *transforming* terms into other terms. Just like we write $t_1 = t_2$ to indicate the equality between terms t_1 and t_2, we will write $t_1 \rightsquigarrow t_2$ to indicate the above equality, but only in one direction, from left to right, which will be called a *conversion* from t_1 to t_2.

For example, if we consider the natural numbers, writing $2 + 5 = 7$ would mean that $2 + 5$ is equal to 7 and that 7 is equal to $2 + 5$ (two directions). Although there are two possible directions, one of the two *simplifies* the term more than the other direction does. In this sense, conversions can be seen to represent the direction that further simplifies the term.

From equalities it is therefore possible to define conversions by choosing a direction. Our goal is now to define the conversions which can be used on the terms of the Star Combinatory Calculus. In Section 1.2.2, we provided the axioms of the combinators and the primary axioms of the star constants. By using the Substitution Axiom Scheme together with Modus Ponens,[3] we obtain the following corresponding formulas for terms:

1. $\Pi tq = t$,
2. $\Sigma tqr = tr(qr)$,
3. $\bigcup(\mathfrak{s}t)q = qt$,
4. $\bigcup(\cup tq)r = \cup\left(\bigcup tr\right)\left(\bigcup qr\right)$.

Each one of these equalities will define a conversion in the Star Combinatory Calculus and these conversions are regrouped in the next definition.

Definition 1.12. Let t, q, r be terms in $\mathscr{L}^{\omega}_{*}$ of appropriate types. The *conversions* of the Star Combinatory Calculus are given by

C1. $\Pi tq \rightsquigarrow t$,
C2. $\Sigma tqr \rightsquigarrow tr(qr)$,
C3. $\bigcup(\mathfrak{s}t)q \rightsquigarrow qt$,
C4. $\bigcup(\cup tq)r \rightsquigarrow \cup\left(\bigcup tr\right)\left(\bigcup qr\right)$.

A conversion can be seen as a binary relation on the set of all terms and will be represented by the symbol $\rightsquigarrow$ to indicate a conversion from left to right. If t, q are terms, we write $t \rightsquigarrow q$ and say that t *converts to* q. For example, Conversion C1 is to be read as "the term Πtq *converts to* the term t". Notice that the terms t, q, r from the definition do not necessarily have the same type in each conversion.

The Star Combinatory Calculus together with the conversions from Definition 1.12 is said to be a term rewriting system. In simple terms, a *term rewriting system* is a system of rules (here of conversions) that allows us to transform and rewrite terms of a language into other, often simpler, terms. Another name for a term rewriting system is *reduction system*, as the goal of conversions is to simplify, that is, to *reduce*, terms. For a more formal treatment of term rewriting systems, refer to Appendix B.

The following remark will be of importance for upcoming proofs. Just like equalities preserve types,[4] so do conversions.

Remark 1.19. *(Conversions preserve types)* Let t_1^{σ} and t_2^{τ} be terms. If $t_1^{\sigma} \rightsquigarrow t_2^{\tau}$, then σ and τ must be the same type. ⊠

Example 1.9. To offer a concrete example of this remark, we return to Conversion C4 of Definition 1.12 given by $\bigcup(\cup tq)r \rightsquigarrow \cup\left(\bigcup tr\right)\left(\bigcup qr\right)$. We will see that the terms on each side of the conversion have the same type.

1. *Lefthand side.* If we suppose $\cup : \sigma^* \to (\sigma^* \to \sigma^*)$, then $t : \sigma^*$, $q : \sigma^*$ and $\cup tq : \sigma^*$. As a consequence, for the term $\bigcup(\cup tq)r$ to be a valid application, we must have $\bigcup : \sigma^* \to ((\sigma \to \tau^*) \to \tau^*)$. But then $\bigcup(\cup tq) : (\sigma \to \tau^*) \to \tau^*$ and $r : \sigma \to \tau^*$, which implies $\bigcup(\cup tq)r : \tau^*$.

2. *Righthand side.* Assuming that the terms t, q, r have the same types as in 1., then $\bigcup tr : \tau^*$, $\bigcup qr : \tau^*$ and finally $\cup\left(\bigcup tr\right)\left(\bigcup qr\right) : \tau^*$.

[3] Recall that even though Modus Ponens is not part of the axiomatic of PL^{ω}_{*}, we can freely use it since Shoenfield's Calculus is complete.

[4] The terms involved in an equation are necessarily of the same type (refer to Definition 1.6).

Both sides of the Conversion C4 have the same type, namely τ^*. Altogether yielding

$$\left(\bigcup_{\sigma,\tau} (\cup_\sigma tq)^{\sigma^*} r^{\sigma \to \tau^*} \right)^{\tau^*} \rightsquigarrow \left(\cup_\tau \left(\bigcup_{\sigma,\tau} tr \right)^{\tau^*} \left(\bigcup_{\sigma,\tau} qr \right)^{\tau^*} \right)^{\tau^*}.$$

The verification of the types involved in the other conversion rules can be computed analogously.⊠

The conversions from Definition 1.12 enhanced the Star Combinatory Calculus to a term rewriting system. If we were to work with a different calculus, we would define a different set of rules, that is, a different set of conversions, and hence obtain different term rewriting systems. From this point onward, and unless stated otherwise, the definitions and results from this subsection (Section 1.2.5) are valid for any term rewriting system, not only for the Star Combinatory Calculus. However, we will use the Star Combinatory Calculus with its conversions from Definition 1.12 when providing examples. We will come back to more specific results about the Star Combinatory Calculus at the end of this subsection.

To motivate the next definition, consider the computation $6 + 3 + 7$. Using the above notation, we could simplify $6 + 3 + 7 \rightsquigarrow 9 + 7$ and then $9 + 7 \rightsquigarrow 16$. In order to compute $6 + 3 + 7$, it was necessary to do two computations until it was no longer possible to simplify the result any further. Just as it was done for natural numbers, it is possible to apply multiple conversions to certain terms until it is no longer possible to apply any conversion to them. This is expressed by the notion of *reducibility.*

Definition 1.13. Let t, q be terms.

a) A term t *reduces* to a term q *in one step* (written $t \succ_1 q$) if q can be obtained by replacing some subterm u of t by some v such that $u \rightsquigarrow v$. We say that q is obtained from t through a *one-step reduction.*
b) A term t *reduces* to a term q (written $t \succcurlyeq q$) if q can be obtained from t by a finite sequence of one-step reductions. If this sequence is composed of n one-step reductions, we say that q is obtained from t through an *n-step reduction.*
c) If t is a term and u is a subterm of t, then u is called a *redex* of t if a conversion can be applied to u, i.e., there is some term v such that $u \rightsquigarrow v$. The corresponding right side of the conversion is called a *reduct* or a *conversum.*
d) If t has a redex, then t is said to be *reducible* or *convertible.* Otherwise, we say that t is *irreducible* or in *normal form.*

In short, a term t reduces to a term q if q can be obtained from t through conversions and a term is in normal form if it can no longer be reduced through the use of conversions. In the above example, $6 + 3 + 7$ is reducible where $6 + 3$ is a redex and 9 is, at first, a reduct. The number 16 is in normal form and is obtained from $6 + 3 + 7$ through a 2-step reduction.

Remark 1.20. In this section, we mostly follow the terminology and notation used in [39]. However, it is worth mentioning that some authors use the terminology *contraction* and *contractum* instead of *conversion* and *conversum.* Moreover, the use of *reduction rules* instead of *conversions* is widespread as well. ⊠

We make two remarks related to Definition 1.13.

Remark 1.21. *(Reductions preserve types)* Since reductions are obtained from successive conversions and since conversions preserve types (see Remark 1.19), reductions also preserve types. ⊠

Remark 1.22. *(Closures of $\succ_1$)* Relative to Definition 1.13, there is an equivalent way to define the reducibility relation $\succcurlyeq$ in a way that further shows the connection between $\succ_1$ and $\succcurlyeq$. In fact, $\succcurlyeq$ can also be defined as the *reflexive and transitive closure* of $\succ_1$, that is, $\succcurlyeq$ is the smallest binary relation that contains $\succ_1$ and that is both reflexive and transitive. Using this terminology, some authors include a different definition stating that the relation $\succ$ is defined to be the transitive closure of $\succ_1$.

What is important to take from this remark is that any term t reduces to itself and that if a term q reduces to a term r and r reduces to t, then q reduces to t. Having $t \succcurlyeq t$, one could now ask whether or not $t \succ_1 t$ also holds. For the Star Combinatory Calculus, when seen as the term rewriting system based of Definition 1.13, it does not hold. As can be seen, in each conversion, every redex is necessarily different from its corresponding reduct. However, for other term rewriting systems $t \succ_1 t$ might hold. ⊠

In order to be more specific on how a term is reduced, we introduce the notion of *reduction sequences.*

Definition 1.14. A *reduction sequence* is a (finite or infinite) sequence of pairs $(t_0, u_0), (t_1, u_1), (t_2, u_2), ...$ where u_i is a redex in t_i and $t_i \succ t_{i+1}$ by the conversion of u_i, for all $i \in \mathbb{N}_{\geq 0}$. Such a reduction sequence is written

$$t_0 \overset{u_0}{\succ_1} t_1 \overset{u_1}{\succ_1} t_2 \overset{u_2}{\succ_1} ...$$

or simply $t_0 \succ_1 t_1 \succ_1 t_2 \succ_1 ...$ if we omit the redexes u_i.

To illustrate the previous definitions, consider the following examples.

Example 1.10. Let p, q, r, t be terms in $\mathscr{L}^\omega_*$ of appropriate types. We will now see how $\Sigma r(\Pi qt) \succcurlyeq \Sigma rq$, $\bigcup(\mathfrak{s}t)pq \succcurlyeq ptq$ and $\bigcup\left(\cup(\mathfrak{s}t)q\right) r \succcurlyeq \cup\,(rt)\left(\bigcup qr\right)$.

1. The term $\Sigma r(\Pi qt)$ is reducible and can be converted to Σrq using a one-step reduction. This is achieved by applying Conversion C1 to the redex Πqt to obtain the reduct q. As a consequence, $\Sigma r(\Pi qt)$ is reduced to Σrq. The reduction sequence is given by
$$\Sigma r(\Pi qt) \overset{\Pi qt}{\succ_1} \Sigma rq. \tag{1.4}$$

2. The term $\bigcup(\mathfrak{s}t)pq$ is reducible and can be reduced in one step to ptq by applying Conversion C3. Here $\bigcup(\mathfrak{s}t)p$ is a redex, pt is a reduct and the reduction sequence is given by
$$\bigcup(\mathfrak{s}t)pq \overset{\bigcup(\mathfrak{s}t)p}{\succ_1} ptq. \tag{1.5}$$

3. The term $\bigcup\left(\cup(\mathfrak{s}t)q\right) r$ is reducible and can be reduced to $\cup\,(rt)\left(\bigcup qr\right)$ by first applying Conversion C4 and then Conversion C3. The 2-step reduction sequence is given by
$$\bigcup\left(\cup(\mathfrak{s}t)q\right) r \overset{\bigcup(\cup(\mathfrak{s}t)q)r}{\succ_1} \cup\left(\bigcup(\mathfrak{s}t)r\right)\left(\bigcup qr\right) \overset{\bigcup(\mathfrak{s}t)r}{\succ_1} \cup\,(rt)\left(\bigcup qr\right). \tag{1.6}$$

   In the first reduction step, the term $\bigcup\left(\cup(\mathfrak{s}t)q\right) r$ is a redex with corresponding reduct given by $\cup\left(\bigcup(\mathfrak{s}t)r\right)\left(\bigcup qr\right)$. In the second reduction step, $\bigcup(\mathfrak{s}t)r$ is a redex and rt is a reduct. ⊠

Moving on, consider again the computation $6+3+7$. If two different people were to be asked to solve it, maybe person 1 would compute it as $(6+3)+7$ while person 2 would compute it as $6+(3+7)$. Both would be right, as they would just be using the associative property of the usual addition in the natural numbers. At the end, independent of the chosen computation path, both would arrive at the result 16 and conclude that $6+3+7 \succcurlyeq 16$. Just like there are different computation paths with the addition of natural numbers, there might also be different reduction sequences for the reduction of terms.

Example 1.11. Let t, q, r, s be normal terms in $\mathscr{L}_*^\omega$ of appropriate types and consider the term $\Sigma r(\Pi qt)s$. We will see that this term can be reduced to the normal form $rs(qs)$ and that there are two possible reduction sequences for $\Sigma r(\Pi qt)s \succcurlyeq rs(qs)$, depending on which conversion we apply first.

Sequence 1: $\quad \Sigma r(\Pi qt)s \overset{\Pi qt}{\succ_1} \Sigma rqs \overset{\Sigma rqs}{\succ_1} rs(qs)$

Sequence 2: $\quad \Sigma r(\Pi qt)s \overset{\Sigma r(\Pi qt)s}{\succ_1} rs((\Pi qt)s) \overset{\Pi qt}{\succ_1} rs(qs)$

Either we first apply Conversion C1 followed by Conversion C2 (Sequence 1) or we first apply Conversion C2 followed by Conversion C1 (Sequence 2). As can be seen, there is no other possible reduction sequence for $\Sigma r(\Pi qt)s$, as $\Sigma r(\Pi qt)s$ only has two redexes: Πqt and $\Sigma r(\Pi qt)s$ itself. Notice that all possible reduction sequences lead to the same normal term, namely $rs(qs)$.⊠

To better visualize situations like this, we introduce the following notation.

Notation 1.8. The reducibility notions from Definition 1.13 can be represented through directed graphs where nodes represent terms and edges represent reductions. We will use an arrow with a 1 as superscript ($\xrightarrow{1}$) to indicate a one-step reduction $\succ_1$ and an arrow with # as superscript ($\xrightarrow{\#}$) to indicate a reduction $\succcurlyeq$.[5] ⊠

Example 1.12. *(Visualisation of Example* 1.11) Using this notation, the two reduction sequences from Example 1.11 would be represented by the following directed graph:

$$\begin{tikzcd}
\Sigma r(\Pi qt)s \arrow[r, "1"] \arrow[d, "1"] & \Sigma rqs \arrow[d, "1"] \\
rs((\Pi qt)s) \arrow[r, "1"] & rs(qs)
\end{tikzcd}$$

Figure 1: Representation of Example 1.11 through a directed graph

⊠

As already mentioned above, both sequences of Example 1.11 not only lead to a normal form, but both lead to the same normal form $rs(qs)$. From this example, it seems plausible to assume that, at least in the Star Combinatory Calculus, this is always the case and that no matter how we reduce a term, we will always arrive at the same normal form. This is no coincidence and, although this observation may seem trivial, it is not. As a matter of fact, it requires the use of powerful theorems that will be discussed at the end of this subsection. But before stating them, it is necessary to introduce some new terminology.

[5] This is not the standard notation for directed graphs representing these notions. Usually a one-step reduction is represented by an arrow without any superscript ($\to$) and a reduction is represented by an arrow with $*$ as superscript ($\xrightarrow{*}$). However, in this dissertation we prefer to leave the symbols $\to$ and $*$ mainly for types.

Definition 1.15. Let t be a term.

a) A term t is *normalizable* if it can be reduced to a term in normal form.
b) A term t is *strongly normalizable* if there are no infinite reduction sequences beginning with t, that is, every such sequence eventually leads to a term in normal form.

Example 1.13. In Example 1.11, the term $\Sigma r(\Pi qt)s$ is both normalizable and strongly normalizable. Indeed, as discussed before, the two finite reduction sequences mentioned from this example are the only two possible ways to reduce the term to its normal form $rs(qs)$. ⊠

The definitions introduced up to now were related to terms. Now we introduce analogue definitions for the whole term rewriting system.

Definition 1.16.

a) A term rewriting system is said to be *normalizing* if every term is normalizable.
b) A term rewriting system is said to be *strongly normalizing* if every term is strongly normalizable.
c) A term rewriting system is said to be *locally confluent*, or said to have the *local Church-Rosser property*, if whenever $t \succ_1 u$ and $t \succ_1 v$, then there is a term q such that $u \succcurlyeq q$ and $v \succcurlyeq q$.

$$\begin{tikzcd} t \arrow[r, "1"] \arrow[d, "1"] & u \arrow[d, "\#"] \\ v \arrow[r, "\#"] & q \end{tikzcd}$$

Figure 2: Directed graph representing the property of local confluence

d) A term rewriting system is said to be *(globally) confluent*, or said to have the *(global) Church Rosser Property*, if whenever $t \succcurlyeq u$ and $t \succcurlyeq v$, then there is a term q such that $u \succcurlyeq q$ and $v \succcurlyeq q$.[(6)]

$$\begin{tikzcd} t \arrow[r, "\#"] \arrow[d, "\#"] & u \arrow[d, "\#"] \\ v \arrow[r, "\#"] & q \end{tikzcd}$$

Figure 3: Directed graph representing the property of confluence

Remark 1.23. Notice that in the definition of local confluence, the reduction sequences associated to the reductions $u \succcurlyeq q$ and $v \succcurlyeq q$ don't need to be of the same length. The same applies to the length of the reduction sequences associated to $t \succcurlyeq u$, $t \succcurlyeq v$, $u \succcurlyeq q$ and $v \succcurlyeq q$ in the definition of confluence. ⊠

In a nutshell, normalization asserts that it is possible to find a normal form of a given term, while confluence asserts that the way you find such a normal form does not matter.

(6) In this dissertation, saying that a term rewriting system has the Church-Rosser Property will be the same as saying that it is confluent. The same is valid for the local Church-Rosser Property being the same as local confluence. Depending on the literature, some authors choose to make a distinction between these notions considering them to be different but equivalent notions.

Remark 1.24. Both normalization and strong normalization are notions about the *existence* of reduction sequences. Normalization says that there always exists a way to reduce a term to a normal form. Strong normalization states that if we start to reduce a term t then there are no infinite reduction sequences $t \succ_1 t' \succ_1 t'' \succ_1 \ldots$ Once we start to reduce a term, *there is no turning back*; we will eventually, and in a finite number of steps, reach a term q that can no longer be reduced, that is, the reduction sequences are of the form $t \succ_1 t' \succ_1 t'' \succ_1 \ldots \succ_1 q$. ⊠

Although the existence of reduction sequences is ensured when the term rewriting system is (strongly) normalizing, observe that neither the confluence nor the normalizing notions state the *uniqueness* of the normal form. It could be that, depending on the term rewriting system and the reduction strategy we choose, a term would have different normal forms. A term rewriting system that guarantees the uniqueness of a normal form, when it exists, is said to have the *Unique Normal Form Property.* The next result states that confluent term rewriting systems have this property.

Proposition 1.2. (Unique Normal Form Property) *In a confluent term rewriting system, any term has at most one normal form.*

Due to this proposition, whenever a term rewriting system is confluent it is possible to refer to *the* normal form of a term: if it exists, then it is unique.

Proof. Let S be a confluent term rewriting system and let t be a term in S. If t does not have a normal form, then the proposition is proven. Assume, therefore, that t has at least one normal form and let u and v be normal forms of t. In particular, we have that $t \succcurlyeq u$ and $t \succcurlyeq v$. Since S is confluent, u and v both reduce to some common term q. However, since u and v were already normal forms they can't be further reduced and therefore must both be the same as q. □

We now shift our attention to another matter. As expected, being locally confluent does not imply being (globally) confluent. However, if we add another condition, then the implication holds and any locally confluent term rewriting system that verifies this extra condition will automatically become confluent. This is given by *Newman's Lemma*, a result developed by British mathematician Maxwell Newman in his pubblication *On theories with a combinatorial definition of "equivalence"* [31].

Lemma 1.1. (Newman's Lemma) *If a term rewriting system is locally confluent and strongly normalizing, then it is also confluent.*

As a result of Newman's Lemma, if a term rewriting system is strongly normalizing, then it suffices to prove a local property (local confluence) in order to prove a global property (confluence).

The original proof of Newman's Lemma given by Newman himself [31] is deemed to be more complicated than necessary and even involves numerical estimations. Various other authors have presented alternative, elegant and shorter proofs for this theorem. We refer to [39] for a simple proof of Newman's Lemma which uses the notion of reduction trees, a notion which will not be explored in this dissertation. Further results together with more extensive discussions about term rewriting systems in general can be found in [6], [29] and [28].

We will now come back to the Star Combinatory Calculus with its conversions from Definition 1.12 and reducibility notions from Definition 1.13. We are able to state that the Star Combinatory Calculus is both confluent and strongly normalizing.

Theorem 1.2. (Strong Normalization Theorem) *The Star Combinatory Calculus with the conversions from Definition* 1.12 *is strongly normalizing.*

Theorem 1.3. (Church-Rosser Theorem) *The Star Combinatory Calculus with the conversions from Definition* 1.12 *is confluent, i.e. it enjoys the Church-Rosser Property.*

To prove the two theorems, it would be necessary to introduce new notions that would exceed the scope of this dissertation. In particular, to show the Strong Normalization Theorem we would need to introduce a technique of reducibility which was first developed by American logician William W. Tait in [37] and [38] and then extended to the star type in [21]. The Church-Rosser Theorem then follows from the Strong Normalization Theorem by applying Newman's Lemma and by proving that the Star Combinatory Calculus is locally confluent. We refer to the proofs of the Strong Normalization Theorem and the Church-Rosser Theorem in [21] (Theorems 1 and 2 respectively).

A direct consequence of the Church-Rosser Theorem is the following proposition.

Corollary 1.1. *Every term in the Star Combinatory Calculus has at most one normal form.*

Proof. Direct consequence of the Church-Rosser Theorem and Proposition 1.2. □

The two theorems of this section together with the above corollary now allow us to state that what was discussed in Example 1.11 really was no coincidence. In the Star Combinatory Calculus all terms reduce to a normal form (*strongly normalizing*) and this normal form is unique (*confluent*).

Now that we have established that every term in the Star Combinatory Calculus is reducible to a term in normal form, we turn our attention to those terms that are simultaneously normal and closed. How these terms can be characterized in PL^ω_* will be discussed next.

1.2.6 Characterizations of closed normal terms

In conclusion to Section 1, we will use the notions from the previous subsection to characterize closed normal terms of $\mathscr{L}^\omega_*$. We will see that, depending on their type, closed normal terms in $\mathscr{L}^\omega_*$ will either be analogue to terms in $\mathscr{L}$ or will be described by the notion of *set-like terms.* We begin by making two insightful observations about the general forms of closed terms (Remark 1.25) and of closed normal terms (Remark 1.26).

Remark 1.25. *(General form of closed terms)* Since by definition terms are obtained from constants, variables and the application of previously constructed terms, any term must either be of the form $kt'_1t'_2...t'_m$ or of the form $xt'_1t'_2...t'_m$,[7] where $m \in \mathbb{N}_{\geq 0}$, k is a constant, x a variable and $t'_1, ..., t'_m$ terms, all in $\mathscr{L}^\omega_*$. However, as by definition variables are not closed terms, closed terms cannot begin with the application of a variable x to some closed term t_1 and must therefore necessarily be of the form

$$kt_1t_2...t_m \tag{1.7}$$

where m and k are as above and $t_1, ..., t_m$ are now *closed* terms in $\mathscr{L}^\omega_*$. To be more precise, by Definition 1.2 the constant k is one of the following: k', f, Σ, Π, $\mathfrak{s}$, $\cup$ or $\bigcup$, where k' is a constant coming from $\mathscr{L}$ and f is an n-ary function symbol from $\mathscr{L}$. Any closed term in $\mathscr{L}^\omega_*$ starts with one of these constants. ⊠

[7] Recall that $kt_1t_2...t_m$ and $xt_1t_2...t_m$ are to be read, respectively, as $(...((kt_1)t_2)...)t_m$ and $(...((xt_1)t_2)...)t_m$.

Remark 1.26. *(General form of closed normal terms)* By the previous remark, closed terms must be of the form (1.7). To find the general form of the closed *normal* terms in $\mathscr{L}_*^\omega$, we must guarantee that when the constant k is applied to the terms $t_1, ..., t_n$ we don't form reducible terms. Going by cases on the constant k of $\mathscr{L}_*^\omega$ in (1.7) and relying on Definition 1.2, a closed normal term t must be of one of the following forms:

1. k',
2. f, fp_1, fp_1p_2, ..., $fp_1p_2...p_n$,
3. Σ, Σq_1, $\Sigma q_1 q_2$,
4. Π, Πr
5. $\mathfrak{s}$, $\mathfrak{s}u$,
6. $\cup$, $\cup v_1$, $\cup v_1 v_2$,
7. $\bigcup$, $\bigcup w_1$ or $\bigcup w_1 w_2$.

where k' and f are as in Remark 1.25 and $p_1, ..., p_n, q_1, q_2, r, u, v_1, v_2, w_1, w_2$ are closed normal terms in $\mathscr{L}_*^\omega$, all of appropriate types.

Notice that cases such as $\Sigma q_1 q_2 q_3$ and $\Pi r_1 r_2$ (where q_3, r_1, r_2 are closed normal terms in $\mathscr{L}_*^\omega$ of appropriate types) are not admissible in the above list because these terms are not normal. In fact, they are reducible to $q_1 q_3 (q_2 q_3)$ and r_1, respectively. $\boxtimes$

As stated in the beginning of this subsection, the characterization of closed normal terms will depend on their types. Equipped with remarks 1.25 and 1.26, we begin with the case where the closed normal terms are of type G before considering closed normal terms of star type.

Proposition 1.3. (Ground Normal Form Proposition) *Let t be a term in $\mathscr{L}_*^\omega$. If t is a closed normal term of ground type, then t corresponds to a (closed) first-order term of the language $\mathscr{L}$.*

Proof. Let t be a closed normal term of ground type. By Remark 1.25, t is of the form $kt_1t_2...t_m$ (refer to form (1.7)). We will prove the proposition by cases on the constant k using the cases from Remark 1.26. We will see that in most of these cases the terms are not of ground type, hence failing the assumptions of the proposition. The cases where the terms are indeed of ground type will precisely be terms coming from the original language $\mathscr{L}$.

Let $k', f, p_1, ..., p_n, q_1, q_2, q_3, r, r_1, r_2, u, v_1, v_2, w_1, w_2$ be closed normal terms in $\mathscr{L}_*^\omega$ as in Remark 1.26. We consider each one of the cases from Remark 1.26 and begin with the $\mathscr{L}$-constants.

1. Since k' is a constant in $\mathscr{L}$, it must be of ground type when seen as a constant in $\mathscr{L}_*^\omega$. As these constants are closed normal ground terms and stem from first-order terms, this first case is proven.

2. For the function symbols, we will use an inductive argument on the complexity of the terms. Let g be a binary function symbol in $\mathscr{L}$. Such a function symbol leads to the constant $g : G \to (G \to G)$ in $\mathscr{L}_*^\omega$. Let p_1 and p_2 be terms which correspond to first-order terms. Since $p_1 : G$ and $p_2 : G$, then $gp_1 : G \to G$ and $gp_1p_2 : G$. As can be seen, although g and gp_1 are closed and normal, they are not ground terms, failing the conditions of the proposition. Only the term gp_1p_2 is closed, normal and of ground type. Since, by induction hypothesis, p_1 and p_2 correspond to first-order terms, so does gp_1p_2.

   A similar argument can be applied when considering an n-ary function symbol f in $\mathscr{L}$. Such a function symbol leads to a constant f in $\mathscr{L}_*^\omega$. Let $p_1, ..., p_n$ be closed, normal terms of ground type which correspond to first-order terms. Any application $fp_1...p_i$ with $1 \leq i < n$ leads to a term which is closed and normal, but not of ground type. Only the term $fp_1...p_n : G$ leads to a closed, normal, ground term. Finally, since, by induction hypothesis, $p_1, ..., p_n$ correspond to first-order terms, then so does $fp_1...p_n$.

Now for the cases involving the combinators.

3. The terms
$$\begin{aligned} \Sigma_{\rho,\sigma,\tau} &: (\rho \to \sigma \to \tau) \to (\rho \to \sigma) \to \rho \to \tau, \\ \Sigma_{\rho,\sigma,\tau} q_1^{\rho\to\sigma\to\tau} &: (\rho \to \sigma) \to \rho \to \tau, \\ \Sigma_{\rho,\sigma,\tau} q_1^{\rho\to\sigma\to\tau} q_2^{\rho\to\sigma} &: \rho \to \tau \end{aligned}$$
are all closed normal terms, but none of them is a ground term. Moreover, the closed term $\Sigma_{\rho,\sigma,\tau} q_1^{\rho\to\sigma\to\tau} q_2^{\rho\to\sigma} q_3^{\rho} : \tau$ is only then a ground term if τ is G. But even if τ were G, the term $\Sigma q_1 q_2 q_3$ would still not be normal, as already mentioned in Remark 1.26.

4. The terms
$$\begin{aligned} \Pi_{\sigma,\tau} &: \sigma \to (\tau \to \sigma), \\ \Pi_{\sigma,\tau} r^{\sigma} &: \tau \to \sigma \end{aligned}$$
are both closed normal terms, but none is of ground type. Similarly to the case with the combinator Σ and the case $\Sigma q_1 q_2 q_3$, the closed term $\Pi r_1 r_2$ might be a ground term but is never normal.

Hence, all the terms involving the combinators fail at least one of the conditions of the proposition. Finally, we consider the cases with the star constants.

5. The terms $\mathfrak{s}_\sigma$ and $\mathfrak{s}_\sigma u^\sigma$ are never ground terms as $\mathfrak{s}_\sigma : \sigma \to \sigma^*$ and $\mathfrak{s}_\sigma u^\sigma : \sigma^*$.

6. The terms $\cup_\sigma$, $\cup_\sigma v_1^{\sigma^*}$ and $\cup_\sigma v_1^{\sigma^*} v_2^{\sigma^*}$ are never ground terms as $\cup_\sigma : \sigma^* \to (\sigma^* \to \sigma^*)$, $\cup_\sigma v_1^{\sigma^*} : \sigma^* \to \sigma^*$ and $\cup_\sigma v_1^{\sigma^*} v_2^{\sigma^*} : \sigma^*$.

7. The terms $\bigcup_{\sigma,\tau} : \sigma^* \to (\sigma \to \tau^*) \to \tau^*$, $\bigcup_{\sigma,\tau} w_1^{\sigma^*} : (\sigma \to \tau^*) \to \tau^*$ and $\bigcup_{\sigma,\tau} w_1^{\sigma^*} w_2^{\sigma\to\tau^*} : \tau^*$ are also never ground terms.

All the terms involving the star constants are closed and normal, but none of them is a ground term, failing again one of the conditions of the proposition.

Hence, in conclusion, closed normal ground terms are of the form $kt_1...t_n$ where the constant k must either be a constant k' of $\mathscr{L}$ or a term of the form $fp_1...p_n$. These two cases correspond to first-order terms, while all the other cases from Remark 1.26 fail at least one of the conditions of the proposition. $\square$

After having considered closed normal terms of ground type, we now study the case where the closed normal terms have star types. For this, we need a preliminary definition.

Definition 1.17. Let σ be a type. A term t of star type σ^* is said to be *set-like* if it is only built from terms of the form $\mathfrak{s}q^\sigma$, where q is any term of type σ, and the union operator $\cup_\sigma$.

Example 1.14. Let p_1, p_2, q_1, q_2, r_1, r_2, t, u_1, u_2 be terms and let σ, τ and ρ be types.

1. The term $\mathfrak{s}t^{G\to G}$ is set-like, while the term $\cup_\sigma r_1^{\sigma^*} \mathfrak{s} r_2^{\sigma}$ is only then set-like if r_1 is set-like. Recall that, from a set-theoretical point of view, these terms can be written, respectively, as $\{t\}$ and $r_1 \cup \{r_2\}$ (*cf.* Remark 1.6).

2. The term $\mathfrak{s}\left(\bigcup_{\sigma,\tau} u_1^{\sigma^*} u_2^{\sigma\to\tau^*}\right)$ is set-like, while $\bigcup_{\sigma,\tau} u_1^{\sigma^*} u_2^{\sigma\to\tau^*}$ is not. $\boxtimes$

As can be seen, set-like terms correspond to terms that, from a set-theoretical point of view, are binary unions of singletons (more on this in Remark 1.27).

With this in mind, we can now state that closed normal terms of star type are precisely terms which are set-like.

Proposition 1.4. (Star Normal Form Proposition) *Let σ be a type. If t is a closed normal term of star type σ^*, then t is set-like.*

Proof. Let t be a closed normal term of some star type. We will again use Remarks 1.25 and 1.26 and follow a reasoning similar to the proof of Proposition 1.3 to characterize the closed normal terms of star type. Looking at the constants of Remark 1.26 and the discussion of their respective types in the proof of Proposition 1.3, the only cases in which a term t can be closed, normal and of star type is when it is of the form $\mathfrak{s}u$, $\cup v_1 v_2$ or $\bigcup w_1 w_2$, where u, v_1, v_2, w_1, w_2 are as before. We prove the proposition by induction on the complexity of t.

1. If t is of the form $\mathfrak{s}_\sigma u^\sigma$, then t has type σ^* and, by Definition 1.17, the term t is set-like.
2. If t is of the form $\cup_\sigma v_1^{\sigma^*} v_2^{\sigma^*}$, then t has type σ^*. Moreover, since by induction hypothesis v_1 and v_2 are set-like terms, t is set-like as well.
3. The case when t is of the form $\bigcup_{\sigma,\tau} w_1^{\sigma^*} w_2^{\sigma \to \tau^*}$ cannot hold as it is never normal. Indeed, since w_1 is a closed normal term of star type σ^*, by induction hypothesis, it is set-like. Since w_1 is set-like, then it must either be of the form $\mathfrak{s}w$ or of the form $\cup w w'$ for closed normal terms w, w' of appropriate types. In both cases, the term $\bigcup w_1 w_2$ is reducible (refer to Definition 1.12) and hence not normal. In fact:
   - If w_1 is $\mathfrak{s}w$, then $\bigcup w_1 w_2$ is not normal, as $\bigcup(\mathfrak{s}w)w_2$ reduces to $w_2 w$.
   - If w_1 is $\cup w w'$, then $\bigcup w_1 w_2$ is not normal, as $\bigcup(\cup w w')w_2$ reduces to $\cup\left(\bigcup w w_2\right)\left(\bigcup w' w_2\right)$.

In conclusion, closed normal terms of star type are necessarily of the form $\mathfrak{s}u$ and $\cup v_1 v_2$ and, in both cases, the terms are set-like. ☐

With these results we can now formulate another remark on the structure of closed normal terms. This remark will also prove to be valuable later on when we direct our attention to Herbrand's Theorem in Section 3.2.3.

Remark 1.27. *(Structure of closed normal terms)* A closed normal term t of star type σ^* is necessarily of the form

$$\{t_1\}^{\sigma^*} \cup \{t_2\}^{\sigma^*} \cup ... \cup \{t_n\}^{\sigma^*} \tag{1.8}$$

where $t_1, ..., t_n$ are closed normal terms of type σ.[(8)] In the particular case where t is of type G^*, then, by the Ground Normal Form Proposition, $t_1, ..., t_n$ are first-order terms. From here on, we will often simply write $\{t_1, ..., t_n\}$ instead of $\{t_1\} \cup \{t_2\} \cup ... \cup \{t_n\}$. ⊠

Proof of the remark. Let t be a closed normal term of type σ^*. By the Star Normal Form Proposition (Proposition 1.4), the term t is set-like and is therefore, by definition, either a singleton or a union of terms. If t is a singleton, then it is given by $\{t_1\}$, where t_1 is a term of type σ. Notice that since t is closed and normal, then so is t_1. Since $\{t_1\}$ already corresponds to a term of the form (1.8), assume now that t is a union such as $p^{\sigma^*} \cup q^{\sigma^*}$. Once again, since t is closed and normal, then so are p and q. Now by the Star Normal Form Proposition, p and q are also set-like and are again either a singleton or a union of two closed normal terms of star

[(8)]Here we opt to use the set-theoretic notation in order to make this remark easier to understand.

type σ^*. By continually applying this proposition, the term t is then necessarily of the form $\{t_1\} \cup \{t_2\} \cup ... \cup \{t_n\}$ as in (1.8).

For the particular case where $t : G^*$, by the previous paragraph, we know that t is of the form $\{t_1\}^{G^*} \cup \{t_2\}^{G^*} \cup ... \cup \{t_n\}^{G^*}$, where each t_i is a closed normal term of type G. But then, by the Ground Normal Form Proposition (Proposition 1.3), each t_i corresponds to a first-order term. $\square$

This remark concludes the first section of the dissertation which focused on defining the required framework to work in PL^{ω}_{*}. Having now introduced most of the fundamental definitions and results needed for this dissertation we will dive into the heart of the matter: *functional interpretations.*

2 Shoenfield's functional interpretation

The present section is about *functional interpretations*, in particular, a herbrandized version of *Shoenfield's functional interpretation.* In short, functional interpretations translate formulas from one theory into formulas of another theory and are therefore also sometimes referred to as *translations.* Before formalizing this concept, we will use the notion of prenexification as a stepping stone to introducing functional interpretations. Although prenexification and functional interpretations serve different purposes, drawing a parallel between them is a good starting point to our discussion.

2.1 A short introduction

From basic logic courses, it is known that any first-order formula can, following certain rules, be transformed into an equivalent formula in prenex normal form. In other terms, it is possible to associate to any formula in PL a new equivalent formula where all quantifiers (if these exist) are joined to the left and which are then followed by a quantifier-free formula. For example, if B and C are quantifier-free first-order formulas, then the formula $\forall w\,(\exists x B(w,x) \to \exists y \forall z C(w,y,z))$ can be written in prenex normal form, for example, as

$$\forall w, x \exists y \forall z\ (B(w,x) \to C(w,y,z)). \tag{2.1}$$

This process of finding a prenex normal form of a formula could now be canonically extended to formulas in PL^ω_*, provided that we accept certain equivalences (refer to E_1 and E_2 below). A formula in prenex normal form in PL^ω_* would also have all unbounded quantifiers joined together, but these would now be followed by a base formula instead of a quantifier-free formula, since PL^ω_* also has bounded quantifiers. To extend this process, we need to show how to put bounded quantifiers (when these exist) in the matrix of the formula. This can be accomplished with the following two transformations:

E1. $\forall x \in t\ \exists y\ A(x,y)$ is equivalent to $\exists z\ \forall x \in t\ \exists y \in z\ A(x,y)$,
E2. $\exists x \in t\ \forall y\ A(x,y)$ is equivalent to $\forall z\ \exists x \in t\ \forall y \in z\ A(x,y)$.

where A is a formula in PL^ω_*. Notice that formulas such as $\forall x \in t\ \forall y\ A(x,y)$ are equivalent to $\forall y\ \forall x \in t\ A(x,y)$ (and dually for formulas with existential bounded and unbounded quantifiers). Hence, if we let B and C be formulas in PL^ω_*, then formula (2.1) is also in prenex normal form in PL^ω_*, since quantifier-free formulas such as $B(w,x) \to C(w,y,z)$ are also base formulas.

Even though formulas in prenex normal form are useful because all unbounded quantifiers are joined to the left, there is a setback we should mention. The more unbounded quantifiers a formula has, the more difficult it becomes to understand its meaning, especially when universal and existential quantifiers are "mixed up".[9] It would be of interest to have a process that, similarly to prenexification, joins all unbounded quantifiers together, but, unlike prenexification, would necessarily separate universal from existential quantifiers. Such a process would associate to any formula an equivalent formula either of the form $\forall \underline{x} \exists \underline{y} A_b(\underline{x}, \underline{y})$ or $\exists \underline{y} \forall \underline{x} A_b(\underline{y}, \underline{x})$, where A_b would be a base formula and where $\underline{x}$ and $\underline{y}$ would be (possibly empty) tuples of variables created during the translation process to encompass, respectively, all universal and all existential variables.

Functional interpretations provide such a process to "organize" the unbounded quantifiers. How these are organized will depend on the functional interpretation we consider. In other words, it

[9] This is the case of (2.1), where the universal quantifiers are separated by an existential quantifier. Moreover, independently of the way we choose to prenexify the original formula, we will always end up with an analogous situation.

will depend on the functional interpretation whether the universal quantifiers come before or after the existential quantifiers. Different frameworks will require different organizations and, therefore, different interpretations.

Two important functional interpretations are *Shoenfield's functional interpretation*, used for classical logic, and Gödel's functional interpretation (also known as *Dialectica interpretation*), used for intuitionistic logic and which was the very first functional interpretation. In the case of Shoenfield's interpretation, all universal quantifications are joined together followed by all existential quantifications, while in the Dialectica interpretation it is the opposite, the existential quantifiers come before the universal ones. Our goal will be to define a herbrandized version of Shoenfield's functional interpretation to translate formulas in PL^{ω}_{*} into formulas in PL^{ω}_{*} as follows:

$$\text{Formula in } \mathsf{PL}^{\omega}_{*} \xrightarrow[\textit{interpretation}]{\textit{Shoenfield's}} \begin{array}{c}\text{Formula in } \mathsf{PL}^{\omega}_{*} \\ \text{of the form } \forall \underline{x} \exists \underline{y} A_b(\underline{x}, \underline{y})\end{array}$$

In light of the above, functional interpretations seem to be a better alternative to prenexification. Unfortunately, there is a price to pay to organise the unbounded quantifiers. Translating a formula is a trade-off in two senses:

1. The first trade-off is between organization of quantifiers and complexity of types. In fact, the translation of some formulas will require the creation of new types of higher complexity, types that were not in the original formulas. This will become apparent when we describe the exact translation process in Definition 2.1.
2. The second trade-off is between organization of quantifiers and absence of logical equivalence between a formula and its interpretation. On the one hand, the quantifiers become organised. On the other hand, contrarily to what happens when prenexifying a formula of first-order logic, the interpretation of a formula is not necessarily equivalent to the original formula. Equivalence only holds under certain conditions and will usually require a stronger theory. This will be discussed in Section 3.2.

To conclude this short introduction, there is one final setback we should mention. Contrary to functional interpretations, the prenexification process is not appropriate for extracting computational information from proofs. But more on this in Section 3.1.

2.2 A herbrandized version of Shoenfield's functional interpretation

As stated above, functional interpretations can, very informally, be viewed as special translation tools between two theories. More formally, the translation process can be described by a map $I : \mathscr{L}_1 \to \mathscr{L}_2$ from a language $\mathscr{L}_1$ to a second language $\mathscr{L}_2$, where every formula of the language $\mathscr{L}_1$ will be translated into a formula of the language $\mathscr{L}_2$. We will now formalize a *herbrandized version of Shoenfield's functional interpretation*, the core of the present dissertation, which makes it possible to interpret the language $\mathscr{L}^{\omega}_{*}$ into itself. In other words, we will describe a map of the form $SH : \mathscr{L}^{\omega}_{*} \to \mathscr{L}^{\omega}_{*}$ (where S stands for Shoenfield and H for herbrandized). We will then provide examples of interpretations and make some useful remarks that will accompany us throughout Sections 2 and 3.

Since our main focus is the herbrandized version of Shoenfield's functional interpretation we are about to define, unless stated otherwise the word *interpretation* will refer to this specific functional interpretation. Moreover, we will from now on refer to the herbrandized version of Shoenfield's functional interpretation simply as Shoenfield's functional interpretation. The reason for which we use the word *herbrandized* to characterize this interpretation will be explained later.

According to Shoenfield's functional interpretation, every formula A in $\mathscr{L}_*^\omega$ will be translated into a formula of the form $\forall \underline{a} \exists \underline{b}\ A'_b(\underline{a}, \underline{b})$ also in $\mathscr{L}_*^\omega$, where $\underline{a}$ and $\underline{b}$ are (possibly empty) disjoint tuples of variables and A'_b is a base formula. Since the primitive logical symbols of the language $\mathscr{L}_*^\omega$ are, in particular, $\in$, $=$, $\vee$, $\neg$, $\forall$ and $\forall x \in t$, we need to define a translation process for formulas involving each one of them.

Definition 2.1. *Shoenfield's functional interpretation* assigns to each formula A in $\mathscr{L}_*^\omega$ two formulas A^{SH} and A_{SH} in $\mathscr{L}_*^\omega$ related by the expression

$$A^{SH} \equiv \forall \underline{a} \exists \underline{b}\ A_{SH}(\underline{a}, \underline{b}) \tag{2.2}$$

where $A_{SH}(\underline{a}, \underline{b})$ is a base formula and $\underline{a}$, $\underline{b}$ are (possibly empty) disjoint tuples of variables. The assignment is done according to the following five clauses.

a) If A is a base formula, then A^{SH} and A_{SH} are A, that is, $A^{SH} :\equiv A_{SH} :\equiv A$ and $\underline{a}, \underline{b}$ are empty tuples.

For the remaining cases, if we already have interpretations for two formulas A and B in $\mathscr{L}_*^\omega$, given respectively by $A^{SH} \equiv \forall \underline{a} \exists \underline{b}\ A_{SH}(\underline{a}, \underline{b})$ and $B^{SH} \equiv \forall \underline{c} \exists \underline{d}\ B_{SH}(\underline{c}, \underline{d})$, where $\underline{a}, \underline{b}, \underline{c}, \underline{d}$ are (possibly empty) tuples of variables, then we define

b) the interpretation of a disjunction by

$$\begin{aligned}(A \vee B)^{SH} &:\equiv \forall \underline{a}, \underline{c}\ \exists \underline{b}, \underline{d}\ (A \vee B)_{SH}(\underline{a}, \underline{c}, \underline{b}, \underline{d}) \\ &:\equiv \forall \underline{a}, \underline{c}\ \exists \underline{b}, \underline{d}\ [A_{SH}(\underline{a}, \underline{b}) \vee B_{SH}(\underline{c}, \underline{d})],\end{aligned}$$

c) the interpretation of a negation by

$$\begin{aligned}(\neg A)^{SH} &:\equiv \forall \underline{f}\ \exists \underline{a}'\ (\neg A)_{SH}(\underline{f}, \underline{a}') \\ &:\equiv \forall \underline{f}\ \exists \underline{a}'\ [\exists \underline{a} \in \underline{a}'\ \neg A_{SH}(\underline{a}, \underline{f}\underline{a})],\end{aligned}$$

d) the interpretation of a universal quantification by

$$\begin{aligned}(\forall x A(x))^{SH} &:\equiv \forall x, \underline{a}\ \exists \underline{b}\ (\forall x A(x))_{SH}(x, \underline{a}, \underline{b}) \\ &:\equiv \forall x, \underline{a}\ \exists \underline{b}\ [A_{SH}(x, \underline{a}, \underline{b})],\end{aligned}$$

e) the interpretation of a bounded universal quantification by

$$\begin{aligned}(\forall x \in t\ A(x))^{SH} &:\equiv \forall \underline{a}\ \exists \underline{b}\ (\forall x \in t\ A(x))_{SH}(x, \underline{a}, \underline{b}) \\ &:\equiv \forall \underline{a}\ \exists \underline{b}\ [\forall x \in t\ A_{SH}(x, \underline{a}, \underline{b})].\end{aligned}$$

The formula A^{SH} is called the *upper SH-formula* or *interpretation* of A while the formula A_{SH} is called the *lower SH-formula* of A. The unbounded quantifications $\forall \underline{a} \exists \underline{b}$ are often called the *prefix* of A^{SH}.

Moreover, we will always explicitly write the universal and existential variables of the prefix in the lower SH-formula. We therefore tend to write $A_{SH}(\underline{a}, \underline{b})$ instead of simply A_{SH} in interpretations such as $A^{SH} \equiv \forall \underline{a} \exists \underline{b} A_{SH}(\underline{a}, \underline{b})$. This does not mean that $A_{SH}(\underline{a}, \underline{b})$ does not have any other variables besides the tuples $\underline{a}$ and $\underline{b}$. In fact, the formula $A_{SH}(\underline{a}, \underline{b})$ will also have the free variables of the original formula A, but these will be omitted unless stated otherwise. In case we wish to explicitly write them down, we will condense them into a single tuple of variables. In these cases, we will write $A_{SH}(\underline{x}, \underline{a}, \underline{b})$ and $\underline{x}$ would be said tuple of variables.

From here on, in order to avoid cumbersome notation we will use the same notation for tuples of variables and individual variables. Hence, instead of writing $\forall \underline{a} \exists \underline{b} A_{SH}(\underline{a}, \underline{b})$, we will write $\forall a \exists b A_{SH}(a,b)$. Unless stated otherwise, the variables in this section are all tuples of variables. An analogous remark holds for types and tuples of types.

Notation 2.1. It was mentioned in Remark 1.14 that base formulas would often be written in square brackets. Since lower SH-formulas are always base formulas, these will also be written between square brackets, as it is seen in the above definition. The use of square brackets is not mandatory, however it helps to visually distinguish the different constituents of the interpretation A^{SH}. We might omit the square brackets when these are not necessary. ⊠

Before making any further considerations about this definition, we look at two easy examples of interpretations of formulas to see how the definition is applied.

Example 2.1. We will compute the interpretation of $A \ \vee \ \forall x \in t \ B_b(x)$, where A is a formula and B_b is a base formula. To apply Definition 2.1, we need the interpretations of the formulas involved in the above disjunction. Let $A^{SH} \equiv \forall a \exists b A_{SH}(a,b)$. Moreover, since the first clause of Definition 2.1 states that the interpretation of a base formula is the base formula itself, we have $(\forall x \in t \ B_b(x))^{SH} \equiv (\forall x \in t \ B_b(x))_{SH} \equiv \forall x \in t \ B_b(x)$, as $\forall x \in t \ B_b(x)$ is a base formula. By applying clause b) of the same definition to A and $\forall x \in t \ B_b(x)$ we obtain the interpretation

$$(A \ \vee \ \forall x \in t \ B_b(x))^{SH} \ \equiv \ \forall a \exists b \ [A_{SH}(a,b) \ \vee \ \forall x \in t \ B_b(x)], \tag{2.3}$$

since clause b) regroups the universal and existential quantifications of A^{SH} and $(\forall x \in t \ B_b(x))^{SH}$ and creates a disjunction with the lower SH-formulas. ⊠

Example 2.2. We will compute the interpretation of $\forall y \in t \ \neg(\exists x \neg A(x) \wedge B(y))$, where A and B are formulas with corresponding interpretations given by $A(x)^{SH} \equiv \forall a \exists b A_{SH}(x,a,b)$ and $B(y)^{SH} \equiv \forall c \exists d B_{SH}(y,c,d)$. First, we need to rewrite $\forall y \in t \ \neg(\exists x \neg A(x) \wedge B(y))$, in order to only have primitive symbols. We will work with the equivalent formula $\forall y \in t \ (\forall x A(x) \vee \neg B(y))$. Applying 2.1.d) and 2.1.c) respectively to $\forall x A(x)$ and $\neg B(y)$ yields

$$(\forall x A(x))^{SH} \equiv \forall x, a \exists b \ A_{SH}(x,a,b), \tag{2.4}$$

$$(\neg B(y))^{SH} \equiv \forall g \exists c' \ [\exists c \in c' \ \neg B_{SH}(y,c,gc)]. \tag{2.5}$$

Applying now clause 2.1.b) using (2.4) and (2.5), we obtain

$$(\forall x A(x) \vee \neg B(y))^{SH} \ \equiv \ \forall x, a, g \ \exists b, c' \ [A_{SH}(x,a,b) \vee \exists c \in c' \ \neg B_{SH}(y,c,gc)] \tag{2.6}$$

and, finally, applying 2.1.e) to (2.6) gives the interpretation of $\forall y \in t \ (\forall x A(x) \vee \neg B(y))$ given by

$$\begin{aligned} & (\forall y \in t \ \neg(\exists x \neg A(x) \wedge B(y)))^{SH} \\ \equiv \quad & (\forall y \in t \ (\forall x A(x) \vee \neg B(y)))^{SH} \\ \equiv \quad & \forall x, a, g \ \exists b, c' \ [\forall y \in t \ (A_{SH}(x,a,b) \vee \exists c \in c' \ \neg B_{SH}(y,c,gc))]. \end{aligned} \tag{2.7}$$

⊠

We now make some remarks related to Definition 2.1. First, notice that a question that may arise at this point is, for example, whether $\neg A^{SH}$ means $\neg(A^{SH})$ or $(\neg A)^{SH}$ for some formula A.

Remark 2.1. *(Precedence of SH over logical symbols)* We will use the convention that SH (both upper and lower) has priority over all logical symbols. By this convention, $\neg A^{SH}$ means $\neg(A^{SH})$ (the negation of an interpretation) and not $(\neg A)^{SH}$ (the interpretation of a negation). To make the distinction more apparent, let A^{SH} be $\forall a \exists b A_{SH}(a,b)$ as in (2.2) from Definition 2.1. Then

$$\begin{aligned} \neg(A^{SH}) \ &\equiv \ \neg(\forall a \exists b \ A_{SH}(a,b)) \\ &\equiv \ \exists a \forall b \ \neg A_{SH}(a,b), \end{aligned}$$

while $(\neg A)^{SH} \equiv \forall f \exists a' \, [\exists a \in a' \, \neg A_{SH}(a, fa)]$. By this convention, $A \vee B^{SH}$ is $A \vee (B^{SH})$ instead of $(A \vee B)^{SH}$ and $\forall x \in t \, A_{SH}(x)$ is $\forall x \in t \, (A_{SH}(x))$ instead of $(\forall x \in t \, A(x))_{SH}$. ⊠

As for all terms in $\mathscr{L}_*^\omega$, the terms in Definition 2.1 are also typed. We now make two observations related to the types in Shoenfield's functional interpretation.

Remark 2.2. *(Types in Shoenfield's interpretation)* Let $\sigma, \tau, \rho, \gamma, \delta$ be types and let A, B be formulas such that $A^{SH} \equiv \forall a^\sigma \exists b^\tau A_{SH}(a, b)$ and $B^{SH} \equiv \forall c^\gamma \exists d^\delta B_{SH}(c, d)$. Moreover, let $x : \rho$ be a variable and $t : \rho^*$ a term in which x does not occur. Then clauses b) to e) from Definition 2.1 are typed as follows:

$$\begin{aligned}
(A \vee B)^{SH} &:\equiv \forall a^\sigma, c^\gamma \, \exists b^\tau, d^\delta \, [A_{SH}(a, b) \vee B_{SH}(c, d)], \\
(\neg A)^{SH} &:\equiv \forall f^{\sigma \to \tau} \, \exists a'^{\sigma^*} \, [\exists a \in a' \, \neg A_{SH}(a, (fa)^\tau)], \\
(\forall x \; A(x))^{SH} &:\equiv \forall x^\rho, a^\sigma \, \exists b^\tau \, [A_{SH}(x, a, b)], \\
(\forall x \in t \; A(x))^{SH} &:\equiv \forall a^\sigma \, \exists b^\tau \, [\forall x^\rho \in t^{\rho^*} \, A_{SH}(x, a, b)].
\end{aligned}$$

Notice that the terms and the types in the interpretations $(A \vee B)^{SH}$, $(\forall x A(x))^{SH}$ and $(\forall x \in t \; A(x))^{SH}$ are exactly the same as the ones in the interpretations of the original formulas A and B. However, this is not the case with the interpretation of the negation. In fact, the interpretation $(\neg A)^{SH}$ is the only case where it is necessary to create new terms with new types that were not originally in the interpretation of A, namely $f : \sigma \to \tau$ and $a' : \sigma^*$. As a consequence, it becomes necessary to elevate the types when computing the interpretation of any formula with a negation. ⊠

Remark 2.3. *(Types of existential variables)* In the definition of Shoenfield's functional interpretation, if A is a formula such that $A^{SH} \equiv \forall a^\sigma \exists b^\tau A_{SH}(a, b)$, then the universal variables a can have any type, while the existential variables b will always be of star type, that is, τ must be of the form ρ^* for some type ρ. This becomes apparent when considering the interpretation of the negation of a formula (refer to the previous remark), where the existential variable a' must be of type σ^*, as it represents a finite and non-empty set containing a. ⊠

Definition 2.1 is meant to be as general as possible: any formula of $\mathscr{L}_*^\omega$ can be interpreted using the five clauses of Definition 2.1. Moreover, as can be seen from this definition, the universal and existential variables, given respectively by a and b in an interpretation such as $A^{SH} \equiv \forall a \exists b A_{SH}(a, b)$, can be empty tuples of variables. The next remark investigates how empty tuples of variables affect the interpretations of formulas.

Remark 2.4. *(Interpretations of formulas with empty tuples)* Let A, B, C be formulas with corresponding interpretations

$$\begin{aligned}
A^{SH} &\equiv \forall a \exists b \; A_{SH}(a, b), \\
B^{SH} &\equiv \forall a \; B_{SH}(a), \\
C^{SH} &\equiv \exists b \; C_{SH}(b).
\end{aligned}$$

Notice that in the interpretation of B, the existential tuple b is empty while in the interpretation of C the universal tuple a is empty. The computation of interpretations of formulas with empty tuples is mostly straightforward. For example,

$$\begin{aligned}
(B \vee C)^{SH} &\equiv \forall a \exists b \; [B_{SH}(a) \vee C_{SH}(b)], \\
(\forall x \; B(x))^{SH} &\equiv \forall x, a \; B_{SH}(x, a), \\
(\forall x \in t \; C(x))^{SH} &\equiv \exists b \; [\forall x \in t \; C_{SH}(x, b)].
\end{aligned}$$

The interpretation of the negation of formulas with empty tuples is a bit less straightforward. For example, the interpretations of $\neg A$, $\neg B$ and $\neg C$ are given by

$$(\neg A)^{SH} \equiv \forall f \exists a' \, [\exists a \in a' \, \neg A_{SH}(a, fa)], \tag{2.8}$$
$$(\neg B)^{SH} \equiv \exists a' \, [\exists a \in a' \, \neg B_{SH}(a)], \tag{2.9}$$
$$(\neg C)^{SH} \equiv \forall b \, [\neg C_{SH}(b)]. \tag{2.10}$$

Notice that in $(\neg A)^{SH}$, the universal variable f accounts for the existential variable b of A^{SH}, while the existential variable a' accounts for the universal variable a of A^{SH}. Hence, in interpretations such as $(\neg B)^{SH}$, since b is an empty tuple in B^{SH}, so is the tuple f in $(\neg B)^{SH}$. Moreover, in interpretations such as $(\neg C)^{SH}$, since a is an empty tuple in C^{SH}, so is the tuple a' in $(\neg C)^{SH}$. ⊠

By Remark 2.2 one could think that only the negation clause plays a special role in the definition of Shoenfield's interpretation (Definition 2.1). The following remark discusses that the unbounded universal quantifier clause is likewise significant.

Remark 2.5. *(Creation of unbounded quantifiers from base formulas)* Let A_b be a base formula. Notice that only the clause on the unbounded universal quantifier from Definition 2.1 is able to transform a base formula such as A_b into a formula that is no longer base. Only with this clause is it possible to insert new universal variables to obtain formulas such as $\forall a A_{SH}(a)$. And only after this procedure is it possible to create existential variables by applying the negation clause of the same definition as to obtain $\exists a'[\exists a \in a' \, \neg A_{SH}(a)]$. Hence, without clause d) of Definition 2.1 base formulas would always remain base and the prefix would always have empty tuples. ⊠

Before moving on to new content, we give two more challenging examples that illustrate the application of Definition 2.1. These examples will also be used later on to illustrate the Soundness Theorem. Notice that in both examples we need to pay attention to two aspects:

1. Except for base formulas which are not affected by interpretations, if we want to interpret a formula which involves defined symbols, we will need to rewrite it in order to only have primitive symbols (this will change as soon as we introduce Proposition 2.1).
2. As is usually the case when working with logical formulas, it might be necessary to rename some of the variables, in order to avoid clashes of variables. For example, if we want to interpret the formula $(A \vee A)^{SH}$, where $A^{SH} \equiv \forall a \exists b A_{SH}(a, b)$, we will rename the variables a and b of the right-hand side A to c and d respectively and, as a consequence, have $(A \vee A)^{SH} \equiv \forall a, c \exists b, d \, [A_{SH}(a, b) \vee A_{SH}(c, d)]$.

Example 2.3. Let B be the formula given by

$$\forall x \in z \, \exists y \, A_b(x, y) \to \exists w \, \forall x \in z \, \exists y \in w \, A_b(x, y), \tag{2.11}$$

where $A_b(x, y)$ is a base formula. We want to compute the interpretation of the above formula, that is, we want to compute B^{SH}. We begin by rewriting B in order to have an equivalent formula only involving primitive symbols. We will use classical logic at will and, as stated above in point 1, we will not rewrite any base formula. Formula B can be rewritten as follows:

$$\begin{aligned} B &\equiv \forall x \in z \, \exists y \, A_b(x, y) \to \exists w \, \forall x \in z \, \exists y \in w \, A_b(x, y) \\ &\equiv \neg(\forall x \in z \, \exists y \, A_b(x, y)) \;\vee\; \exists w \, \forall x \in z \, \exists y \in w \, A_b(x, y) \\ &\equiv \neg(\forall x \in z \, \neg(\forall y \, \neg A_b(x, y))) \;\vee\; \neg(\forall w \, \neg(\forall x \in z \, \exists y \in w \, A_b(x, y))). \end{aligned} \tag{2.12}$$

In order to compute B^{SH}, we will first interpret the left-hand side and the right-hand side of disjunction (2.12) separately to then apply clause 2.1.b) and obtain B^{SH}. Notice that since $A_b(x,y)$ is a base formula, then so are the formulas $\neg A_b(x,y)$ and $\neg(\forall x \in z\, \exists y \in w\, A_b(x,y))$ on the left and on the right side of the disjunction (2.12), respectively.

For the left-hand side of (2.12), since $\neg A_b(x,y)$ is a base formula, then by 2.1.a) we have $(\neg A_b(x,y))^{SH} \equiv \neg A_b(x,y)$. Moreover, by applying clauses 2.1.d), c), e) and c) in this order and by referring to Remark 2.4, we obtain

$$\begin{aligned}
(\forall y\ \neg A_b(x,y))^{SH} &\equiv \forall y\ [\neg A_b(x,y)],\\
(\neg(\forall y\ \neg A_b(x,y)))^{SH} &\equiv \exists y'\ [\exists y \in y' \neg(\neg A_b(x,y))]\\
&\equiv \exists y'\ [\exists y \in y' A_b(x,y)],\\
(\forall x \in z\ \neg(\forall y\ \neg A_b(x,y)))^{SH} &\equiv \exists y'\ [\forall x \in z\ \exists y \in y'\ A_b(x,y)],\\
(\neg(\forall x \in z\ \neg(\forall y\ \neg A_b(x,y))))^{SH} &\equiv \forall y'\ [\neg(\forall x \in z\ \exists y \in y'\ A_b(x,y))]\\
&\equiv \forall y'\ [\exists x \in z\ \forall y \in y'\ \neg A_b(x,y))]. \qquad (2.13)
\end{aligned}$$

For the right-hand side, since $\neg(\forall x \in z\ \exists y \in w\ A_b(x,y))$ is a base formula, by clause 2.1.a),

$$(\neg(\forall x \in z\ \exists y \in w\ A_b(x,y)))^{SH} \equiv \neg(\forall x \in z\ \exists y \in w\ A_b(x,y)).$$

Moreover, by clauses 2.1.d) and c),

$$\begin{aligned}
(\forall w\ \neg(\forall x \in z\ \exists y \in w\ A_b(x,y)))^{SH} &\equiv \forall w\ [\neg(\forall x \in z\ \exists y \in w\ A_b(x,y))],\\
(\neg(\forall w\ \neg(\forall x \in z\ \exists y \in w\ A_b(x,y))))^{SH} &\equiv \exists w'\ [\exists w \in w'\ \neg\neg(\forall x \in z\ \exists y \in w\ A_b(x,y))]\\
&\equiv \exists w'\ [\exists w \in w'\ \forall x \in z\ \exists y \in w\ A_b(x,y)]. \qquad (2.14)
\end{aligned}$$

Finally, putting everything together by applying clause 2.1.b) and rewriting the formula in terms of the original defined symbols, we obtain the interpretation of B given by

$$\begin{aligned}
B^{SH} &\equiv (\ \forall x \in z\ \exists y\ A_b(x,y) \to \exists w\ \forall x \in z\ \exists y \in w\ A_b(x,y)\ )^{SH}\\
&\equiv (\ \neg(\forall x \in z\ \neg(\forall y\ \neg A_b(x,y)))\ \vee\ \neg(\forall w\ \neg(\forall x \in z\ \exists y \in w\ A_b(x,y)))\ )^{SH}\\
&\equiv \forall y' \exists w'\ [\exists x \in z \forall y \in y' \neg A_b(x,y)\ \vee\ \exists w \in w'\ \forall x \in z\ \exists y \in w\ A_b(x,y)]\\
&\equiv \forall y' \exists w'\ [\neg(\exists x \in z \forall y \in y' \neg A_b(x,y))\ \to\ \exists w \in w'\ \forall x \in z\ \exists y \in w\ A_b(x,y)]\\
&\equiv \forall y' \exists w'\ [\forall x \in z \exists y \in y'\ A_b(x,y)\ \to\ \exists w \in w'\ \forall x \in z\ \exists y \in w\ A_b(x,y)]. \qquad (2.15)
\end{aligned}$$

Notice that $B_{SH}(y', w') \equiv \forall x \in z \exists y \in y' A_b(x,y) \to \exists w \in w' \forall x \in z \exists y \in w\ A_b(x,y)$. ⊠

Example 2.4. *(Interpretation of the Bounded Axiom of Choice)* The next example is the interpretation of the Bounded Axiom of Choice $\mathsf{bAC}^{\omega}_{*}$ (Axiom 1.8) given by

$$\forall x \exists y\ A_b(x,y)\ \to\ \exists f \forall x \exists y \in fx\ A_b(x,y)$$

where A_b is a base formula. We begin by renaming some variables in order to avoid any confusion and we obtain $\forall x \exists y\ A_b(x,y)\ \to\ \exists f \forall a \exists b \in fa\ A_b(a,b)$. Once again, we rewrite the formula using primitive symbols (except for base formulas) as follows:

$$\begin{aligned}
&\forall x \exists y\ A_b(x,y) \to \exists f \forall a \exists b \in fa\ A_b(a,b)\\
&\equiv \neg(\forall x \exists y A_b(x,y)) \vee \exists f \forall a \exists b \in fa A_b(a,b)\\
&\equiv \neg(\forall x\ \neg(\forall y\ \neg A_b(x,y))) \vee \neg(\forall f\ \neg(\forall a \exists b \in fa\ A_b(a,b))). \qquad (2.16)
\end{aligned}$$

As in the previous example, we first interpret the left-hand side and the right-hand side of (2.16) separately before merging their interpretations together to find the interpretation of the Bounded Axiom of Choice using clause 2.1.b).

For the left-hand side of (2.16), since $A_b(x,y)$ is a base formula, then so is $\neg A_b(x,y)$. By clause 2.1.a), $(\neg A_b(x,y))^{SH} \equiv \neg A_b(x,y)$. By applying clauses 2.1.d), c), d), and c) and Remark 2.4 we have

$$\begin{aligned}
(\forall y\ \neg A_b(x,y))^{SH} &\equiv \forall y\ [\neg A_b(x,y)],\\
(\neg(\forall y \neg A_b(x,y)))^{SH} &\equiv \exists y'\ [\exists y \in y'\ \neg(\neg A_b(x,y))]\\
&\equiv \exists y'\ [\exists y \in y'\ A_b(x,y)],\\
(\forall x \neg(\forall y \neg A_b(x,y)))^{SH} &\equiv \forall x \exists y'\ [\exists y \in y'\ A_b(x,y)],\\
(\neg(\forall x \neg(\forall y \neg A_b(x,y))))^{SH} &\equiv \forall g \exists x'\ [\exists x \in x'\ \neg(\exists y \in gx\ A_b(x,y))]\\
&\equiv \forall g \exists x'\ [\exists x \in x'\ \forall y \in gx\ \neg A_b(x,y)]. \qquad (2.17)
\end{aligned}$$

For the right-hand side of (2.16), since $A_b(x,y)$ is a base formula, then so is $\exists b \in fa\ A_b(a,b)$ which then coincides with its interpretation, that is, $(\exists b \in fa\ A_b(a,b))^{SH} \equiv \exists b \in fa\ A_b(a,b))$. By applying 2.1.d), c), d), and c) and Remark 2.4 we obtain

$$\begin{aligned}
(\forall a\ \exists b \in fa\ A_b(a,b))^{SH} &\equiv \forall a\ [\exists b \in fa\ A_b(a,b))],\\
(\neg(\forall a\ \exists b \in fa\ A_b(a,b)))^{SH} &\equiv \exists a'\ [\exists a \in a' \neg(\exists b \in fa\ A_b(a,b)))]\\
&\equiv \exists a'\ [\exists a \in a' \forall b \in fa\ \neg A_b(a,b))],\\
(\forall f \neg(\forall a\ \exists b \in fa\ A_b(a,b)))^{SH} &\equiv \forall f \exists a'\ [\exists a \in a'\ \forall b \in fa\ \neg A_b(a,b))],\\
(\neg(\forall f \neg(\forall a\ \exists b \in fa\ A_b(a,b))))^{SH} &\equiv \forall \Phi \exists f'\ [\exists f \in f'\ \neg(\exists a \in \Phi f\ \forall b \in fa\ \neg A_b(a,b)))]\\
&\equiv \forall \Phi \exists f'\ [\exists f \in f'\ \forall a \in \Phi f\ \exists b \in fa\ A_b(a,b)))]. \qquad (2.18)
\end{aligned}$$

Finally, applying 2.1.b) to (2.17) and (2.18) yields the interpretation of the Bounded Axiom of Choice given by

$$\begin{aligned}
&(\forall x \exists b A_b(x,y)\ \rightarrow\ \exists f \forall a \exists b \in fa\ A_b(a,b))^{SH}\\
&\equiv (\neg(\forall x\ \neg(\forall y\ \neg A_b(x,y))) \vee \neg(\forall f\ \neg(\forall a \exists b \in fa\ A_b(a,b))))^{SH}\\
&\equiv \forall g, \Phi \exists x', f'\ [\exists x \in x' \forall y \in gx\ \neg A_b(x,y) \vee \exists f \in f' \forall a \in \Phi f \exists b \in fa\ A_b(a,b)]\\
&\equiv \forall g, \Phi \exists x', f'\ [\neg(\exists x \in x' \forall y \in gx\ \neg A_b(x,y)) \rightarrow \exists f \in f' \forall a \in \Phi f \exists b \in fa\ A_b(a,b)]\\
&\equiv \forall g, \Phi \exists x', f'\ [\forall x \in x' \exists y \in gx\ A_b(x,y) \rightarrow \exists f \in f' \forall a \in \Phi f \exists b \in fa\ A_b(a,b)]. \qquad (2.19)
\end{aligned}$$

⊠

As could be seen from the above examples, it is tedious to find the interpretations of formulas that use defined symbols. The next proposition states a straightforward method to interpret formulas involving defined symbols.

Proposition 2.1. (Interpretation of formulas with defined symbols) *Let A and B be formulas such that $A^{SH} \equiv \forall a \exists b\ A_{SH}(a,b)$ and $B^{SH} \equiv \forall c \exists d\ B_{SH}(c,d)$. Then,*

a) $(A \rightarrow B)^{SH} \equiv \forall f, c \exists a', d\ [\forall a \in a' A_{SH}(a, fa) \rightarrow B_{SH}(c,d)]$,

b) $(A \wedge B)^{SH} \equiv \forall \Phi, \Psi \exists f', g'\ [\exists f \in f'\ \exists g \in g'\ (\forall a \in \Phi fg A_{SH}(a, fa) \wedge \forall c \in \Psi fg B_{SH}(c, gc))]$,

c) $(\exists x A(x))^{SH} \equiv \forall \Phi \exists x', f'\ [\exists x \in x'\ \exists f \in f'\ \forall a \in \Phi xf\ A_{SH}(x,a,fa)]$,

d) $(\exists x \in t\ A(x))^{SH} \equiv \forall \Phi \exists f'\ [\exists f \in f'\ \exists x \in t\ \forall a \in \Phi f\ A_{SH}(x,a,fa)]$.

This proposition can be rewritten in a way that shows how interpretations of formulas involving defined symbols are "well-behaved" as can be seen in the next remark.

Proposition 2.2. (Proposition 2.1 - rewritten) *Let A and B be as above. Then,*

a) $(A \to B)^{SH} \equiv \forall f, c \exists a', d\ [\neg(\neg A)_{SH}(f, a') \to B_{SH}(c, d)]$,

b) $(A \wedge B)^{SH} \equiv \forall \Phi, \Psi \exists f', g'\ [\exists f \in f' \exists g \in g'\ (\neg(\neg A)_{SH}(f, \Phi fg) \wedge \neg(\neg B)_{SH}(g, \Psi fg))]$,

c) $(\exists x A(x))^{SH} \equiv \forall \Phi \exists x', f'\ [\exists x \in x' \exists f \in f'\ \neg(\neg A(x))_{SH}(f, \Phi x f)]$,

d) $(\exists x \in t\ A(x))^{SH} \equiv \forall \Phi \exists f'\ [\exists f \in f' \exists x \in t\ \neg(\neg A(x))_{SH}(f, \Phi f)]$.

Although Proposition 2.1 helps us deal with formulas with defined symbols, it is rather easy to lose oversight of the variables and terms involved in the interpretation process. Fortunately, the proof provides clarity on this matter. We now prove Proposition 2.1. For the proof of the rewritten proposition (Proposition 2.2) refer to Appendix C.

Proof of Proposition 2.1. Let A and B be formulas with interpretations given by

$$A^{SH} \equiv \forall a \exists b\ A_{SH}(a, b), \tag{2.20}$$
$$B^{SH} \equiv \forall c \exists d\ B_{SH}(c, d). \tag{2.21}$$

We begin by stating some of the interpretations that we need for the proof. Applying Definition 2.1.c) to A and B gives

$$(\neg A)^{SH} \equiv \forall f \exists a'\ [\exists a \in a'\ \neg A_{SH}(a, fa)], \tag{2.22}$$
$$(\neg B)^{SH} \equiv \forall g \exists c'\ [\exists c \in c'\ \neg B_{SH}(c, gc)]. \tag{2.23}$$

By applying Definition 2.1.d) to (2.22) we obtain

$$(\forall x\ \neg A(x))^{SH} \equiv \forall x, f \exists a'\ [\exists a \in a'\ \neg A_{SH}(x, a, fa)]. \tag{2.24}$$

Before computing the interpretations of $A \to B$, $A \wedge B$, $\exists x A(x)$, and of $\exists x \in t A(x)$, we will need to rewrite each formula using the primitive symbols.

Interpretation of the implication. Since $A \to B \equiv \neg A \vee B$, by applying Definition 2.1.b) to $\neg A$ and B using (2.22) and (2.21), we obtain

$$\begin{aligned}(A \to B)^{SH} &\equiv (\neg A \vee B)^{SH} \\ &\equiv \forall f, c \exists a', d\ [\exists a \in a' \neg A_{SH}(a, fa) \vee B_{SH}(c, d)] \\ &\equiv \forall f, c \exists a', d\ [\forall a \in a' A_{SH}(a, fa) \to B_{SH}(c, d)].\end{aligned} \tag{2.25}$$

Interpretation of the conjunction. Since $A \wedge B \equiv \neg(\neg A \vee \neg B)$, we first compute $(\neg A \vee \neg B)^{SH}$ before computing $(\neg(\neg A \vee \neg B))^{SH}$. By applying Definition 2.1.b) to (2.22) and (2.23) we have

$$(\neg A \vee \neg B)^{SH} \equiv \forall f, g \exists a', c'\ [\exists a \in a'\ \neg A_{SH}(a, fa) \vee \exists c \in c'\ \neg B_{SH}(c, gc)] \tag{2.26}$$

and, by Definition 2.1.c), we obtain

$$\begin{aligned}(A \wedge B)^{SH} &\equiv (\neg(\neg A \vee \neg B))^{SH} \\ &\equiv \forall \Phi, \Psi \exists f', g'\ [\exists f \in f' \exists g \in g'\ \neg\big(\exists a \in \Phi fg\ \neg A_{SH}(a, fa) \vee \exists c \in \Psi fg\ \neg B_{SH}(c, gc)\big)] \\ &\equiv \forall \Phi, \Psi \exists f', g'\ [\exists f \in f' \exists g \in g'\ \big(\forall a \in \Phi fg\ A_{SH}(a, fa) \wedge \forall c \in \Psi fg\ B_{SH}(c, gc)\big)].\end{aligned} \tag{2.27}$$

Interpretation of the unbounded existential quantification. Since $\exists x A(x) \equiv \neg(\forall x \neg A(x))$, we apply Definition 2.1.c) to (2.24) and we obtain

$$\begin{aligned}(\exists x A(x))^{SH} &\equiv (\neg(\forall x\ \neg A(x)))^{SH} \\ &\equiv \forall \Phi \exists x', f'\ [\exists x \in x'\ \exists f \in f'\ \neg\big(\exists a \in \Phi x f\ \neg A_{SH}(x, a, fa)\big)] \\ &\equiv \forall \Phi \exists x', f'\ [\exists x \in x'\ \exists f \in f'\ \forall a \in \Phi x f\ A_{SH}(x, a, fa)].\end{aligned} \tag{2.28}$$

Interpretation of the bounded existential quantification. Similarly to the interpretation of $\exists x A(x)$, since $\exists x \in t\ A(x) \equiv \neg(\forall x \in t\ \neg A(x))$, we first apply Definition 2.1.e) to (2.22)

$$(\forall x \in t\ \neg A(x))^{SH} \equiv \forall f \exists a' \left[\forall x \in t\ \exists a \in a'\ \neg A_{SH}(x, a, fa)\right] \tag{2.29}$$

and then, by applying Definition 2.1.c) to (2.29), we obtain

$$\begin{aligned}(\exists x \in t\ A(x))^{SH} &\equiv (\neg(\forall x \in t\ \neg A(x)))^{SH} \\ &\equiv \forall \Phi \exists f' \left[\exists f \in f'\ \neg \left(\forall x \in t\ \exists a \in \Phi f\ \neg A_{SH}(x, a, fa)\right)\right] \\ &\equiv \forall \Phi \exists f' \left[\exists f \in f'\ \exists x \in t\ \forall a \in \Phi f\ A_{SH}(x, a, fa)\right].\end{aligned} \tag{2.30}$$

This concludes the proof of the proposition. □

In order to illustrate the usefulness of Proposition 2.1 when computing formulas with defined symbols, we will revisit the interpretation of $\mathsf{bAC}^{\omega}_{*}$ (Example 2.4). To do so, we start off by making two remarks about the interpretation of existential formulas. Proposition 2.1.c) states that for any formula A with $A^{SH} \equiv \forall a \exists b A_{SH}(a, b)$ we have

$$(\exists x A(x))^{SH} \equiv \forall \Phi \exists x', f' \left[\exists x \in x' \exists f \in f'\ \forall a \in \Phi x f\ A_{SH}(x, a, fa)\right]. \tag{2.31}$$

We will now see how the interpretation of $\exists x\ A(x)$ changes when formula A is a base formula.

Remark 2.6. Let $A_b(x)$ be a base formula of $\mathscr{L}^{\omega}_{*}$. Then

$$(\exists x A_b(x))^{SH} \equiv \exists x' \left[\exists x \in x' A_b(x)\right]. \tag{2.32}$$

To see why, first notice that since $A_b(x)$ is a base formula, then so is $\neg A_b(x)$. Therefore, by Definition 2.1.a), $(\neg A_b(x))^{SH} \equiv (\neg A_b(x))_{SH} \equiv \neg A_b(x)$. Now using Definitions 2.1.c) and 2.1.d), we get $(\forall x\ \neg A_b(x))^{SH} \equiv \forall x\ \neg A_b(x)$ and, finally, referring to Remark 2.4,

$$\begin{aligned}(\exists x A_b(x))^{\mathrm{SH}} &\equiv (\neg(\forall x\ \neg A_b(x)))^{SH} \\ &\equiv\ \exists x' \left[\exists x \in x'\ \neg(\neg A_b(x))\right] \\ &\equiv\ \exists x' \left[\exists x \in x'\ A_b(x)\right].\end{aligned}$$ ⊠

Just as it was done in Remark 2.4, we discuss how empty tuples in A^{SH} affect the interpretation of $\exists x A(x)$.

Remark 2.7. Similarly to Remark 2.4, let $A(x), B(x), C(x)$ be formulas such that

$$\begin{aligned}A(x)^{SH} &\equiv \forall a \exists b A_{SH}(x, a, b), \\ B(x)^{SH} &\equiv \forall a B_{SH}(x, a), \\ C(x)^{SH} &\equiv \exists b C_{SH}(x, b).\end{aligned}$$

The interpretation $(\exists x A(x))^{SH}$ is given in 2.31. To compute the interpretations $(\exists x B(x))^{SH}$ and $(\exists x C(x))^{SH}$, notice that in the proof of Proposition 2.1.c) we have seen the following:

1. The universal variable Φ in $(\exists x A(x))^{SH}$ accounts for the existential variable a' in $(\forall x\ \neg A(x))^{SH}$, which in turn accounts for the universal variable a in A^{SH}.
2. The existential variable f' in $(\exists x A(x))^{SH}$ accounts for the universal variable f in $(\forall x\ \neg A(x))^{SH}$, which in turn accounts for the existential variable b in A^{SH}.

Taking this into consideration, since $B(x)^{SH}$ has no existential variable and since $C(x)^{SH}$ has no universal variable, we obtain

$$(\exists x B(x))^{SH} \equiv \forall \Phi \exists x' \ [\exists x \in x' \ \forall a \in \Phi x \ B_{SH}(x,a)], \tag{2.33}$$

$$(\exists x C(x))^{SH} \equiv \exists x', b' \ [\exists x \in x' \ \exists b \in b' \ C_{SH}(x,b)]. \tag{2.34}$$

A similar reasoning can be employed to find interpretations of the form $(\exists x \in t \ A(x))^{SH}$. ⊠

Using these remarks together with Proposition 2.1, it is now easier to compute the interpretation of formulas involving defined symbols. For example, the interpretation of $\mathsf{bAC}^{\omega}_{*}$ can be computed more directly as follows.

Example 2.5. *(Example 2.4 revisited)* As in Example 2.4, we begin by renaming the variables of the consequent of $\mathsf{bAC}^{\omega}_{*}$ as follows

$$\forall x \exists y \ A_b(x,y) \ \to \ \exists f \forall a \exists b \in fa \ A_b(a,b). \tag{2.35}$$

To interpret this implication, let $B :\equiv \forall x \exists y \ A_b(x,y)$ and $C :\equiv \exists f \forall a \exists b \in fa \ A(a,b)$, so that the computation of the interpretation of $\mathsf{bAC}^{\omega}_{*}$ corresponds to the computation of $(B \to C)^{SH}$. To apply Proposition 2.1.a), we need to compute B^{SH} and C^{SH}.

We begin by computing B^{SH}. Since $A_b(x,y)$ is a base formula, we apply Remark 2.6 to $A_b(x,y)$ and obtain $(\exists y A_b(x,y))^{SH} \equiv \exists y' \ [\exists y \in y' \ A_b(x,y)]$. By Definition 2.1.d), we then conclude

$$\begin{aligned} B^{SH} &\equiv \ (\forall x \exists y \ A_b(x,y))^{SH} \\ &\equiv \ \forall x \exists y' \ [\exists y \in y' \ A_b(x,y)]. \end{aligned} \tag{2.36}$$

Hence, $B_{SH}(x,y') \equiv \exists y \in y' \ A_b(x,y)$.

For the computation of C^{SH}, since $A_b(x,y)$ and $\exists b \in fa \ A_b(x,y)$ are base formulas, by Definition 2.1.a) and 2.1.d) we have

$$\begin{aligned} (\exists b \in fa \ A_b(a,b))^{SH} &\equiv \exists b \in fa \ A_b(a,b), \\ (\forall a \exists b \in fa \ A_b(a,b))^{SH} &\equiv \forall a \ [\exists b \in fa \ A_b(a,b)]. \end{aligned} \tag{2.37}$$

By using Remark 2.7 and (2.37), we have

$$\begin{aligned} C^{SH} &\equiv (\exists f \forall a \exists b \in fa \ A_b(a,b))^{SH} \\ &\equiv \forall \Phi \exists f' \ [\exists f \in f' \ \forall a \in \Phi f \ \exists b \in fa \ A_b(a,b)]. \end{aligned} \tag{2.38}$$

Hence, $C_{SH}(\Phi, f') \equiv \exists f \in f' \ \forall a \in \Phi f \ \exists b \in fa \ A_b(a,b)$.

Finally, by applying Proposition 2.1.a) to B and C, we obtain the interpretation of the Bounded Axiom of Choice as was already the case in Example 2.4

$$\begin{aligned} &(B \to C)^{SH} \\ &\equiv \forall g, \Phi \exists x', f' \ [\forall x \in x' \ B_{SH}(x, gx) \to C_{SH}(\Phi, f')] \\ &\equiv \forall g, \Phi \exists x', f' \ [\forall x \in x' \ \exists y \in gx \ A_b(x,y) \to \exists f \in f' \ \forall a \in \Phi f \ \exists b \in fa \ A_b(a,b)]. \end{aligned} \tag{2.39}$$

⊠

This example concludes Section 2 in which we described our herbrandized version of Shoenfield's functional interpretation. With the definitions and results from this section we can now discuss the Soundness Theorem, a generalization of Herbrand's Theorem and other associated results.

3 The Soundness Theorem and other results

In this section, we will study important and interesting results which can be shown using the framework around PL^ω_* that we have constructed in the previous sections. As will be seen, some of these results will require us to strengthen our theory with an extra logical principle. Doing so is a standard procedure when studying functional interpretations. In our case, we will strengthen our theory PL^ω_* with the Bounded Axiom of Choice bAC^ω_* from Section 1.2.3, and the first result we will thoroughly study is the *Soundness Theorem.*

3.1 The Soundness Theorem

To understand the statement of the Soundness Theorem, consider a formula A in $\mathscr{L}^\omega_*$ with its interpretation given by $A^{SH} \equiv \forall a \exists b A_{SH}(a, b)$. The theorem states that if A is provable in PL^ω_* strengthened with bAC^ω_*, then there exist terms that witness the existential quantifier b of A^{SH} in PL^ω_*. In other terms, from the proof of A in PL^ω_*+bAC^ω_*, it is possible to obtain a proof of $\forall a A_{SH}(a, ta)$ in PL^ω_*, where t is a term that we can algorithmically extract from the proof of A.

Before stating and proving the Soundness Theorem, there is an important tool we have to consider which is encoded in the *Monotonicity Lemma.*

Lemma 3.1. (*Monotonicity Lemma*) *Let A be a formula in $\mathscr{L}^\omega_*$ with interpretation given by $A^{SH} \equiv \forall a \exists b\ A_{SH}(a, b)$. Then*

$$\mathsf{PL}^\omega_* \vdash (b \subseteq b' \wedge A_{SH}(a, b)) \to A_{SH}(a, b').$$

In short, this lemma states that if the lower SH-formula A_{SH} holds for the existential variable b of A^{SH}, then it also holds for any "bigger" variable b' which contains b. We say that $A_{SH}(a, b)$ is *monotone* in the existential variable b.

Remark 3.1. *(Inclusion symbol)* We haven't yet defined the symbol $\subseteq$ which was just used in the statement of the Monotonicity Lemma. Even though semantically it may be clear what is meant by $b \subseteq b'$, note that it is an abbreviation for $\forall x\ (x \in b \to x \in b')$. The Monotonicity Lemma can be rewritten without the use of this abbreviation, accordingly. Notice that the inclusion symbol is also typed. It would therefore be more precise to write $b^{\sigma^*} \subseteq_{\sigma^*} b'^{\sigma^*}$, for some type σ. Naturally, both b and b' must have the same type because of the types involved in formulas $x^\sigma \in_\sigma b^{\sigma^*}$ and $x^\sigma \in_\sigma b'^{\sigma^*}$. ⊠

The proof of the lemma is done by induction on the complexity of the formula, that is, we will assume that the Monotonicity Lemma is valid for all the subformulas and then show the lemma for more complex formulas. For this, we need to consider all possible formulas in PL^ω_* and, hence, all the cases of the definition of Shoenfield's functional interpretation (Definition 2.1).

Proof of the Monotonicity Lemma. Let A and B be formulas in $\mathscr{L}^\omega_*$ with corresponding interpretations given by $A^{SH} \equiv \forall a \exists b\ A_{SH}(a, b)$ and $B^{SH} \equiv \forall c \exists d\ B_{SH}(c, d)$. Let b' and d' be tuples of variables and suppose that $b \subseteq b'$ and $d \subseteq d'$. We now check the five cases of Definition 2.1.

The first case is automatically proven because in interpretations of base formulas the tuples of variables are empty. Indeed, let C be a base formula. Since by definition we have $C \equiv C^{SH} \equiv C_{SH}$, we are done since by induction hypothesis we already have C_{SH}.

For the disjunction case, assume we have $(A \vee B)_{SH}(a, c, b, d)$, which is given by $A_{SH}(a, b) \vee B_{SH}(c, d)$. We want to show that we also have $(A \vee B)_{SH}(a, c, b', d')$, which is given by $A_{SH}(a,b') \vee B_{SH}(c,d')$. By induction hypothesis on A, since we have $A_{SH}(a,b)$, we have $A_{SH}(a,b')$. Analogously for B, since we have $B_{SH}(c,d)$, we also have $B_{SH}(c,d')$. Since we have both $A_{SH}(a,b')$ and $B_{SH}(c,d')$, we also have their disjunction which is what we wanted to show.

For the negation case, suppose that we already have $(\neg A)_{SH}(f,a')$ given by $\exists a \in a' \; \neg A_{SH}(a,fa)$. We want to show that given a variable a'' such that $a' \subseteq a''$ we also have $(\neg A)_{SH}(f,a'')$, which is given by $\exists a \in a'' \; \neg A_{SH}(a,fa)$. Since $a \in a'$ and $a' \subseteq a''$, we have $a \in a''$. We conclude that the a that works for $\exists a \in a' \; \neg A_{SH}(a,fa)$ also works for $\exists a \in a'' \; \neg A_{SH}(a,fa)$.

For the fourth case, suppose we have $(\forall x A(x))_{SH}(x,a,b)$ given by $A(x,a,b)$. We want to show that we have $(\forall x A(x))_{SH}(x,a,b')$ which is given by $A(x,a,b')$. This follows automatically, since by induction hypothesis on A, we already have $A(x,a,b')$, which is what we wanted to prove.

For the last case, assume we have $(\forall x \in t \; A(x))_{SH}(a,b)$ given by $\forall x \in t \; A(x,a,b)$. We want to show $(\forall x \in t \; A(x))_{SH}(a,b')$ which is given by $\forall x \in t \; A(x,a,b')$. Again, this follows automatically from the induction hypothesis on A. □

With the Monotonicity Lemma at hand, we can now state and prove the Soundness Theorem.

Theorem 3.1. (Soundness Theorem) *Let A be a sentence of $\mathscr{L}^\omega_*$ with interpretation given by $A^{SH} \equiv \forall a \exists b A_{SH}(a,b)$. If*

$$\mathsf{PL}^\omega_* + \mathsf{bAC}^\omega_* \vdash A,$$

then there exist closed terms t (of appropriate types) of $\mathscr{L}^\omega_*$ such that

$$\mathsf{PL}^\omega_* \vdash \forall a A_{SH}(a,ta).$$

Remark 3.2. *(Soundness Theorem for arbitrary formulas)* Although the theorem is stated for sentences in PL^ω_*, we will prove it for arbitrary formulas. In the case where we have a formula $A(x)$ that is not a sentence where x is the tuple of all its free variables, then the term t can depend on x and is therefore not necessarily closed anymore. In most cases though, whether the formula A does or does not have free variables will not drastically affect the proof. ⊠

In short, the Soundness Theorem states that if a sentence A is provable in $\mathsf{PL}^\omega_* + \mathsf{bAC}^\omega_*$, then PL^ω_* proves the particular case of A^{SH} where the existential variable is explicitly given by the term application ta. Notice that the verification does not require bAC^ω_* anymore. In order to prove the theorem we will have to show the existence of appropriate terms t, which will be defined using λ-notation. To make this process clear, we will now apply the Soundness Theorem to two of our previous examples. In these two examples, we will write the terms which are affected by the Soundness Theorem in bold to make the process easier to follow. These examples are a good start-off to better understand the upcoming proof.

Example 3.1. *(Continuation of Example* 2.3) In Example 2.3 we computed the interpretation of the formula

$$B \;\equiv\; \forall x \in z \; \exists y \; A_b(x,y) \;\to\; \exists w \; \forall x \in z \; \exists y \in w \; A_b(x,y)$$

where we found its interpretation to be given by

$$\begin{aligned} B^{SH} \;&\equiv\; \forall y' \; \exists \boldsymbol{w'} \; [\forall x \in z \; \exists y \in y' \; A_b(x,y) \;\to\; \exists w \in \boldsymbol{w'} \; \forall x \in z \; \exists y \in w \; A_b(x,y)] \\ &\equiv\; \forall y' \; \exists \boldsymbol{w'} \; B_{SH}(y', \boldsymbol{w'}). \end{aligned}$$

To apply the Soundness Theorem we need to define a term t such that the application $\boldsymbol{ty'}$ witnesses the existential variable $\boldsymbol{w'}$ in B^{SH}.

In other words, we need to check that PL^ω_* proves $\forall y' B_{SH}(y', \boldsymbol{ty'})$, which is given by

$$\forall y' \; [\forall x \in z \; \exists y \in y' \; A_b(x,y) \;\rightarrow\; \exists w \in \boldsymbol{ty'} \; \forall x \in z \; \exists y \in w \; A_b(x,y)]. \tag{3.1}$$

By defining the term t as $t := \lambda y'.\mathfrak{s}y'$, then $\boldsymbol{ty'} = (\lambda y'.\mathfrak{s}y')y' = \boldsymbol{\mathfrak{s}y'}$ and formula (3.1) becomes

$$\forall y' \; [\forall x \in z \; \exists y \in y' \; A_b(x,y) \;\rightarrow\; \exists w \in \boldsymbol{\mathfrak{s}y'} \; \forall x \in z \; \exists y \in w \; A_b(x,y)], \tag{3.2}$$

which is provable in PL^ω_*. To understand why it is provable, recall that by Axiom AxS_1 (Axiom 1.6), writing $w \in \boldsymbol{\mathfrak{s}y'}$ yields $w = y'$[10] and, as a consequence, formula (3.2) can be rewritten as

$$\forall y' \; [\forall x \in z \; \exists y \in y' \; A_b(x,y) \;\rightarrow\; \forall x \in z \; \exists y \in y' \; A_b(x,y)]. \tag{3.3}$$

Since the antecedent and the consequent of the implication are the same, the formula (3.3) is provable in PL^ω_*. ⊠

Example 3.2. *(Continuation of Example* 2.4) In Example 2.4 we computed the interpretation of the Bounded Axiom of Choice

$$B \;\equiv\; \forall x \exists y \; A_b(x,y) \rightarrow \exists f \forall x \exists y \in fx \; A_b(x,y)$$

where, after renaming the variables of the consequent (renaming x as a and y as b), its interpretation was given by

$$\begin{aligned} B^{SH} \;&\equiv\; \forall g, \Phi \exists \boldsymbol{x'}, \boldsymbol{f'} \; [\forall x \in \boldsymbol{x'} \; \exists y \in gx \; A_b(x,y) \rightarrow \exists f \in \boldsymbol{f'} \; \forall a \in \Phi f \; \exists b \in fa \; A_b(a,b)] \\ &\equiv\; \forall g, \Phi \exists \boldsymbol{x'}, \boldsymbol{f'} \; B_{SH}(g, \Phi, \boldsymbol{x'}, \boldsymbol{f'}). \end{aligned}$$

As there are two existential variables in B^{SH}, we need to define two terms p and t (one for each existential variable) that, applied to the universal variables g and Φ, witness the existential variables $\boldsymbol{x'}$ and $\boldsymbol{f'}$. Hence, our goal is to define p and t in such a way that PL^ω_* proves $\forall g, \Phi B_{SH}(g, \Phi, \boldsymbol{pg\Phi}, \boldsymbol{tg\Phi})$. Since $\forall g, \Phi B_{SH}(g, \Phi, \boldsymbol{pg\Phi}, \boldsymbol{tg\Phi})$ is given by

$$\forall g, \Phi \; [\forall x \in \boldsymbol{pg\Phi} \; \exists y \in gx \; A_b(x,y) \rightarrow \exists f \in \boldsymbol{tg\Phi} \; \forall a \in \Phi f \; \exists b \in fa \; A_b(a,b)]$$

if we define $p := \lambda g, \Phi.\Phi g$ and $t := \lambda g, \Phi.\mathfrak{s}g$, we then have $\boldsymbol{pg\Phi} = (\lambda g, \Phi.\Phi g)g\Phi = \boldsymbol{\Phi g}$ and $\boldsymbol{tg\Phi} = (\lambda g, \Phi.\mathfrak{s}g)g\Phi = \boldsymbol{\mathfrak{s}g}$. We see that these definitions work since we then have

$$\forall g, \Phi \; [\forall x \in \boldsymbol{\Phi g} \; \exists y \in gx \; A_b(x,y) \rightarrow \exists f \in \boldsymbol{\mathfrak{s}g} \; \forall a \in \Phi f \; \exists b \in fa \; A_b(a,b)] \tag{3.4}$$

and, by Axiom AxS_1, we can rewrite $f \in \boldsymbol{\mathfrak{s}g}$ as $f = g$ and formula (3.4) can be rewritten as

$$\forall g, \Phi \; [\forall x \in \boldsymbol{\Phi g} \; \exists y \in gx \; A_b(x,y) \rightarrow \; \forall a \in \Phi g \; \exists b \in ga \; A_b(a,b)]. \tag{3.5}$$

Just like in the previous example, the antecedent and the consequent are the same after going back to the original variable names and hence it is provable in PL^ω_*. ⊠

After these first examples, we will focus on the proof of the Soundness Theorem. The proof will be done by induction on the proof of A. Due to this, we need to take into account all possible axioms and rules that could be used in the proof of A, *i.e.*, we will need to prove the Soundness Theorem for every axiom and every inference rule in PL^ω_*.

When proving the theorem for the axioms, we will simply need to define appropriate witnesses for the existential variables (analogously to the two examples we just discussed). For the inference rules, since by induction hypothesis we will already have proven the theorem for the premises, we will have to construct the witnesses for the conclusion based on the witnesses we have for the premises.

[10] It might be helpful to remember that $\mathfrak{s}y'$ means $\{y'\}$ from a semantical point of view.

Because this must be done for every axiom and every inference rule, the proof becomes quite extensive. However, it is not overly complex and is, in most cases, easy to follow. The next remark will be useful in the simpler cases of the proof, namely when the axioms are universal closures of base formulas.

Remark 3.3. *(Soundness Theorem for universal closures of base formulas)* Various axioms from Section 1.2.2 were universal closures of base formulas. This will now prove to be useful since for these axioms the Soundness Theorem is automatically proven. In fact, since base formulas are their own interpretation, when a formula is the universal closure of a base formula, it is of the form $\forall x A_b(x)$ and hence there are no existential variables to witness.$\boxtimes$

As was seen in the previous two examples, the first step in the proof will always be to compute the interpretations we need. After having done so, we start defining the appropriate witnesses and then check that the witnessed formulas are provable in PL^ω_*.

Proof of the Soundness Theorem. We will need to check all the axioms from Section 1.2.2, together with the Bounded Axiom of Choice from Section 1.2.3. We begin by considering the axioms of Section 1.2.2 that are universal closures of base formulas. By Remark 3.3, for those axioms we merely need to check that these are indeed universal closures of *base* formulas. Recall that formulas of the form $p =_\sigma q$, where p and q are terms, are called equations of type σ and are, by definition, atomic formulas and, therefore, also base formulas.

- *Equality axioms* AxE_1 *and* AxE_2

Axiom AxE_1 is the universal closure of the equation $x = x$ and hence the universal closure of a base formula. Axiom AxE_2 is the universal closure of $x = y \wedge A_b(x) \to A_b(y)$. Since $x = y$, $A_b(x)$ and $A_b(y)$ are base formulas, by Remark 1.10 so is the whole formula. According to Remark 3.3, there is nothing to prove for either AxE_1 or AxE_2.

- *Axioms for the combinators* AxC_1 *and* AxC_2

Axioms AxC_1 and AxC_2 are the universal closures of the equations $\Pi xy = x$ and $\Sigma xyz = xz(yz)$, respectively, and are therefore both base formulas. Once again, by Remark 3.3 there is nothing to prove.

- *Primary axioms for the star constants* AxP_1 *and* AxP_2

As for the combinator axioms, the primary axioms for the star constants are the universal closures of the equations $\bigcup(\mathfrak{s}x)y = yx$ and $\bigcup(\cup xy)z = \cup(\bigcup xz)(\bigcup yz)$, respectively. Since equations are base formulas, by Remark 3.3 there is nothing to prove.

- *Secondary axioms for the star constants* AxS_1, AxS_2, AxS_3 *and* AxS_4

Each of the secondary axioms for the star constants is the universal closure of equivalences between base formulas. By Remark 1.10 such equivalences are base formulas and, by Remark 3.3, there is nothing to prove. For example, Axiom AxS_1 is the universal closure of $x \in \mathfrak{s}y \leftrightarrow x = y$. Since $x \in \mathfrak{s}y$ and $x = y$ are two base formulas, the equivalence is also a base formula. Analogously with axioms AxS_2, AxS_3 and AxS_4 (refer to Remark 1.15 for the latter).

- *Axiom for the bounded universal quantifier* AxU

Axiom AxU requires a bit more work since it is not the universal closure of a base formula. It is the universal closure of $\forall x \in t\ A(x) \leftrightarrow \forall x(x \in t\ \to A(x))$ which consists of the implications

$$\forall x \in t\ A(x) \to \forall y(y \in t \to A(y)), \tag{3.6}$$

$$\forall y(y \in t \to A(y)) \to \forall x \in t\ A(x), \tag{3.7}$$

where we used an appropriate change of variables. Let $A(x)^{SH} \equiv \forall a \exists b A_{SH}(a,b,x)$ and let $A(y)^{SH} \equiv \forall \alpha \exists \beta A_{SH}(\alpha,\beta,y)$. By Definition 2.1 we have

$$(\forall x \in t\ A(x))^{SH} \equiv \forall a \exists b\ [\forall x \in t\ A_{SH}(a,b,x)], \tag{3.8}$$

$$(\forall y (y \in t \to A(y)))^{SH} \equiv \forall y, \alpha \exists \beta\ [y \in t \to A_{SH}(\alpha,\beta,y)]. \tag{3.9}$$

To interpret formula (3.6), we apply Proposition 2.1 using (3.8) and (3.9) to obtain the interpretation of $\forall x \in t\ A(x) \to \forall y(y \in t \to A(y))$ given by

$$\forall f, y, \alpha \exists a', \beta\ [\forall a \in a'\ \forall x \in t\ A_{SH}(a, fa, x) \to (y \in t \to A_{SH}(\alpha,\beta,y))]. \tag{3.10}$$

We need to show the existence of terms p and q such that

$$\forall f, y, \alpha\ [\forall a \in pfy\alpha\ \forall x \in t\ A_{SH}(a, fa, x) \to (y \in t \to A_{SH}(\alpha, qfy\alpha, y))] \tag{3.11}$$

is provable in PL^{ω}_{*}. By rewriting this formula equivalently as

$$\forall f, y, \alpha\ [(\forall a \in pfy\alpha\ \forall x \in t\ A_{SH}(a, fa, x) \land y \in t) \to A_{SH}(\alpha, qfy\alpha, y)] \tag{3.12}$$

and by defining $p := \lambda f, y, \alpha.\mathfrak{s}\alpha$ and $q := \lambda f, y, \alpha.f\alpha$ we are done. In fact, since $pfy\alpha = \mathfrak{s}\alpha$, then $a \in \mathfrak{s}\alpha$ and, by axiom AxS_1, we have $a = \alpha$. Moreover, since $qfy\alpha = f\alpha$ we conclude

$$(\forall x \in t\ A_{SH}(\alpha, f\alpha, x) \land y \in t) \to A_{SH}(\alpha, f\alpha, y), \tag{3.13}$$

which is provable in PL^{ω}_{*}.

To interpret formula (3.7), we apply Proposition 2.1 using (3.9) and (3.8) to obtain the interpretation of $\forall y(y \in t \to A(y)) \to \forall x \in t\ A(x)$, which is then given by

$$\forall f, a \exists y', \alpha', b\ [\forall y \in y'\ \forall \alpha \in \alpha'\ (y \in t \to A_{SH}(\alpha, fy\alpha, y)) \ \to\ \forall x \in t\ A_{SH}(a,b,x))]. \tag{3.14}$$

We need to show the existence of terms p, q and r such that PL^{ω}_{*} proves

$$\forall f, a\ [\forall y \in pfa\ \forall \alpha \in qfa\ (y \in t \to A_{SH}(\alpha, fy\alpha, y)) \ \to\ \forall x \in t\ A_{SH}(a, rfa, x))]. \tag{3.15}$$

We define $p := \lambda f, a.t$, $q := \lambda f, a.\mathfrak{s}a$ and $r := \lambda f, a.\bigcup_{z\in t} fza$ as to obtain $pfa = t$, $qfa = \mathfrak{s}a$ and $rfa = \bigcup_{z \in t} fza$. First, notice that since $\alpha \in \mathfrak{s}a$, by Axiom AxC_1 we have $\alpha = a$. Second, since $y \in t$, we have

$$fya \subseteq \bigcup_{z \in t} fza = rfa \tag{3.16}$$

and hence, by the Monotonicity Lemma applied to $A_{SH}(a, fya, y)$ we finally obtain

$$\forall y \in t\ (y \in t \to A_{SH}(a, rfa, y)) \to \forall x \in t\ A_{SH}(a, rfa, x). \tag{3.17}$$

- *Shoenfield's Calculus*

To prove the axioms and the rules of Shoenfield's Calculus, let A, B and C be formulas of $\mathscr{L}^{\omega}_{*}$ with corresponding interpretations

$$A^{SH} \equiv \forall a \exists b\ A_{SH}(a,b), \tag{3.18}$$

$$B^{SH} \equiv \forall c \exists d\ B_{SH}(c,d), \tag{3.19}$$

$$C^{SH} \equiv \forall u \exists v\ C_{SH}(u,v). \tag{3.20}$$

In cases where we need to explicitly write the free variables, we will assume x to have all free variables of A and write $A(x)^{SH} \equiv \forall a \exists b A_{SH}(a,b,x)$. In cases where the formula A appears more than once (as is the case, for example, for the Excluded Middle) we will use the change of variables $A^{SH} \equiv \forall \alpha \exists \beta\ A_{SH}(\alpha,\beta)$.

1. *Excluded Middle:* $\neg A \vee A$
For the interpretation of $\neg A \vee A$, since $(\neg A)^{SH} \equiv \forall f \exists a' \, [\exists a \in a' \, \neg A_{SH}(a, fa)]$ we obtain

$$(\neg A \vee A)^{SH} \equiv \forall f, \alpha \exists a', \beta \, [\exists a \in a' \, \neg A_{SH}(a, fa) \vee A_{SH}(\alpha, \beta)]. \tag{3.21}$$

In order to prove the theorem for the excluded middle, we need to define terms p and q (for the existential variables a' and β, respectively) such that $\forall f, \alpha \, (\neg A \vee A)_{SH}(f, \alpha, pf\alpha, qf\alpha)$ is provable in PL^{ω}_{*}, that is,

$$\mathsf{PL}^{\omega}_{*} \vdash \forall f, \alpha \, [\exists a \in pf\alpha \, \neg A_{SH}(a, fa) \vee A_{SH}(\alpha, qf\alpha)]. \tag{3.22}$$

By defining terms p and q as

$$p := \lambda f, \alpha . \mathfrak{s}\alpha,$$
$$q := \lambda f, \alpha . f\alpha,$$

we get the appropriate terms. Indeed, since $pf\alpha = \mathfrak{s}\alpha$ and $qf\alpha = f\alpha$, we can rewrite (3.22) as

$$\forall f, \alpha \, [\exists a \in \mathfrak{s}\alpha \, \neg A_{SH}(a, fa) \vee A_{SH}(\alpha, f\alpha)]. \tag{3.23}$$

Since, by Axiom AxS_1, if a belongs to $\mathfrak{s}\alpha$ then a must be α, then (3.23) is actually $\forall f, \alpha \, [\neg A_{SH}(\alpha, f\alpha) \vee A_{SH}(\alpha, f\alpha)]$, which is simply the Excluded Middle Rule in PL^{ω}_{*}.

2. *Substitution:* $\forall x A(x) \to A(t)$
To interpret $\forall x A(x) \to A(t)$, since

$$(\forall x A(x))^{SH} \equiv \forall x, a \exists b \; A_{SH}(x, a, b), \tag{3.24}$$

we use Proposition 2.1 and obtain the interpretation of the implication given by

$$(\forall x A(x) \to A(t))^{SH} \equiv \forall f, \alpha \exists x', a', \beta \, [\forall x \in x' \, \forall a \in a' \; A_{SH}(x, a, fxa) \to A_{SH}(t, \alpha, \beta)]. \tag{3.25}$$

We need to define terms p, q and r for the existential variables x', a' and β, such that

$$\mathsf{PL}^{\omega}_{*} \vdash \forall f, \alpha \, [\forall x \in pf\alpha \, \forall a \in qf\alpha \; A_{SH}(x, a, fxa) \to A_{SH}(t, \alpha, rf\alpha)]. \tag{3.26}$$

Defining them as

$$p := \lambda f, \alpha . \mathfrak{s}t,$$
$$q := \lambda f, \alpha . \mathfrak{s}\alpha,$$
$$r := \lambda f, \alpha . ft\alpha,$$

that is, $pf\alpha = \mathfrak{s}t$, $qf\alpha = \mathfrak{s}\alpha$ and $rf\alpha = ft\alpha$, by Axiom AxS_1 we have $x = t$ and $a = \alpha$. Replacing these in (3.26), we get $\forall f, \alpha \, [A_{SH}(t, \alpha, ft\alpha) \to A_{SH}(t, \alpha, ft\alpha)]$, which proves the theorem for the second axiom scheme.

Now that we have taken care of the excluded middle and the substitution axiom schema, we continue analogously for the proof of the five inference rules.

3. *Expansion:* $A \Rightarrow B \vee A$
Since the interpretation of the disjunction $B \vee A$ is given by

$$(B \vee A)^{SH} \equiv \forall c, a \exists d, b \, [B_{SH}(c, d) \vee A_{SH}(a, b)] \tag{3.27}$$

we need to define terms p and q such that

$$\forall c, a \, [B_{SH}(c, pca) \vee A_{SH}(a, qca)] \tag{3.28}$$

is provable in PL^{ω}_{*}. Since, by induction hypothesis, we have proved the theorem for A, we have already proved that there exists a term t such that $\forall a A_{SH}(a, ta)$ is provable in PL^{ω}_{*}. Hence, it is enough to define $q := \lambda c, a.ta$, so that $qca = ta$. Indeed, formula (3.28) then becomes $\forall c, a \, [B_{SH}(c, pca) \vee A_{SH}(a, ta)]$, which is provable in PL^{ω}_{*} since we already have $\forall a A_{SH}(a, ta)$.

We are done if we are able to show the existence of the term p. Since p needs to be defined such that pca is a valid term application and since we made no assumptions on the types of c and a, we need to provide a definition of p for any possible type. For this, it suffices to refer to Remark 1.18 that stated that there exists at least one term for every possible type. Take k to be a constant of $\mathscr{L}$ of appropriate type as in Remark 1.18. We now simply define the term p as $p := \lambda c, a.k$ (*i.e.* $pca = k$) and (3.28) becomes $\forall c, a\ [B_{SH}(c,k) \vee A_{SH}(a,ta)]$, which concludes this part of the proof, since by induction hypothesis we already had $\forall a A_{SH}(a,ta)$.

4. *Contraction:* $A \vee A \Rightarrow A$
Using the change of variables mentioned at the beginning of the proof, we have

$$(A \vee A)^{SH} \equiv \forall a, \alpha \exists b, \beta\ [A_{SH}(a,b) \vee A_{SH}(\alpha,\beta)]. \tag{3.29}$$

Since we have already proven the theorem for $A \vee A$, by induction hypothesis there exist terms p and q such that

$$\forall a, \alpha\ [A_{SH}(a, pa\alpha) \vee A_{SH}(\alpha, qa\alpha)]. \tag{3.30}$$

Given that a and α are of the same type and since (3.30) is valid for all a and α, considering the particular case where we let $\alpha = a$ we obtain

$$\forall a\ [A_{SH}(a, paa) \vee A_{SH}(a, qaa)]. \tag{3.31}$$

Our goal is to define a term t that witnesses the existential variable b in A^{SH} such that

$$\mathsf{PL}^{\omega}_{*} \vdash \forall a\ A_{SH}(a, ta).$$

Let t be defined by

$$t := \lambda a.(\cup(paa)(qaa)),$$

that is, $ta = \cup(paa)(qaa)$. To make this part of the proof more easily readable, we will rely on the set-theoretic notation introduced in Remark 1.6. By doing so, we can write $ta = paa \cup qaa$. There are two cases to consider. For the first case assume $A_{SH}(a, paa)$ in (3.31). Since $paa \subseteq paa \cup qaa = ta$, by the Monotonicity Lemma we also have $A_{SH}(a, ta)$. For the second case, assume $A_{SH}(a, qaa)$ in (3.31). Since $qaa \subseteq paa \cup qaa = ta$, by the Monotonicity Lemma we also have $A_{SH}(a, ta)$. In both cases we obtain $\forall a[A_{SH}(a, ta)]$ which concludes this part of the proof.

5. *Associativity:* $A \vee (B \vee C) \Rightarrow (A \vee B) \vee C$
For the associativity rule, we need the following interpretations:

$$(B \vee C)^{SH} \equiv \forall c, u \exists d, v\ [B_{SH}(c,d) \vee C_{SH}(u,v)], \tag{3.32}$$

$$(A \vee (B \vee C))^{SH} \equiv \forall a, c, u \exists b, d, v\ [A_{SH}(a,b) \vee (B_{SH}(c,d) \vee C_{SH}(u,v))], \tag{3.33}$$

$$(A \vee B)^{SH} \equiv \forall a, c \exists b, d\ [A_{SH}(a,b) \vee B_{SH}(c,d)], \tag{3.34}$$

$$((A \vee B) \vee C)^{SH} \equiv \forall a, c, u \exists b, d, v\ [(A_{SH}(a,b) \vee B_{SH}(c,d)) \vee C_{SH}(u,v)]. \tag{3.35}$$

By induction hypothesis, we have shown the theorem for $A \vee (B \vee C)$, and therefore there exist terms p, q and r such that

$$\forall a, c, u\ [A_{SH}(a, pacu) \vee (B_{SH}(c, qacu) \vee C_{SH}(u, racu))]. \tag{3.36}$$

For $(A \vee B) \vee C$ we need to find terms p', q' and r' such that

$$\forall a, c, u\ [(A_{SH}(a, p'acu) \vee B_{SH}(c, q'acu)) \vee C_{SH}(u, r'acu)]. \tag{3.37}$$

The terms that will work for $(A \vee B) \vee C$ are precisely those that worked for $A \vee (B \vee C)$. It suffices to define $p' := p$, $q' := q$ and $r' := r$.

6. *Cut:* $A \vee B, \neg A \vee C \Rightarrow B \vee C$
Let $(A \vee B)^{SH}$ and $(B \vee C)^{SH}$ be as in the previous rule and notice that

$$(\neg A \vee C)^{SH} \equiv \forall f, u \exists a', v\ [\exists a \in a'\ \neg A_{SH}(a, fa) \vee C_{SH}(u, v)]. \tag{3.38}$$

Since by induction hypothesis we have deduced the theorem for $A \vee B$ and $\neg A \vee C$, we have the following assumptions:

Assumption 1. For $A \vee B$, there exist terms t and q such that

$$\forall a, c\ [A_{SH}(a, tac) \vee B_{SH}(c, qac)]. \tag{3.39}$$

Assumption 2. For $\neg A \vee C$, there exist terms r and s such that

$$\forall f, u\ [\exists a \in rfu\ \neg A_{SH}(a, fa) \vee C_{SH}(u, sfu)]. \tag{3.40}$$

Our goal is to construct terms j and l (based on the terms t, q, r and s) to witness the conclusion of the cut rule so that

$$\mathsf{PL}^{\omega}_{*} \vdash \forall c, u\ [B_{SH}(c, jcu) \vee C_{SH}(u, lcu)]. \tag{3.41}$$

From here on, we will once again rely on the set-theoretic notation for the star constants to make the proof more easily readable. Let j and l be the terms defined as follows[11]

$$jcu := \bigcup_{w \in r(\lambda a.tac)u} qwc, \tag{3.42}$$

$$lcu := s(\lambda a.tac)u. \tag{3.43}$$

We need to show that these definitions allow us to conclude (3.41). To do so, let c and u be given. Our goal is now to show $B_{SH}(c, jcu)$ or $C_{SH}(u, lcu)$.

We begin by noticing that, since the matrix of (3.40) is valid for all f and u, we can choose f to be given by $\lambda a.tac$, as both terms f and $\lambda a.tac$ have the same type (refer to Remark 3.4 after the proof). In order to avoid any confusion, we apply a change of variables and replace f by $\lambda x.txc$ instead of $\lambda a.tac$ in (3.40) so that

$$\exists a \in r(\lambda x.txc)u\ \neg A_{SH}(a, (\lambda x.txc)a) \vee C_{SH}(u, s(\lambda x.txc)u), \tag{3.44}$$

which can be simplified to

$$\exists a \in r(\lambda x.txc)u\ \neg A_{SH}(a, tac) \vee C_{SH}(u, lcu) \tag{3.45}$$

since $(\lambda x.txc)a$ is tac and $s(\lambda x.txc)u$ is lcu by definition of l. Regarding (3.45), we will distinguish two cases to conclude the proof.

Case 1. If we assume the right-hand side of disjunction (3.45), we trivially have $C_{SH}(u, lcu)$.

Case 2. If we assume the left-hand side of (3.45), we have $B_{SH}(c, jcu)$. Indeed, assume

$$\exists a \in r(\lambda x.txc)u\ \neg A_{SH}(a, tac) \tag{3.46}$$

and call a_0 a witness for (3.46), yielding $\neg A_{SH}(a_0, ta_0c)$. Then, by Assumption 1, we must have $B_{SH}(c, qa_0c)$. Second, since $a_0 \in r(\lambda x.txc)u$, by definition of j we have

$$qa_0c \subseteq \bigcup_{w \in r(\lambda x.txc)u} qwc = jcu.$$

Hence, $qa_0c \subseteq jcu$. By the Monotonicity Lemma, since we have $qa_0c \subseteq jcu$ and $B_{SH}(c, qa_0c)$, we conclude $B_{SH}(c, jcu)$.

[11] In case we would not want to rely on the set-theoretic notation, the term j would be defined as being $\lambda c, u. \bigcup (r(\lambda a.tac)u)qc$ and hence $jcu = \bigcup (r(\lambda a.tac)u)qc$.

In Case 1, we have shown $C_{SH}(u, lcu)$ while in Case 2 we have shown $B_{SH}(c, jcu)$. Joining both cases, we have shown $B_{SH}(c, jcu)$ or $C_{SH}(u, lcu)$, which proves (3.41) and concludes the proof of the Cut Rule.

7. $\forall$-*introduction:* $A(x) \vee B \Rightarrow \forall x A(x) \vee B$, provided x does not occur free in B.
Let the interpretation of $\forall x A(x)$ be given as in (3.24) and consider the interpretations

$$(A(x) \vee B)^{SH} \equiv \forall a, c \exists b, d \; [A_{SH}(x, a, b) \vee B_{SH}(c, d)], \tag{3.47}$$

$$(\forall x A(x) \vee B)^{SH} \equiv \forall x, a, c \exists b, d \; [A_{SH}(x, a, b) \vee B_{SH}(c, d)]. \tag{3.48}$$

As by induction hypothesis we've already shown the theorem for $A(x) \vee B$, there exist terms p and q such that

$$\mathsf{PL}^{\omega}_{*} \vdash \forall a, c \; [A_{SH}(x, a, pxac) \vee B_{SH}(c, qxac)]. \tag{3.49}$$

Notice that p and q may depend on x. We now have to define terms r and s such that

$$\mathsf{PL}^{\omega}_{*} \vdash \forall x, a, c \; [A_{SH}(x, a, rxac) \vee B_{SH}(c, sxac)]. \tag{3.50}$$

Defining

$$r := \lambda x, a, c.pxac,$$
$$s := \lambda x, a, c.qxac,$$

that is, respectively, $rxac = pxac$ and $sxac = qxac$, concludes the proof of the $\forall$-introduction Rule, as (3.50) becomes $\forall x, a, c \; [A_{SH}(x, a, pxac) \vee B_{SH}(c, qxac)]$.

- *The Bounded Axiom of Choice* $\mathsf{bAC}^{\omega}_{*}$

Since the Soundness Theorem is to hold for every formula that can be proven from $\mathsf{PL}^{\omega}_{*}+ \mathsf{bAC}^{\omega}_{*}$, for the last part of the proof we need to consider the case where formula A is the Bounded Axiom of Choice. However, this case can be directly retrieved from Examples 2.4 and 3.2.

This concludes the proof of the Soundness Theorem. $\square$

The Soundness Theorem was proven without explicitly mentioning the types involved. If done otherwise, it would have turned the proof much heavier in terms of notation. The next remark discusses an observation about types that was used during the proof of the Cut Rule.

Remark 3.4. *(Types in the proof of the Soundness Theorem - Cut Rule)* Let A and B be formulas as in the proof of the Soundness Theorem with interpretations given by

$$A^{SH} \equiv \forall a^{\sigma} \exists b^{\tau} \; A_{SH}(a, b),$$
$$B^{SH} \equiv \forall c^{\gamma} \exists d^{\delta} \; B_{SH}(c, d),$$

where σ, τ, γ and δ are types. With these types, we then have

$$(\neg A)^{SH} \equiv \forall f^{\sigma \to \tau} \exists a'^{\sigma^*} \; [\exists a \in a' \; \neg A_{SH}(a, fa)].$$

In the proof of the Cut Rule, we mentioned that the term $f : \sigma \to \tau$ has the same type as $\lambda a.tac$. To see why $\lambda a.tac$ also has type $\sigma \to \tau$, notice that the term tac is used to witness the term $b : \tau$ in (3.39), and hence we must have $tac : \tau$. But then, since $a : \sigma$ we have $\lambda a.tac : \sigma \to \tau$, just like f. On another note, we also conclude that since $tac : \tau$ and $c : \gamma$, we have $t : \sigma \to \gamma \to \tau$. $\boxtimes$

Before moving on to further results in PL^{ω}_{*}, we finish off this subsection with a remark on *herbrandized interpretations*. An example of such an interpretation is already known to us: Shoenfield's interpretation is precisely a herbrandized interpretation.

Remark 3.5. (*Herbrandized interpretations*) Shoenfield's functional interpretation as given in Definition 2.1 is oftentimes referred to as a *herbrandized* functional interpretation. A functional interpretation is said to be *herbrandized* when the witnesses of the existential quantifiers of the interpretations do not need to be exact; we merely need to provide a finite and non-empty set of witnesses. This is usually expressed as saying that the witnesses are *accumulated* into finite sets.

This relates to a previous remark, namely Remark 2.3, where we mentioned that the existential variables of the interpretation of a formula are precisely of star type. Recalling that the intended meaning of the star type is to represent finite and non-empty sets, it means that in an interpretation such as $A^{SH} \equiv \forall a \exists b A_{SH}(a, b)$, the existential variable b is actually of the form $\{b_1, ..., b_n\}$, where $b_1, ..., b_n$ represent possible witnesses.

Another aspect that further establishes that Shoenfield's interpretation is a herbrandized interpretation is expressed in the Monotonicity Lemma. In the lemma, we introduce the notion of inclusion of terms $b \subseteq b'$, where b and b' are terms of the same star type, where $b \subseteq b'$ stands for $\forall x \ (x \in b \to x \in b')$. When thinking of these terms as finite and non-empty sets, the definition becomes even more clear. A set S is contained in a bigger set S' precisely when the elements of S are all included in S'.

The accumulation of the witnesses of existential quantifiers can also be found in other frameworks with different functional interpretations, oftentimes slightly adapted as needed. For example, some functional interpretations even accumulate the witnesses below certain majorants (refer to *Bounded functional interpretation* [22] by F. Ferreira and P. Oliva for such a discussion). ⊠

3.2 Further results about the Star Combinatory Calculus

We will now investigate further results in PL^{ω}_{*}. First, we will discuss under which conditions a formula is equivalent to its interpretation (Section 3.2.1). We will then analyze the role of the Bounded Axiom of Choice in PL^{ω}_{*} (Section 3.2.2) and, to conclude this section, we will present results that will ultimately allow us to slowly but surely state Herbrand's Theorem in the context of the present dissertation (Section 3.2.3).

3.2.1 The Characterization Theorem

At the beginning of Section 2.1, we discussed that we could extend the notion of prenexification to formulas in PL^{ω}_{*}, as long as we accepted equivalences E1 and E2 (refer to page 34). We then also mentioned that the prenexification of a formula is always equivalent to the original formula. A question that may now arise is whether or not the interpretation of a formula is also equivalent to the original formula. We already saw that this holds for at least some formulas. For example, base formulas are, by definition, always equivalent to their interpretation. Is it possible to affirm the same for other formulas? What is the relationship between a formula and its interpretation? The *Characterization Theorem* provides answers to these questions and states that the equivalence between a formula A and its interpretation A^{SH} is guaranteed for every single formula, but only under a certain condition.

Theorem 3.2. (Characterization Theorem) *Let A be a formula of $\mathscr{L}^{\omega}_{*}$. Then*

$$\mathsf{PL}^{\omega}_{*} + \mathsf{bAC}^{\omega}_{*} \vdash A \leftrightarrow A^{SH}.$$

In short, if we add the Bounded Axiom of Choice $\mathsf{bAC}^{\omega}_{*}$ as an extra principle to the theory PL^{ω}_{*}, then any formula is equivalent to its interpretation. To understand why this theorem is noteworthy, we need to refer to previous considerations. We have discussed that the interest of a functional interpretation lies, in particular, in the fact that it can provide information about formulas. For example, in the previous section it was seen that the Soundness Theorem provides information encoded in terms about the interpretation A^{SH}, but not about the formula A itself. This is where the Characterization Theorem comes in. Since it states the equivalence between a formula and its interpretation, the information retrieved from the Soundness Theorem for A^{SH} can now be transferred back to the original formula A.

In order to prove the Characterization Theorem, we will need two new principles that follow directly from the Bounded Axiom of Choice.

Lemma 3.2. *Let A_b be a base formula in $\mathscr{L}^\omega_*$. Then*

$$\mathsf{PL}^\omega_* + \mathsf{bAC}^\omega_* \ \vdash \ \forall x \in t\ \exists y\ A_b(x,y) \ \rightarrow \ \exists b\ \forall x \in t\ \exists y \in b\ A_b(x,y).$$

Proof. Assume $\forall x \in t\ \exists y\ A_b(x,y)$. By Axiom AxU, this can be rewritten as $\forall x\ (x \in t \rightarrow \exists y A_b(x,y))$ and further as

$$\forall x \exists y\ (x \in t \rightarrow A_b(x,y)) \tag{3.51}$$

by applying the usual prenexification rules. Since the matrix $x \in t \rightarrow A_b(x,y)$ is a base formula, we can apply bAC^ω_* to it as to obtain

$$\exists f\ \forall x\ \exists y \in fx\ (x \in t \rightarrow A_b(x,y)),$$

which can then be rewritten back as

$$\exists f\ \forall x \in t\ \exists y \in fx\ A_b(x,y). \tag{3.52}$$

Let f_0 be a witness for (3.52) such that $\forall x \in t\ \exists y \in f_0 x\ A_b(x,y)$ and let

$$b := \bigcup_{x \in t} f_0 x.$$

Since every y in $f_0 x$ is necessarily in the union b, we have shown $\exists b\ \forall x \in t\ \exists y \in b\ A_b(x,y)$, the conclusion of the lemma. □

Lemma 3.3. *Let A be a formula in $\mathscr{L}^\omega_*$ such that $A^{SH} \equiv \forall a \exists b\ A_{SH}(a,b)$. Then*

$$\mathsf{PL}^\omega_* + \mathsf{bAC}^\omega_* \ \vdash \ \forall a\ \exists b\ A_{SH}(a,b) \ \leftrightarrow \ \exists g\ \forall a\ A_{SH}(a,ga).$$

Proof. The backward direction is straightforward: if we have $\exists g \forall a\ A_{SH}(a,ga)$, then we can define b to be given by ga and hence $\forall a A_{SH}(a,ga)$.

For the forward direction, assume $\forall a \exists b\ A_{SH}(a,b)$. By using bAC^ω_*, we then have $\exists f \forall a \exists b \in fa\ A_{SH}(a,b)$. Let f_0 be under these conditions, i.e. $\forall a \exists b \in f_0 a\ A_{SH}(a,b)$, and let g be defined such that

$$ga := \bigcup_{b \in f_0 a} b.$$

We are going to see that with this definition we have $\forall a\ A_{SH}(a,ga)$, hence proving the lemma. Let a be given. Then there is $b \in f_0 a$ such that $A_{SH}(a,b)$. Since we have that $b \subseteq ga$, by the Monotonicity Lemma, we also have $A_{SH}(a,ga)$. We conclude that we have found some g such that given a we have $A_{SH}(a,ga)$. □

With these two lemmas at hand, we will now prove the Characterization Theorem. Lemma 3.2 will be used to prove the Characterization Theorem for formulas of the form $\forall x \in t\ A(x)$ while Lemma 3.3 will be used for formulas of the form $\neg A$.

Proof of the Characterization Theorem. The proof will be done by induction on the complexity of the formulas. We must therefore once again check the theorem for the five clauses of Definition 2.1.

1. *Base formulas:* Let A be a base formula in $\mathscr{L}^\omega_*$. Since $A \equiv A^{SH}$, we automatically have

$$\mathsf{PL}^\omega_* + \mathsf{bAC}^\omega_* \vdash A \leftrightarrow A^{SH}.$$

For the remaining cases, let A and B be formulas in $\mathscr{L}^\omega_*$ such that $A^{SH} \equiv \forall a \exists b\ A_{SH}(a, b)$ and $B^{SH} \equiv \forall c \exists d\ B_{SH}(c, d)$.

2. *Disjunction:* We have to show that $\mathsf{PL}^\omega_* + \mathsf{bAC}^\omega_* \vdash A \vee B \leftrightarrow (A \vee B)^{SH}$. By induction hypothesis, we have $\mathsf{PL}^\omega_* + \mathsf{bAC}^\omega_* \vdash A \leftrightarrow A^{SH}$ and $\mathsf{PL}^\omega_* + \mathsf{bAC}^\omega_* \vdash B \leftrightarrow B^{SH}$. Now, since $(A \vee B)^{SH} \equiv \forall a, c \exists b, d\ [A_{SH}(a, b) \vee B_{SH}(c, d)]$, then we have

$$\begin{aligned} A \vee B &\leftrightarrow A^{SH} \vee B^{SH} \\ &\leftrightarrow \forall a \exists b\ A_{SH}(a, b) \vee \forall c \exists d\ B_{SH}(c, d) \\ &\leftrightarrow \forall a, c \exists b, d\ [A_{SH}(a, b) \vee B_{SH}(c, d)] \\ &\equiv (A \vee B)^{SH}, \end{aligned}$$

where for the third equivalence we have used the usual prenexification rules.

3. *Unbounded universal quantification:* To show $\mathsf{PL}^\omega_* + \mathsf{bAC}^\omega_* \vdash \forall x A(x) \leftrightarrow (\forall x A(x))^{SH}$, recall that $(\forall x A(x))^{SH} \equiv \forall x, a \exists b\ A_{SH}(x, a, b)$. By induction hypothesis we know that $A(x) \leftrightarrow A(x)^{SH}$ is provable from $\mathsf{PL}^\omega_* + \mathsf{bAC}^\omega_*$. Hence,

$$\begin{aligned} \forall x\ A(x) &\leftrightarrow \forall x\ A(x)^{SH} \\ &\leftrightarrow \forall x\ \forall a \exists b A_{SH}(x, a, b) \\ &\equiv (\forall x A(x))^{SH}. \end{aligned}$$

4. *Bounded universal quantification:* To show that $\forall x \in t\ A(x) \leftrightarrow (\forall x \in t\ A(x))^{SH}$ is provable in $\mathsf{PL}^\omega_* + \mathsf{bAC}^\omega_*$, recall that $(\forall x \in t\ A(x))^{SH} \equiv \forall a \exists b\ [\forall x \in t\ A_{SH}(x, a, b)]$. Since by induction hypothesis we have $\mathsf{PL}^\omega_* + \mathsf{bAC}^\omega_* \vdash A(x) \leftrightarrow A(x)^{SH}$, we then have

$$\begin{aligned} \forall x \in t\ A(x) &\leftrightarrow \forall x \in t\ A(x)^{SH} \\ &\leftrightarrow \forall x \in t\ \forall a\ \exists b\ A_{SH}(x, a, b) && (3.53) \\ &\leftrightarrow \forall a\ \forall x \in t\ \exists b\ A_{SH}(x, a, b) && (3.54) \\ &\leftrightarrow \forall a\ \exists b\ \forall x \in t\ A_{SH}(x, a, b) && (3.55) \\ &\equiv (\forall x \in t\ A(x))^{SH}. \end{aligned}$$

Equivalence between (3.53) and (3.54) is simply the usual prenexification rules. We now discuss the equivalence between (3.54) and (3.55). That (3.55) implies (3.54) is fairly simple: if there exists a b for all x in t, then we can take this same b for every single x in t and obtain (3.54).

To show that (3.54) implies (3.55), assume $\forall x \in t\ \exists b\ A_{SH}(x, a, b)$, where we treat a as a parameter. By Lemma 3.2, we then have $\exists b'\ \forall x \in t\ \exists b \in b'\ A_{SH}(x, a, b)$. Now consider the union

$$b'' := \bigcup_{b \in b'} b.$$

We will see that this b'' is the one whose existence we are trying to prove to show (3.55). To prove this, let $x_0 \in t$. Then there exists $b_0 \in b'$ such that $A_{SH}(x_0, a, b_0)$. Moreover, since $b_0 \in b'$ we have $b_0 \subseteq b''$. But then, by the Monotonicity Lemma, since $A_{SH}(x_0, a, b_0)$ is monotonous in b_0, we also have $A_{SH}(x_0, a, b'')$.

In conclusion, since x_0 in t was arbitrary, we have shown the existence of some b (here given by b'') such that $\forall x \in t\ A_{SH}(x, a, b)$.

5. *Negation:* In order to show $\mathsf{PL}_*^\omega + \mathsf{bAC}_*^\omega \vdash \neg A \leftrightarrow (\neg A)^{SH}$, recall that

$$(\neg A)^{SH} \equiv \forall f \exists a' \ [\exists a \in a' \ \neg A_{SH}(a, fa)].$$

By induction hypothesis, we have $\mathsf{PL}_*^\omega + \mathsf{bAC}_*^\omega \vdash A \leftrightarrow A^{SH}$ and, therefore,

$$\begin{aligned}
\neg A &\leftrightarrow \neg(A^{SH}) \\
&\leftrightarrow \neg(\forall a \exists b\ A_{SH}(a, b)) && (3.56) \\
&\leftrightarrow \neg(\exists f \forall a\ A_{SH}(a, fa)) && (3.57) \\
&\leftrightarrow \forall f \exists a\ \neg A_{SH}(a, fa) && (3.58) \\
&\leftrightarrow \forall f \exists a' \ [\exists a \in a' \ \neg A_{SH}(a, fa)] && (3.59) \\
&\leftrightarrow (\neg A)^{SH}.
\end{aligned}$$

The equivalence between (3.56) and (3.57) is a direct application of Lemma 3.3. We now discuss the equivalence between (3.58) and (3.59). The backward implication is straightforward: given some f, from (3.59) there exists an a (that belongs to some a') such that $\neg A_{SH}(a, fa)$. But the information that $a \in a'$ does not change the existence of said a, hence yielding (3.58). For the forward implication, by AxS_1 we can define a' to be given by $\mathfrak{s}a$ and hence obtain (3.59). □

Referring to the proof of the negation case of the Characterization Theorem, it is now possible to make an interesting observation on how Shoenfield's functional interpretation treats the interpretations of negations.

Remark 3.6. Definition 2.1 stated that if A is a formula in $\mathscr{L}_*^\omega$ with interpretation A^{SH} given by $\forall a \exists b\ A_{SH}(a, b)$, then the interpretation $(\neg A)^{SH}$ is given by $\forall f \exists a' \ [\exists a \in a' \ \neg A_{SH}(a, fa)]$. A question that may arise is why we did not choose to define the interpretation of the negation as simply being $(\neg A)^{SH} \equiv \forall f \exists a\ \neg A_{SH}(a, fa)$. As shown in the above proof, both versions are equivalent (refer to the equivalence between (3.58) and (3.59)). But why choose the more complex version? The reason is because only the version from Definition 2.1 makes the existential variable monotonous and, as has been seen, the Monotonicity Lemma is fundamental in the discussion of the present dissertation. Only the version from Definition 2.1 guarantees that we can always take *bigger* terms and everything is still valid. This would no longer be the case if we were to choose to define $(\neg A)^{SH} \equiv \forall f \exists a\ \neg A_{SH}(a, fa)$. ⊠

3.2.2 Characteristic principles of the Star Combinatory Calculus

For the Soundness and the Characterization Theorems to work, it was necessary to strengthen the theory PL_*^ω with the extra principle bAC_*^ω. Moreover, it was seen that bAC_*^ω *disappears* in the conclusion of the Soundness Theorem. It is common terminology in the study of functional interpretations to call such principles, that vanish in the conclusion of the Soundness Theorem, *characteristic principles* of the functional interpretation. In this sense, bAC_*^ω is said to be a characteristic principle of Shoenfield's functional interpretation. More interestingly though, bAC_*^ω is *the* characteristic principle of Shoenfield's functional interpretation.

Remark 3.7. We will now see that there are no other characteristic principles for Shoenfield's functional interpretation when interpreting PL_*^ω into itself. This will be done by showing that any extra principle we add to the assumptions of the Soundness Theorem necessarily follows from PL_*^ω and bAC_*^ω. The Characterization Theorem will help us achieve this.

Assume there is a principle P missing from the Soundness Theorem, that is, assume that, instead of the original Soundness Theorem hypothesis $\mathsf{PL}_*^\omega + \mathsf{bAC}_*^\omega \vdash A$, we have

$$\mathsf{PL}_*^\omega + \mathsf{bAC}_*^\omega + \mathsf{P} \vdash A, \tag{3.60}$$

where A is a formula in $\mathscr{L}_*^\omega$. By (3.60), we automatically have

$$\mathsf{PL}_*^\omega + \mathsf{bAC}_*^\omega + \mathsf{P} \vdash \mathsf{P}$$

since P is a consequence of itself. Let $\mathsf{P}^{SH} \equiv \forall a \exists b\ \mathsf{P}_{SH}(a,b)$. Then, by the Soundness Theorem with this extra principle, there exists a term t such that $\mathsf{PL}_*^\omega \vdash \forall a\ \mathsf{P}_{SH}(a, ta)$. Since $\forall a\ \mathsf{P}_{SH}(a, ta)$ is simply P^{SH} with an explicit existential witness (given by ta), we also have

$$\mathsf{PL}_*^\omega \vdash \mathsf{P}^{SH}. \tag{3.61}$$

Now, by the Characterization Theorem, we have

$$\mathsf{PL}_*^\omega + \mathsf{bAC}_*^\omega \vdash \mathsf{P} \leftrightarrow \mathsf{P}^{SH}. \tag{3.62}$$

Finally, by combining (3.61) and (3.62) we obtain $\mathsf{PL}_*^\omega + \mathsf{bAC}_*^\omega \vdash \mathsf{P}$. This shows that any other principle we add to the Soundness Theorem (such as P above) can be obtained by the principles already involved in the theorem. ⊠

Shoenfield's functional interpretation when interpreting PL_*^ω into PL_*^ω has therefore only one characteristic principle, namely the Bounded Axiom of Choice. Naturally, other functional interpretations will have other characteristic principles. For example, Gödel's Dialectica interpretation has three characteristic principles that vanish in the Soundness Theorem under the Dialectica interpretation. For a concise discussion of the characteristic principles of Gödel's Dialectica interpretation, refer to [20], Section II.

3.2.3 Relation between Herbrand's Theorem and the Soundness Theorem

In basic logic courses it is routine to explore the well-known Herbrand's Theorem, a result of fundamental importance that extracts terms from existential statements that are provable in classical first-order logic. In first-order logic, Herbrand's Theorem can be stated as follows.

Theorem 3.3. (Herbrand's Theorem) *Let $A_{qf}(x)$ be a quantifier-free formula of a first-order language $\mathscr{L}$, with x as the only free variable. If $\mathsf{PL} \vdash \exists x\ A_{qf}(x)$, then there exist closed first-order terms $t_1, ..., t_n$ in $\mathscr{L}$ such that*

$$\mathsf{PL} \vdash A_{qf}(t_1) \vee ... \vee A_{qf}(t_n).$$

The importance of this theorem for Mathematical Logic is well established. As stated by Swiss mathematician Paul Bernays [8]

"Herbrand's Theorem can be seen as the central theorem of predicate logic."

Due to its importance, it has been proved in various different ways since its publication [27] in 1930. In light of the discussions from the previous sections, the goal now will be to provide such a proof and to consider a more general form of Herbrand's Theorem. Contrary to the above formulation of Herbrand's Theorem stated in PL, this more general form will be stated in PL_*^ω. We begin by discussing a preliminary result called the *Main Theorem*. In what follows, let $A_b(x)$ be a base formula in $\mathscr{L}_*^\omega$, with x as its only free variable.

Theorem 3.4. (Main Theorem) *If $\mathsf{PL}_*^\omega + \mathsf{bAC}_*^\omega \vdash \exists x^\sigma\ A_b(x)$, then there exists a closed term t of type σ^* in $\mathscr{L}_*^\omega$ such that*

$$\mathsf{PL}_*^\omega \vdash \exists x \in t^{\sigma^*}\ A_b(x).$$

To understand what this theorem entails, first notice that we can assume t to also be normal and not just closed. In fact, if t is not normal it can simply be normalized by using the

Strong Normalization Theorem. But then, since t is both normal and closed, by Remark 1.27, the term t, being of star type σ^*, represents a finite non-empty set of terms of type σ. In other words, it represents a set of the form $\{t_1, ..., t_n\}$, where $t_1, ..., t_n : \sigma$ are terms. The Main Theorem therefore states that if we can prove the existence of a witness x for $A_b(x)$ in the strengthened theory $\mathsf{PL}^\omega_* + \mathsf{bAC}^\omega_*$, then it is possible to choose such a witness from a finite and non-empty set of possible witnesses in the original theory PL^ω_*, without needing bAC^ω_*.

Taking this information into account, if we choose to rewrite the Main Theorem we get a corollary with the same conditions as the Main Theorem, but with a conclusion that strongly resembles Herbrand's Theorem. In fact, this corollary is a generalization of Herbrand's Theorem in PL^ω_*.

Corollary 3.1. (General form of Herbrand's Theorem) *If* $\mathsf{PL}^\omega_* + \mathsf{bAC}^\omega_* \vdash \exists x^\sigma\, A_b(x)$, *then there exist closed terms* $t_1, ..., t_n$ *of type* σ *in* $\mathscr{L}^\omega_*$ *such that*

$$\mathsf{PL}^\omega_* \vdash A_b(t_1) \vee ... \vee A_b(t_n).$$

The resemblance with Herbrand's Theorem becomes even more apparent if we let $t_1, ..., t_n$ and x be of ground type G (instead of a general type σ). Since, by the above discussion, we can assume the terms $t_1, ..., t_n$ to be closed normal terms of ground type, then, by the Ground Normal Form Proposition (Proposition 1.3), they correspond to closed terms of the original language $\mathscr{L}$. Hence, since the terms $t_1, ..., t_n$ correspond to terms of *first-order logic*, we can derive a particular case of Corollary 3.1 that takes us back to first-order logic.

Corollary 3.2. *If* $\mathsf{PL}^\omega_* + \mathsf{bAC}^\omega_* \vdash \exists x^G\, A_b(x)$, *then there exist closed first-order terms* $t_1, ..., t_n$ *of type* G in $\mathscr{L}$ *such that*

$$\mathsf{PL}^\omega_* \vdash A_b(t_1) \vee ... \vee A_b(t_n).$$

One could now ask how these two last corollaries are generalizations of Herbrand's Theorem, since Herbrand's Theorem neither mentions the Bounded Axiom of Choice nor base formulas in PL. Consider the following remark.

Remark 3.8. *(Generalization of Herbrand's Theorem)* Notice that our first formulation of Herbrand's Theorem (Theorem 3.3) is stated in PL instead of PL^ω_*. Moreover, it is stated for quantifier-free formulas instead of base formulas and it does not include the Bounded Axiom of Choice in its premise, contrary to Corollaries 3.1 and 3.2. These corollaries are nevertheless generalizations of Herbrand's Theorem as we will now discuss.

First, as in PL there are no bounded quantifiers, the analogues to quantifier-free formulas in PL^ω_* are the base formulas. Second, as the principle bAC^ω_* is a principle which involves finite types of higher order, it cannot be visible in a first-order logic setting such as PL. However, when Herbrand's Theorem is stated in PL^ω_*, the Bounded Axiom of Choice unveils itself in the premises. Hence, Corollaries 3.1 and 3.2 not only resemble Herbrand's Theorem, but they are, in fact, generalizations of it. ⊠

Next, we will prove the three results we just discussed, that is, the Main Theorem (Theorem 3.2) and its corollaries (Corollaries 3.1 and 3.2). These proofs will require our understanding of the previous sections, namely that of Section 1.2.5 where we discussed the conversions and normalization properties of the Star Combinatory Calculus. We begin by proving the Main Theorem, which, as will now be seen, is a direct consequence of the Soundness Theorem from Section 3.1.

Proof of the Main Theorem. Assume $\mathsf{PL}^\omega_* + \mathsf{bAC}^\omega_* \vdash \exists x A_b(x)$, where $A_b(x)$ is a base formula in $\mathscr{L}^\omega_*$. Since the Main Theorem and the Soundness Theorem have the same assumptions and since $\exists x A_b(x)$ is a formula in $\mathscr{L}^\omega_*$, we can apply the Soundness Theorem to it. To do so, we first need to compute $(\exists x A_b(x))^{SH}$. Referring to Remark 2.6, we have

$$(\exists x A_b(x))^{SH} \equiv \exists x' \, [\exists x \in x' A_b(x)]. \tag{3.63}$$

By the Soundness Theorem, there exists a closed term t that witnesses the existential variable x' in (3.63). Hence, since there are no universal quantifiers in (3.63), the formula $\exists x' \, [\exists x \in x' A_b(x)]$ can be rewritten simply as $\exists x \in t \; A_b(x)$. This corresponds to the conclusion of the Main Theorem, finishing off the proof. Ultimately, the closed term that needed to be constructed to prove the Main Theorem is precisely the closed term that comes from the Soundness Theorem. □

We now prove the corollaries of the Main Theorem. To do so, we will need to use the results from Section 1.2.5 together with the following lemma.

Lemma 3.4. *If t^{σ^*} is a closed term in $\mathscr{L}^\omega_*$, then there exist closed terms $t_1, ..., t_n$ in $\mathscr{L}^\omega_*$ such that*

$$\mathsf{PL}^\omega_* \vdash \forall x \; (x \in t \leftrightarrow x = t_1 \vee ... \vee x = t_n).$$

Proof of Lemma 3.4. We will prove the lemma by induction on the complexity of the terms. Let t^{σ^*} be a closed term in $\mathscr{L}^\omega_*$. We can assume t to be normal, since, by the Strong Normalization Theorem, if the term t is not normal it can be reduced to one. Since t is normal and closed, by the Ground Normal Form Proposition, the term t is then set-like and, by definition, it is either of the form $\mathfrak{s}r$ or of the form $\cup pq$, where r, p, q are closed normal terms. The first case is straightforward since by Axiom AxS_1 we have

$$\forall x \; (x \in \mathfrak{s}r \leftrightarrow x = r),$$

which corresponds to the conclusion of the lemma when t is of the form $\mathfrak{s}r$. In the second case, if t is of the form $\cup pq$, then by Axiom AxS_2 we have

$$\forall x \; (x \in \cup pq \leftrightarrow x \in p \vee x \in q). \tag{3.64}$$

By induction hypothesis on the complexity of the term, since the terms p and q are of lesser complexity, the lemma holds for p and q and, hence, $x \in p$ is equivalent to $x = p_1 \vee ... \vee x = p_k$ and $x \in q$ is equivalent to $x = q_1 \vee ... \vee x = q_l$, for closed terms $p_1, ..., p_k, q_1, ..., q_l$. Finally, from (3.64) we retrieve

$$\forall x \; (x \in \cup pq \leftrightarrow x = p_1 \vee ... \vee x = p_k \vee x = q_1 \vee ... \vee x = q_l).$$

This concludes the proof, as in both cases we have found a finite number of terms that satisfy the conclusion of the lemma. □

Using this lemma, we now prove the general form of Herbrand's Theorem (Corollary 3.1).

Proof of Corollary 3.1. Assume $\mathsf{PL}^\omega_* + \mathsf{bAC}^\omega_* \vdash \exists x^\sigma A_b(x)$. By the Main Theorem, there exists a closed term t^{σ^*} such that $\mathsf{PL}^\omega_* \vdash \exists x \in t \; A_b(x)$. Now, by applying Lemma 3.4 to t, there exist closed terms $t_1, ..., t_n$ such that

$$\mathsf{PL}^\omega_* \vdash \forall x \; (x \in t \leftrightarrow x = t_1 \vee ... \vee x = t_n). \tag{3.65}$$

But then, from $\mathsf{PL}^\omega_* \vdash \exists x \in t \; A_b(x)$ and (3.65) we can conclude that $t_1, ..., t_n$ are possible witnesses for A_b and hence $\mathsf{PL}^\omega_* \vdash A_b(t_1) \vee ... \vee A_b(t_n)$. □

Next we prove Corollary 3.2.

Proof of Corollary 3.2. The proof of Corollary 3.2 follows directly from Corollary 3.1 and the considerations right after the statements of Theorem 3.4 and Corollary 3.1. □

Now that the Main Theorem and its corollaries have been proven, there is still a remark that should be made about the relation between Herbrand's Theorem in first-order logic (Theorem 3.3) and the generalization of Herbrand's Theorem in PL^ω_* (Corollary 3.1).

Remark 3.9. Depending on the perspective we choose to adopt, the general version of Herbrand's Theorem (Corollary 3.1) can either be seen as a weakening or a strengthening of the original version of Herbrand's Theorem (Theorem 3.3).

Straightforwardly, Corollary 3.1 can be seen as a strengthening as it is, in fact, more general. While the original theorem is stated in PL, Corollary 3.1 is stated in its higher order analogue PL^ω_* which makes it possible to apply it to a broader range of scenarios. After all, the typed version of PL is included in PL^ω_*.

However, Corollary 3.1 can also be seen as a weakening of Herbrand's Theorem due to where the verification of $A(t_1) \vee ... \vee A(t_n)$ takes place. Since the original version of Herbrand's Theorem is stated in PL, the verification of $A(t_1) \vee ... \vee A(t_n)$ is also done in PL. When going to the higher order logic PL^ω_*, as is the case with Corollary 3.1, the verification of $A(t_1) \vee ... \vee A(t_n)$ is no longer done in the original first-order theory PL; we are forced to do the verification in PL^ω_*. In this sense, it is not possible to affirm that Corollary 3.1 is a *true* generalization of Herbrand's Theorem. In fact, Herbrand's Theorem is then not a direct consequence of Corollary 3.1.

Nevertheless, it is justified to consider the general version of Herbrand's Theorem as it unravels information that was hidden in the original statement, such as the Bounded Axiom of Choice bAC^ω_* that appears in the general form of the theorem. The understanding which is gained by going to higher order logic then makes up for what might be lost by requiring that the verification be made in PL^ω_* instead of its first order analogue PL. Going to higher order logic compensates this small weakening in the verification of the disjunction. ⊠

The generalizations of Herbrand's Theorem are of great interest because they made it possible to analyze Herbrand's Theorem in a broader context through the use of functional interpretations. However, we could now ask if it is possible to retrieve the original formulation of Herbrand's Theorem (as stated in Theorem 3.3) as a a direct consequence of our previous corollaries. The next corollary is the final piece of the puzzle to retrieve Herbrand's Theorem as stated in Theorem 3.3.

Corollary 3.3. Let A be a quantifier-free formula of $\mathscr{L}$ such that $\mathsf{PL}^\omega_* \vdash A_{qf}(t_1) \vee ... \vee A_{qf}(t_n)$. Then,

$$\mathsf{PL} \vdash A_{qf}(t_1) \vee ... \vee A_{qf}(t_n).$$

To show this corollary, we will provide a semantical argument that involves Gödel's Completeness Theorem. In short, it states that if a formula is true in every structure of PL, then it is provable in PL. This will allow us to prove the syntactical derivability from discussions about semantical truths in the first-order context of $\mathscr{L}$ and PL. To do so we make some preliminary considerations on how to associate a structure to $\mathscr{L}^\omega_*$.

Right at the beginning of the dissertation we discussed the intended meaning of each of the finite types (after Definition 1.1) and of the terms of $\mathscr{L}^\omega_*$ (Remark 1.6). For example, a term $t^{\sigma\to\tau}$ represents a function which takes arguments of type σ and returns elements of type τ. Although we did not explicitly mention it, the intended meaning is associated to a certain canonical structure.

Remark 3.10. *(Set theoretic structure)* Any structure of $\mathscr{L}$ can be canonically extended to a structure of $\mathscr{L}_*^\omega$. ⊠

We now prove Corollary 3.3. We will prove it by contradiction.

Proof. Assume by contradiction that the first-order formula $A_{qf}(t_1) \vee ... \vee A_{qf}(t_n)$ is not provable in PL. By Gödel's Completeness Theorem, it must be false in some structure of $\mathscr{L}$. Then it must also be false in the corresponding set-theoretical structure of $\mathscr{L}_*^\omega$. Since $A_{qf}(t_1) \vee ... \vee A_{qf}(t_n)$ is provable in PL_*^ω, then it has to be true in any structure that satisfies PL_*^ω. In particular, it must be true in the above mentioned set-theoretical structure. This is a contradiction. □

Now the proof of Herbrand's Theorem (Theorem 3.3) follows automatically from these results and we obtain a new proof for this well-known theorem.

Proof of Herbrand's Theorem. Direct consequence of Corollaries 3.2 and 3.3 for quantifier-free formulas of $\mathscr{L}$. □

Our discussion around Herbrand's Theorem together with the new proof for this theorem showed that, in the end, Herbrand's Theorem follows directly from our functional interpretation from Section 2. This leads us to the conclusion that functional interpretations can be seen as *generalizations* of Herbrand's Theorem. But how did it all start?

A *functional* narrative on Herbrand's Theorem

Herbrand's Theorem has a long history.
It was first stated and proven under the name of *Théorème Fondamental* by 22-year-old Jacques Herbrand himself in his doctoral thesis *Recherches sur la Théorie de la Démonstration* [27] from 1930. Even though his thesis received much praise, the proof of said theorem was wrong. A corrected version was given by American philosopher Burton Dreben et al., more than 30 years later, in a collection of papers during the 1960's [17, 18, 19]. The proof was also corrected by Gödel himself, although he never published it.

Although Herbrand could at that time probably not have fathomed it, his *Théorème Fondamental* would be of crucial significance. Apart from the now well-known impact it had during the 20th century in mathematical logic, Austrian mathematician Georg Kreisel made the first step to connect Herbrand's Theorem with functional interpretations. Kreisel kept a professional relationship with Gödel which made them have numerous conversations about Mathematical Logic and, in particular, about Gödel's Dialectica interpretation. These interactions allowed Kreisel to have a deeper understanding of Gödel's interpretation which then allowed Kreisel to develop his own functional interpretation called *Kreisel's modified realizability* [30]. Being hence well acquainted with the world of functional interpretations, Kreisel wondered whether it was possible to prove Herbrand's Theorem using functional interpretations. It was then that Philipp Gerhardy, together with his PhD advisor Ulrich Kohlenbach, were able to give a solution to Kreisel's question in their paper *Extracting Herbrand disjunctions by functional interpretation* [24] from 2005. In this paper they present an interpretation of classical logic which was based on the original Dialetica interpretation. However, in order for them to prove Herbrand's Theorem, Gerhardy and Kohlenbach had to introduce so called *conditional functionals*, just for this specific task. It is then that, as a response, Portuguese mathematicians Gilda Ferreira and Fernando Ferreira proved Herbrand's Theorem in *A herbrandized functional interpretation of classical first-order logic* [21] also through the use of functional interpretations, but without needing these new functionals. To do so, they introduced a new finite type, the star-type, that made it possible to deal with the concept of finiteness in classical logic. This is the point at which the present dissertation comes in.

One of the main focuses of this dissertation was to prove Herbrand's Theorem in a new and more general setting. While Ferreira and Ferreira have proven the theorem using a functional interpretation based on a modified version of Shoenfield's functional interpretation to interpret first-order logic into the their more restricted version of PL^ω_* (namely because they do not use bounded quantifiers), here we go further and we interpret PL^ω_* into itself. In this document the theorem has also been shown when considering an interpretation of PL^ω_* into itself. It is only in this more general setting that we are able to see the characteristic principle bAC^ω_* appear in the premise of Herbrand's Theorem. In [21], the interpreting theory is solely based on first-order logic and it is therefore not possible to even state Herbrand's Theorem with the characateristic principle. It is only when going further, when interpreting PL^ω_* into itself, that it becomes possible to see the Bounded Axiom of Choice.

All of the aforementioned accomplishments are due to one young mathematician, Jacques Herbrand, who completed his PhD in Mathematical Logic in France, an area that, at the time, was not looked upon with great interest by his compatriots. Nevertheless, Herbrand was able to prove a logical milestone right at the beginning of his adulthood. Unfortunately, on the 27th of July 1931, Herbrand went on a climbing trip with some friends, a climbing trip from which he would never come back, as it is where he fell to his death at the mere age of 23. This tragic and untimely passing away, places Herbrand among the ranks of other brilliant yet short-lived mathematicians, such as Évariste Galois,(12) Niels Abel,(13) and Srinivasa Ramanujan,(14) all of whom were cut down in the prime of their potential. $\boxtimes$

We have concluded the first part of the present dissertation. Sections 1 to 3 focused on discussing the Star Combinatory Calculus and defining the herbrandized Shoenfield interpretation from a mathematical point of view. We now dive into the subject from a computer science perspective by using the proof assistant Lean in Section 4.

(12) Évariste Galois, outstanding French algebrist, deceased at the age of 20 during a duel.
(13) Niels Abel, remarkable Norwegian analyst, deceased at the age of 26 due to tuberculosis.
(14) Srinivasa Ramanujan, self-taught Indian mathematician, deceased at the age of 32 due to the aftermath of tuberculosis.

4 Formalization in Lean

Precision is key.

We will now enter the second part of the dissertation with the goal of formalizing certain topics of the previous sections, that is, we will construct algorithms which will make it possible for a computer to verify our proofs. This formalization will be done using a proof assistant called Lean introduced in Section 4.2. As it would exceed the scope of this dissertation, the goal of the upcoming section was not to formalize every aspect of Sections 1 to 3, but to formalize the Combinatorial Completeness Theorem (Theorem 1.1), the herbrandized version of Shoenfield's functional interpretation (Definition 2.1) and certain corresponding results. Naturally, this also meant formalizing the finite types (Section 1.1), the syntax of $\mathscr{L}^{\omega}_{*}$ (Section 1.2.1), the axiomatic of PL^{ω}_{*} (Section 1.2.2) and many other aspects that had to be added in terms of rigour, as we will soon discuss.

Although much of mathematics has already been formalized in Lean and other proof assistants, most of the content of the present dissertation has not yet been formalized in this or any other proof assistant. For example, first-order logic, finite types (without the star type) and the Dialectica interpretation [12] have all already been formalized in Lean. However, the finite types with the star type, the Combinatorial Completeness Theorem, the herbrandized version of Shoenfield's functional interpretation together with its Soundness Theorem have not yet been formalized in any proof assistant. We begin this second part of the dissertation by giving a general overview of proof assistants.

4.1 What are proof assistants?

Right from the beginning of the computer era, mathematicians have increasingly relied on computers to develop and further their research. This even led them to use computers to prove mathematical results. In very simple terms, any proof that involves a computer can be viewed as a *computer-assisted proof*.

A *proof assistant* is a tool to formalize mathematics through the use of computer-assisted proofs. In other words, it is a software that helps write down mathematical definitions, results and proofs. The main goal of a proof assistant is to construct such proofs, but also to verify the correctness of each step of the proof. This process then guarantees the validity of the computer-assisted proof. Just as the name points out, it is important to note that proof assistants merely *assist* users in their proofs. Currently, they are not user independent and require a considerable amount of human input.(15)

For further context, proof assistants are sometimes also called *interactive theorem provers* in contrast to *automated theorem provers*. While proof assistants are interactive and include a fair share of human input, automated theorem provers are tools to fully automatize the verification process and this with minimal user interaction. Even though automated theorem provers benefit from this kind of automation, they are less expressive than proof assistants and their automation only works in restricted frameworks. Although we won't discuss automated theorem provers here, it is interesting to see how the process of proving mathematical results is becoming more and more diverse.

Even though proof assistants seem to be a fairly recent invention to formalize mathematics on a computer, the first proof assistants are only roughly 30 years older than Turing machines. The very first attempt at developing a proof assistant was made in 1967 by Dutch mathematician

(15) Although this might change in the near future with the development of artificial intelligence.

Nicolaas de Bruijn and was called *Automath* [11]. Although Automath did not gain a lot of momentum in the mathematical community, it will forever be the predecessor of the more sophisticated proof assistants currently used and developed. Nowadays, the most famous proof assistants include *Coq* [33], *Isabelle* [32] and *Lean* [3], the proof assistant we will use (however, this really is not an extensive list). Coq's first version was developed in 1989, Isabelle's first version stems from 1999 and Lean's first version dates from 2013.

Advantages and disadvantages of proof assistants. As can be seen, proof assistants are not something of the past and have been developed on a consistent basis since the creation of Automath. However, there is still a large portion of the mathematical community that has never worked with a proof assistant. We now mention some advantages and disadvantages to shed some light onto this situation.

1. The use of proof assistants requires a very rigorous handling of the mathematical definitions and results, much more rigorous than the already rigorous mathematical writing. As a consequence, formalizations of mathematical results using proof assistants can guarantee a higher level of rigor and precision. However, this rigor comes at the cost of more effort in writing proofs.

2. Traditional proof verification has always been done by mathematicians through the process of peer-reviewing. Even though this practice has proven to be rather successful, it is impossible to exclude human error. By using proof assistants, checking the correctness and validity of a proof becomes more reliable and less prone to human error. However, it is important to remember that proof assistants are not yet perfect and may occasionally produce bugs. One should therefore always be vigilant and not blindly trust their output.

Even though the above two points mention disadvantages related to the use of proof assistants, the advantages are strong enough to explain why nowadays certain mathematicians predict that in the future only papers including a formal verification of the results will be accepted for publishing. Just as mathematician Andrej Bauer noted in *The Dialectica intepretation in Coq* [7],

> "I would like to think we are not too far from a future in which the standard in mathematical research will be to always publish papers together with computer-verified proofs."

Finally, there is one last aspect to mention that might be of great interest to any mathematician who regularly uses the words *obvious* or *trivial*.

3. Proof assistants such as Lean benefit from a certain degree of automation in their proofs. This means that they can take care of basic and routine aspects of proofs autonomously. On the one hand, this leaves mathematicians with more time and energy to focus on the more interesting and intriguing parts of the proof. On the other hand, this is not valid for every proof assistant. In less automated proof assistants, writing certain proofs is still very time consuming to.

Hence, although proof assistants can be an added value in mathematics, there are still some significant drawbacks that can leave mathematicians suspicious and hesitant about their use.

Having discussed some pros and cons of proof assistants, we now mention some real life examples of their use.

Real life examples. We begin by mentioning the well-known *Four Color Theorem* as a first example. The proof of this theorem had, so to say, a "rocky start". Its very first proof was a computer-assisted proof from 1976 which caused a lot of skepticism. The proof was an enormous case study that involved computer programs and human checking. Although it was a computer-assisted proof, it could not truly be verified by a computer. And, since verifying the proof by hand was extremely tedious, the proof was not widely accepted. Had the proof been done in a proof assistant, this would have been less of a problem. The proof assistant would have verified each step of the proof and there would have been less skepticism. Fortunately, in 2005 Benjamin Werner and Georges Gonthier were able to formalize the result in the proof assistant Coq. In conclusion, a computer-assisted proof that was initially frowned upon was replaced by another computer-assisted proof which is now widely accepted.

As another example, there are nowadays many renowned mathematicians using proof assistants. Field's medalists Peter Scholze and Terence Tao are two mathematicians currently involved in using the proof assistant Lean to formalize results. Peter Scholze has been responsible for the *Liquid Tensor Experiment*, an ongoing project to formalize one of his results that has joined together mathematicians and computer scientists from all over the world. Terence Tao is currently formalizing one of his most recent papers in Lean and, interestingly enough, was able to find a small, yet non-trivial mistake in his paper with the help of Lean.

Finally, proof assistants such as Lean have a significant amount of applications beyond pure mathematics. Because proof assistants verify the correctness and validity of algorithms, they can have applications in a wide range of areas. In computer science, they can help prove properties of algorithms and even of whole systems. In cryptography, they can verify protocols, hence enhancing their security. In biology, they can help verify properties related to biological models. In short, any area that relies on verifiable properties and algorithms can be positively influenced by the development of proof assistants.

Having established what proof assistants are and how they are useful to mathematics, we will discuss the proof assistant Lean in which parts of the present dissertation were formalized.

4.2 A proof assistant called Lean

The proof assistant Lean began as a Microsoft Research project [5] launched by Leonardo de Moura in 2013. The current version of Lean is called Lean4 (being its fourth version) and was released in 2021. Lean is an open source project in the sense that anyone can use its code and extend it. Lean has a constantly growing community and, as mentioned above, is used by important figures in the mathematical community. It is worth noting that the Lean community is a very helpful and interactive community. Through various platforms such as Zulip and Discord, Lean users can interact with one another, ask questions and clarify any doubts (with a bit of luck, users might even receive answers by Terence Tao, as this has already happened more than once). Moreover, there are currently a lot of Lean related events such as the conferences *Lean for the Curious Mathematician* and *Lean Together* that are occurring annually.(16) These conferences are usually recorded and the recordings are made available on Youtube. Because of the traction Lean has gained in the more recent years, it is becoming one of the proof assistants with the most potential.

But there is another point that explains the sheer potential of Lean: more than being a proof assistant, it can also be used as a programming language, making it a powerful tool to both mathematicians and computer scientists. Although Lean has this dual feature, this is not the case with every proof assistant. Lean can assume the role of a programming language because it is based on *Dependent Type Theory*. In simple terms, this means that every expression in Lean receives a type and this type can depend on additional information. Dependent Type Theory makes it possible to create more expressive and precise programs and this is why

(16) A list of Lean related events can be accessed in `https://leanprover-community.github.io/events.html`.

proof assistants based on Dependent Type Theory are of particular interest. How Dependent Type Theory is used in mathematical proofs is explained by the *Propositions as Types* paradigm.

Propositions as Types is a direct correspondence between mathematical logic and computer science and it is the basis of how proofs are constructed in Lean. In short, and as stated in *PlanetMath* [34],

> "Propositions are nothing but special types - namely, types whose elements are proofs."

In other words, this correspondence states that propositions are represented by types and proofs are represented by terms. To prove a proposition in Lean, we will always have to construct an object of a certain type. For example, if P is a proposition, then a proof of P will be a term of type P.

Logic		**Lean**
proposition P	$\Leftrightarrow$	type P
proof of P	$\Leftrightarrow$	term $p : P$

Table 3: Propositions as Types

For a more in-depth explanation accompanied by a historical explanation on proposition as types, refer to *Propositions as Types* [40] by Philip Wadler.

4.2.1 A crash course through the most important notions

Unfortunately, giving a detailed introduction to Lean is a very extensive topic and would exceed the scope of the dissertation. However, in order to guarantee that even inexperienced Lean users can follow the main ideas behind the code associated to the present dissertation, we provide a very condensed list of the most important commands appearing in the code.

In order to be able to communicate with Lean, we have to use certain commands that Lean recognizes and understands how to interpret. For example, to write definitions or results, we have to use the following commands:

1. `definition` and `inductive`: commands to define new notions and structures
2. `axiom`: one of the commands to define axioms
3. `lemma` and `theorem`: commands with which every result starts
4. `notation`: a command to introduce a custom notation

Once the relevant definitions have been introduced through these commands, we can begin a proof. In Lean4, we will use proofs known as *tactic-based proofs*. Such proofs always begin with the word `by` followed by a list of commands called *tactics*. A proof is concluded once the words "no goals" are displayed. The most important tactics to follow the code are:

5. `have`: tactic to introduce a new hypothesis
6. `apply`: tactic to use previously established results, such as lemmas or theorems
7. `rw`: a tactic to rewrite equalities and equivalences
8. `simp`: a tactic that can simplify straightforward parts of the proof
9. `exact`: a tactic used to close a goal with a specific result
10. `sorry`: a tactic used to conclude a proof, even if the goal is not closed. This is particularly useful to introduce auxiliary results that are purely Lean technicalities and that are not always worth proving from a mathematical point of view.

Even though there is a large number of tactics, it is often enough to know just a handful of them to get started. We hope that the above list will allow the inexperienced reader to follow the main ideas behind the code. To make the code as transparent as possible, we have used comments throughout the code. Comments in Lean are written between `/-` and `-/` or right after `--`.

Finally, just as it is usually the case with proving mathematical results, we might need to refer to results that other mathematicians have proved. These results are all combined in *Mathlib*, which can be viewed as a readily available Lean encyclopedia containing the most important definitions and results in areas such as number theory, algebra and category theory.

In what follows we assume that the reader already has some knowledge on how to use and manipulate Lean. If this is not yet the case, there are some excellent resources to get started with Lean.

1. For a very fun and interactive introduction to proofs in Lean, we recommend the *Natural Number Game* or *A Lean intro to Logic*, two games designed to teach the most basic notions of Lean4. Both games are in the online Lean game server repository.(17)
2. For a more theoretical and detailed approach, we refer to the documentation *Mathematics in Lean* [3] (more targeted towards mathematicians) and *Theorem proving in Lean* [4] (more targeted to computer scientists). Since Lean4 is a work in progress, both documentations are being updated and developed over time. Information on how to download Lean4 is explained in both documents.

With all of this at hand, we can now finally discuss our original formalization.

4.3 Formalizing the Star Combinatory Calculus and Shoenfield's functional interpretation

In this project we have used the code editor Visual Studio Code (in short, VSCode) with the current Lean4 version of Lean to write our code. The code is fully available on Github, a platform where most public contributions to Lean are uploaded and maintained. The code of the dissertation can be accessed at `www.github.com/LiaMalato/StarCombinatoryCalculus`.

Before getting started with the explanations around our formalization, we must make some disclaimers. Formalizing mathematics is not simply copying the content of a mathematical text. It involves a lot of adaptations and modifications. It also involves filling a considerable amount of gaps usually not mentioned in mathematical texts. A lot of so-called trivial aspects that are usually not worth mentioning from a mathematical point of view have to be mentioned and fully formalized in Lean. Without this, there will be missing information and this is often enough to hinder the conclusion of proofs. Hence, it is sometimes necessary to add extra results and definitions as we go, in order to be able to express every mathematical idea we need. An example of this is the Modus Ponens Rule, as we will later discuss.

We now provide a short overview of the main Lean files associated with the present dissertation, before discussing some parts of the code in greater detail.

1. *StarLanguage:* folder with the basic notions of $\mathscr{L}^{\omega}_{*}$ and PL^{ω}_{*} with the following files:
   a) *FiniteTypes.lean*: definitions and examples of finite types (Section 1.1)
   b) *Syntax.lean*: syntax of $\mathscr{L}^{\omega}_{*}$ (Section 1.2.1)

(17)The repository can be accessed under `https://adam.math.hhu.de`. Accessing the games does not require the users to install Lean on their computer.

   c) *Axioms.lean*: axiomatic of PL_*^ω and the Bounded Axiom of Choice (Section 1.2.2)
   d) *CombinatorialCompleteness.lean*: Combinatorial Completeness (Section 1.2.4)
2. *SHFunctInterp.lean*: Shoenfield's functional interpretation (Section 2.1)
3. *Soundness.lean*: Soundness Theorem (Section 3.1, future work).

Since explaining the entirety of the code would take up a considerable amount of space, we will only focus on the most relevant parts of the code: the Combinatorial Completeness Theorem, the herbrandized Shoenfield functional interpretation and some of the necessary preliminary notions. We will explain how the formalization was done and how some of the mathematical content had to be slightly adapted to fit the mold of a formalization in Lean. In the end, we will even explain how Lean helped us spot minor errors originally present in the main text.

4.3.1 The syntax

We begin by explaining how we implemented the syntax of $\mathscr{L}_*^\omega$ (Section 1.2.1), to then move on to the axiomatic of PL_*^ω (Section 1.2.2). The first formalization we will discuss is the formalization of the finite types from Definition 1.1.

Finite types. We chose the name `FType` for the finite types in Lean and we used the following inductive definition for them:

```
1 -- Definition of finite types
2 inductive FType
3 | ground:  FType                         -- the ground type G
4 | arrow:   FType → FType → FType         -- the function type
5 | star:    FType → FType                 -- the star type
```

This definition is to be read as follows: The finite types `FType` consist of three clauses. The ground type is a type (line 3), the function type called `arrow` receives two types and outputs a type (line 4) and the star type called `star` receives a type and outputs a type (line 5).[18] For example, if we want to write the type $\sigma \to \tau$, we will write `FType.arrow` σ τ. To not always have to write `FType` before defining a type, we use `open FType` and now we can simply write `arrow` σ τ. Notice how Lean also uses currying to express these definitions without the need for a cartesian product (as discussed in Remark 1.3).

We then used `def` and `notation` to make sure that the finite types would look more like the mathematical definition.

```
1 -- Notation for finite types
2 def G := ground
3 notation σ "⟶" τ => arrow σ τ
4 notation σ "⋆" => star σ
```

For example, line 4 states that we can now write σ ⋆ for the type σ^* instead of `star` σ. With this notation, a type such as $G \to G^*$ can directly be written as `G ⟶ (G ⋆)` instead of `arrow G (star G)`, making the code more mathematically intuitive.

Terms, formulas and base formulas. Analogously to the definition of types, we used inductive definitions for the terms and formulas of $\mathscr{L}_*^\omega$.

```
-- Definition of terms
inductive Term
| lcons:  String → Term               -- L-constants:     constants of L
| lfun:   String → Term               --                  function symbols of L
```

[18] It helps to read the colon "`:`" as the word "is" to better understand such definitions.

```
| pi                                -- combinators:     Π combinator
| sigma                             --                  Σ combinator
| sing                              -- star constants:  singleton
| bUnion                            --                  binary union
| iUnion                            --                  indexed union
| var:    String → Term             -- variables
| app:    Term → Term → Term        -- term application
```

This definition is similar to the previous one in terms of its structure. However, there are some differences. For example, the combinators and the star constants do not receive any input. Other differences are in the definition of the $\mathscr{L}$-constants and the variables. In all three cases, writing `String → Term` means that we have to give a string as input in order to receive a term. For example, to write a variable x in Lean, we will write `var "x"`, since strings in Lean are written between quotation marks. Analogously, a constant k is written `lcons "k"` and a function symbol f of $\mathscr{L}$ is written `lfun "f"`. Once again, we then introduced a custom notation. We used Π_1 and Σ_1 for the combinators, $\mathtt{s}_1$ for the singleton, $\cup_1$ for the binary union and $\mathtt{i}\cup_1$ for the indexed union. We then also created the following notation for the application of two terms:

```
notation t₁ "·" t₂ => app t₁ t₂      -- Notation for the application of terms
```

We chose to add the subscript $_1$ in the symbols related to $\mathscr{L}^{\omega}_{*}$, since most symbols are already internally defined in Lean and have a fixed associated meaning. This is the case for $\cup$, Σ, Π and many others. We then defined the formulas of $\mathscr{L}^{\omega}_{*}$ as follows:

```
-- Definition of formulas
inductive Formula
| rel:        String → List Term → Formula                 -- R(t₁, ..., tₙ)
| eq:         Term → Term → Formula                        -- t = q
| mem:        Term → Term → Formula                        -- t ∈ q
| not:        Formula → Formula                            -- ¬A
| or:         Formula → Formula → Formula                  -- A∨B
| unbForall:  String → Formula → Formula                   -- ∀x A
| bForall:    String → Term → Formula → Formula            -- ∀x∈t A
```

Once again, we defined notations for the formulas: $=_1$ for the equality symbol, $\in_1$ for the membership symbol, $\neg_1$ for the negation symbol and $\vee_1$ for the disjunction symbol. We also defined the following notations for the universal quantifiers:

```
notation "∀₁₁" => unbForall
notation "b∀₁₁" => bForall
```

For example, to write $\forall x \in t\ A$, we will write `∀₁₁ x t A` where `x` is a string, `t` a term and `A` a formula. We used the subscript $_{11}$ for both the unbounded and bounded universal quantifiers to distinguish them from their analogous definitions for tuples, given respectively by $\forall_1$ and $\mathtt{b}\forall_1$ that we defined afterwards. Using the classical abbreviations, we also created analogous definitions to obtain the defined symbols $\wedge_1$, $\rightarrow_1$, $\leftrightarrow_1$, $\exists_{11}$, $\mathtt{b}\exists_{11}$, $\exists_1$ and $\mathtt{b}\exists_1$. The subscript is again used to avoid confusion with other symbols in Lean.

Type checking An adaptation that had to be done is related to the way the language $\mathscr{L}^{\omega}_{*}$ is typed. The terms of the definition `Term` are not explicitly typed. To express the fact that terms have to be typed in a certain way, we use an auxiliary definition called `Term_TypeChecking`. We will now consider part of this definition. From here on, we will use an ellipsis "..." to indicate that we are not presenting the whole definition from the code.

```
inductive Term_TypeChecking : Term → FType → Prop
| tcLcons {k}:       Term_TypeChecking (lcons k) G
| tcPi {σ τ}:        Term_TypeChecking pi (σ ⟶ (τ ⟶ σ))
...
```

```
5  | tcApp {t₁ t₂ σ τ}: Term_TypeChecking t₁ (σ ⟶ τ) →
6                       Term_TypeChecking t₂ σ →
7                       Term_TypeChecking (app t₁ t₂) τ
```

In this definition, line 2 states that constants that stem from constants in $\mathscr{L}$ have type G. Line 3 states that the Π combinator has type $\sigma \to (\tau \to \sigma)$. Lines 5 to 7 states that if a term t_1 has type $\sigma \to \tau$ and a term t_2 has type σ, then their application has type τ. Every time it was necessary to specify the type of a term in the code, we used the above definition.

There are some general Lean aspects we can mention by looking at this definition. First, notice that its type is `Term → FType → Prop`. This means that it can take a term and a type as input to then output a proposition. In Lean, the type of the propositions is `Prop` and when an object has type `Prop` we can use it as a true or false statement in other results and proofs. For example, in line 3, the definition takes the term `pi` and the type `(σ ⟶ (τ ⟶ σ))` and outputs the proposition `Term_TypeChecking pi (σ ⟶ (τ ⟶ σ))`. Another important aspect is related to the use of curly brackets before the use of the colon. The information between curly brackets always corresponds to implicit arguments that Lean will try to infer when we call on a definition. For example, if we write `Term_TypeChecking.tcLcons` as in line 2, Lean does not need any other information to state the proposition `Term_TypeChecking (lcons _) G`. The underscore here is used to indicate that we did not specify the implicit argument. More on this later.

Other definitions. To guarantee that the syntax of $\mathscr{L}^{\omega}_{*}$ was accurately defined in Lean, we also included definitions such as term substitution, free variables, closed terms, base formulas, etc. These notions are all formalized in the code. Moreover, we also had to formalize various notions related to the use of term tuples.

4.3.2 The axioms

To easily show when axioms were universal closures of base formulas, we have defined them in three steps: we first defined the matrix of the axiom (a base formula), then we defined their universal closure and finally we defined them in the context of the axiomatic of the Star Combinatory Calculus. We will use Axiom AxC_1 as an example to showcase these steps.

```
1  -- The matrix of AxC₁
2  def AxiomC1_matrix (x₁ x₂ : String) : Formula :=      -- Given variables x₁ x₂:
3    ((Π₁·(var x₁))·(var x₂)) =₁ (var x₁)                -- (Π·x₁)·x₂ = x₁
4
5  -- AxC₁ as the universal closure of (Π·x₁)·x₂ = x₁
6  def AxiomC1 (x₁ x₂ : String) : Formula :=             -- Given variables x₁ x₂:
7    ∀₁₁ x₁ (∀₁₁ x₂ (AxiomC1_matrix x₁ x₂))              -- ∀x₁,x₂ [(Π·x₁)·x₂ = x₁]
8
9  -- AxC₁ in the context of the axiomatic of the Star Combinatory Calculus
10 inductive ProvableFrom : Set Formula → Formula → Prop
11 | AxC₁:    Γ ⊢ AxiomC1 x₁ x₂       -- ∀x₁,x₂ [(Π·x₁)·x₂ = x₁] is provable from Γ
12 ...
```

Notice that here we did not use curly brackets before the colon. We used parentheses. This is to indicate that the arguments are explicit and not implicit. We can only use these definitions if we provide the explicit arguments. Lean will not infer them.

In terms of the proofs we have formalized in Lean, `ProvableFrom` is the most valuable definition. It states that, given a set of formulas and a formula, we can associate a proposition about the provability of the formula from the set of formulas Γ. In this definition we have used the notation `Γ ⊢ A` to indicate that we can prove the formula `A` from the assumptions from Γ. Hence, `AxC₁` (line 11) states that given a set of formulas Γ and the formula `AxiomC1 x₁ x₂` (that is `∀x₁,x₂ [(Π·x₁)·x₂ = x₁]`), then we have the proposition `Γ ⊢ AxiomC1 x₁ x₂`,. In other words, `∀x₁,x₂ [(Π·x₁)·x₂ = x₁]` is provable from Γ.

Then, we also added a lemma stating that the formulas in the axioms could also be used for terms and not just variables. From a mathematical point of view, this comes from the Substitution Axiom Scheme and the Modus Ponens Rule. For example, for Axiom AxC_1 we created the following lemma:

```
lemma AxC₁_term {Γ : Set Formula} (t₁ t₂ : Term):
    Γ ⊢ ((Π₁·t₁)·t₂) =₁ t₁
```

Since in our discussion below we will also need the corresponding lemma for Axiom AxC_2, we present it here:

```
lemma AxC₂_term {Γ : Set Formula} (t₁ t₂ t₃ : Term):
    Γ ⊢ (((Σ₁·t₁)·t₂)·t₃) =₁ ((t₁·t₃)·(t₂·t₃))
```

Notice the use of both types of brackets in the hypotheses of the lemmas. We used curly brackets for the set of formulas Γ but parentheses for the terms. To invoke lemma `AxC₂_term` we have to give three terms as input but we do not need to specify the set of formulas. Hence, when writing `AxC₂_term t₁ t₂ t₃` Lean will try to infer what the set of formulas is. If Lean is not capable, we will write `@AxC₂_term Γ t₁ t₂ t₃` to make the implicit argument Γ explicit.

The other axioms were defined analogously, with adaptations made when necessary. The inference rules were then also defined in the context of the inductive definition `ProvableFrom`. For example, the Expansion Rule was defined as follows:

```
inductive ProvableFrom : Set Formula → Formula → Prop
| exp :  ∀ {A B},   Γ ⊢ A  →  Γ ⊢ B∨₁A         -- Expansion Rule
...
```

Writing `ProvableFrom.exp A B` (or simply `exp A B` after writing `open ProvableFrom`) gives the proposition `Γ ⊢ A → Γ ⊢ B∨₁A`, which corresponds to the usual Expansion Rule: given any two formulas A and B, if we can prove A from Γ, then we can also prove $B \vee A$ from Γ.

4.3.3 The Combinatorial Completeness Theorem

We now present the formalization of the Combinatorial Completeness Theorem (Theorem 1.1). We will first give the statement and then explain each line.

```
1 -- Combinatorial Completeness Theorem
2 theorem CombinatorialCompleteness
3 (x:String) (s:Term):                    -- Given a variable x and a term s,
4     ∀(t:Term),                          -- then for every term t we have
5     Γ ⊢ (((la x t).to_term)·s) =₁ (t.subst ([x]⟹[s]))  -- explained below
```

Line 5 corresponds to $\mathsf{PL}^\omega_* \vdash qs = t[{}^s/_x]$ from the mathematical statement of the theorem. We further explain this line:

- In Lean, PL^ω_* is represented by Γ.
- When we discussed the Combinatorial Completeness Theorem in section 1.2.4, we mentioned that the term q from the statement of the theorem would be represented by the notation $\lambda x.t$. In Lean, instead of writing q, we directly use the notation $\lambda x.t$, which is represented above by `(la x t).to_term`. Hence, the term application qs is written `((la x t).to_term)·s` in Lean. We will explain this point in greater detail below.
- The substitution of the variable x by the term s in the term t, that is $t[{}^s/_x]$, is written `t.subst ([x]⟹[s])` in our definition in Lean.[19]

[19] To also be able to deal with simultaneous substitutions, the substitution operator was defined to work with lists of variables and lists of terms instead of just individual variables and terms. Writing `t.subst ([x]⟹[s])` represents substituting the list containing just `x` by the list containing just `s` in `t`.

We now discuss how the term q was defined in Lean and how this definition is translated by `(la x t).to_term`.

Lambda notation. Just like in the mathematical proof of the Combinatorial Completeness Theorem, it was necessary to construct a term q that can be used in the proof of the theorem. But first we need to define the lambda notation used in the proof. The first step is to define an object such as λ that takes a variable and a term. This is done by the inductive definition `lambda` that takes a string `x` and a term `t`

```
-- Construction of the lambda notation for the proof
inductive lambda
| la (x : String) (t : Term): lambda
```

With this definition, if `x` is a string and `t` is a term, then we can use `lambda.la x t` (or simply `la x t` after using `open lambda`) to represent $\lambda x.t$. With this constructor, we could now define the term q from Definition 1.11. For technical reasons, the definition of the term q has been adapted to the following definition.

Definition 4.1. The term q is defined as follows:

a) If the term t is the variable x, then $q := \lambda x.x = \Sigma\Pi\Pi$.
b) If the term t is any other variable or if t is a constant, then $q := \lambda x.t = \Pi t$.
c) If the term t is the application of terms t_1 and t_2, then $q := \lambda x.(t_1 t_2) = \Sigma(\lambda x.t_1)(\lambda x.t_2)$.

In Lean, the term q is then represented by the following definition. Again, we only present parts of the definition.

```
-- Construction of the term q for the proof
def lambda.to_term : lambda → Term
| la x Π₁ => Π₁·Π₁                                  -- λx.Π = Π·Π
...
| la x (var y) => if x=y then ((Σ₁·Π₁)·Π₁)          -- λx.x = (Σ·Π)·Π
                  else (Π₁·(var y))                  -- λx.y = Π·y
| la x (t₁·t₂)
    => ((Σ₁·(la x t₁).to_term)·(la x t₂).to_term) -- λx.(t₁·t₂)=Σ·(λx.t₁)·(λx.t₂)
```

For example, when t is the term Π, then the term q is mathematically written as $\lambda x.\Pi$ and in Lean as `(la x Π₁).to_term`. In this definition, `=>` is part of the syntax of Lean when using `def` for definitions. In simple terms, it states, for example, that `(la x Π₁).to_term` is to be transformed into $\Pi_1 \cdot \Pi_1$ when the definition is applied. In more technical terms, this process is called *pattern matching*. It means that when Lean finds an input that matches the pattern on the left of the arrow, it evaluates to the result on the right.

Proof. Since certain parts of the proof are repetitive, we will only discuss some of them.

```
-- Proof of the Combinatorial Completeness Theorem
theorem CombinatorialCompleteness (x:String) (s:Term):
  ∀(t:Term), (Γ ⊢ ((((la x t).to_term)·s) =₁ (t.subst ([x] ⟹ [s])))) :=
by
  intro t                   -- Let t be a term.
  induction t with          -- We will prove the theorem by induction on t.
  | pi => ...                  -- Case 1: when t is the combinator Π
  | var y => ...               -- Case 2: Case when t is a variable y:
                                 -- a) when t is the variable x
                                 -- b) when t is a variable different from x
    ...
```

As in the mathematical proof, we prove the theorem by induction on the structure of t (line 6). We will discuss the three cases displayed above. In inductive proofs in Lean, the proof of each case starts after the symbol `=>` (as we are again using patternmatching).

Case 1: when t is the combinator Π. The goal displayed by Lean at the start of this case is

```
Γ ⊢ ((la x Π₁).to_term·s) =₁ Π₁.subst ([x]⟹[s])      -- goal displayed by Lean
```

We need to show that the formula `((la x Π₁).to_term)·s =₁ Π₁.subst ([x]⟹[s])` is provable from Γ. That is, we want to show that $(\lambda x.\Pi)s$ is equal to Π when x is substituted by s. The proof was done as follows:

```
1  -- Proof of the Combinatorial Completeness Theorem
2  | pi =>                         -- Case 1: when t is the combinator Π
3      rw [to_term]                   -- simplifies ((la x Π₁).to_term)·s to (Π₁·Π₁)·s
4      rw [Term.subst]                -- simplifies Π₁.subst ([x]⟹[s]) to Π₁
5      exact AxC₁_term Π₁ s           -- closes the goal by using AxC₁ for terms
```

In lines 3 and 4, we made two simplifications with the tactic `rw`. In line 3, we rewrote the definition `to_term`, so that Lean could use the appropriate case of definition 4.1. Since Π is a constant, Lean rewrote `(la x Π₁).to_term` as $\Pi_1{\cdot}\Pi_1$. Then, in line 4 we rewrote the definition `Term.subst`, which is the simultaneous term substitution in Lean. Here `Π₁.subst ([x]⟹[s])` says that we are replacing `x` by `s` in Π_1. But since Π_1 is a constant, there is nothing to substitute and the result of `rw [Term.subst]` is simply Π_1 itself. At this point the goal displayed by Lean is:

```
Γ ⊢ ((Π₁·Π₁)·s)=₁Π₁      -- new goal
```

But this is simply Lemma `AxC₁_term` we described previously using as arguments the terms Π_1 and `s`. Hence, we write `exact AxC₁_term Π₁ s`, which concludes the proof of this case in line 5.

Case 2: when t is a variable. In Lean, saying that t is a variable is not enough because variables depend on strings (in short, on their names). This is why saying "when t is a variable" will here be represented by "when t is the variable y", which in Lean becomes `var y`. We then need to distinguish two cases: when $x = y$ and when $x \neq y$ (written `x = y` and `¬(x = y)` in Lean). To do so, we use `by_cases h: x = y` to split the hypothesis `h` into two subgoals:

```
1  | var y =>                       -- Case 2: when t is a variable y
2    by_cases h: x = y
3    .                              -- a) when t is the variable x: (x=y)
4    .                              -- b) when t is a different variable: ¬(x=y)
```

a) When t is the variable x. The proof for this case starts off in line 3 and ends in line 9.

```
1  | var y =>                       -- Case 2: when t is a variable y
2    by_cases h: x = y
3    . rw [to_term]                 -- a) when t is the variable x: (x=y)
4      simp [h]
5      rw [Term.subst]
6      rw [helper_cc4]
7      have H1 := @AxC₂_term Γ Π₁ Π₁ s    -- introduction of a new hypothesis H1
8      have H2 := @AxC₁_term Γ s (Π₁·s)    -- introduction of a new hypothesis H2
9      exact eq_trans H1 H2
```

The goal at the beginning of line 3 is given by

```
Γ ⊢ ((la x (var y)).to_term·s)=₁(var y).subst ([x]⟹[s])        -- initial goal
```

Similarly to the previous case, we use `rw [to_term]`. By the definition of `to_term`, we obtain:

```
Γ ⊢ ((if x = y then (Σ₁·Π₁)·Π₁ else Π₁·var y)·s)=₁(var y).subst ([x]⟹[s])
```

Since hypothesis `h` states `x = y`, we can simplify the above goal with this hypothesis by writing `simp [h]` (line 4). We then once again use `rw [Term.subst]` in line 5 and obtain the new goal:

$$\Gamma \vdash (((\Sigma_1\cdot\Pi_1)\cdot\Pi_1)\cdot \mathtt{s}) =_1 \mathtt{(HashMap.ofList\ [(y,\ s)]).findD\ y\ (var\ y)}$$

As can be seen, we are now dealing with `HashMap.ofList [(y, s)]`. The reason why HashMap appears after applying `rw [Term.subst]` is due to the fact that the definition of term substitution uses HashMaps. In short, a HashMap is used to store pairs of keys and values, where keys are associated to values. It is almost like a function with an input and output. Even though some properties valid for functions are not valid for HashMaps, HashMaps are still useful for definitions such as substitutions. When writing `HashMap.ofList [(y, s)]`, Lean is creating a HashMap out of the list with the single pair `(y, s)`.

With this in mind, `(HashMap.ofList [(y, s)]).findD y (var y)` now means that Lean will try to retrieve the value associated with the key `y`. If `y` exists in the HashMap, then we obtain `s` (its key). If `y` does not exist, then Lean returns `var y`. Because of how HashMaps are internally defined in Lean, we use the following auxiliary lemma to simplify the proof as we just described:

```
lemma helper_cc4 (y : String) (s t : Term):
    ((HashMap.ofList [(y, s)]).findD y t) = s
```

With this lemma, by using `rw [helper_cc4]` (line 6), we obtain a new goal given by $\Gamma \vdash (((\Sigma_1\cdot\Pi_1)\cdot\Pi_1)\cdot \mathtt{s}) =_1 \mathtt{s}$. Analogously to the mathematical proof, we now use the lemma $\mathtt{AxC_1\_term}$ with the terms Π_1 and `s` to obtain $(((\Sigma_1\cdot\Pi_1)\cdot\Pi_1)\cdot \mathtt{s}) =_1 (\Pi_1\cdot \mathtt{s})\cdot\Pi_1\cdot \mathtt{s}$ (hypothesis `H1`, line 7) and the lemma $\mathtt{AxC_1\_term}$ with the terms `s` and $\Pi_1\cdot \mathtt{s}$ to obtain $\Gamma \vdash ((\Pi_1\cdot \mathtt{s})\cdot\Pi_1\cdot \mathtt{s}) =_1 \mathtt{s}$ (hypothesis `H2`, line 8). Notice that when defining these hypothesis, we used `@` before the lemma to specify which set of formulas Lean should use for the hypothesis. Then it suffices to use the transitivity of the equality given by this lemma:

```
lemma eq_trans {t_1 t_2 t_3 : Term}:
    (Γ ⊢ t_1 =_1 t_2) → (Γ ⊢ t_2 =_1 t_3) → (Γ ⊢ t_1 =_1 t_3)
```

We conclude the proof of this part by using `exact eq_trans H1 H2`, that is by using lemma `eq_trans` with the assumptions `H1` and `H2`.

b) When t is a different variable than x. This case starts off very similar to the previous case. We begin by rewriting `to_term` and then by simplifying the goal with the hypothesis `h` which is now `h: ¬(x = y)`.

```
1    . rw [to_term]       -- b) when t is a different variable: ¬(x = y)
2      simp [h]
3      rw [Term.subst]
4      rw [helper_cc3]
5      exact AxC_1_term (var y) s
```

After rewriting `to_term` in line 1, the goal is given by:

$$\Gamma \vdash (\mathtt{if}\ x = y\ \mathtt{then}\ (\Sigma_1\cdot\Pi_1)\cdot\Pi_1\ \mathtt{else}\ ((\Pi_1\cdot \mathtt{var}\ y)\cdot s) =_1 (\mathtt{var}\ y).\mathtt{subst}\ ([x]\Longrightarrow[s]))$$

Since we are in the case where `h:¬(x = y)`, by using `simp [h]` in line 2 the goal is simplified to:

$$\Gamma \vdash ((\Pi_1\cdot \mathtt{var\ y})\cdot \mathtt{s}) =_1 \mathtt{(var\ y).subst\ (HashMap.ofList\ [(x,\ s)])}$$

After using `rw [Term.subst]` as before, at the end of line 3 the goal is:

$$\Gamma \vdash ((\Pi_1\cdot \mathtt{var\ y})\cdot \mathtt{s}) =_1 \mathtt{(HashMap.ofList\ [(x,\ s)]).findD\ y\ (var\ y)}$$

To simplify `(HashMap.ofList [(x, s)]).findD y (var y)`, we created this auxiliary lemma:

```
lemma helper_cc3 (x y : String) (s : Term):
    ((HashMap.ofList [(x, s)]).findD y (var y)) = (var y)
```

After using `rw [helper_cc3]` in line 4, the goal becomes $\Gamma \vdash ((\Pi_1\cdot \mathtt{var\ y})\cdot \mathtt{s}) =_1 \mathtt{var\ y}$ and it suffices to use the tactic `exact` with the lemma $\mathtt{AxC_1\_term}$ with the terms `var y` and `s` (line 5).

The other cases of this proof are either very similar to the above cases or involve other simple auxiliary lemmas. We now focus on how the herbrandized Shoenfield functional interpretation was formalized.

4.3.4 The herbrandized Shoenfield functional interpretation

Formalizing the herbrandized Shoenfield functional interpretation from Section 2.2 perfectly shows that a formalization is not a straight-forward translation of the mathematical content. In order to easily prove results involving the functional interpretation, we chose to adapt the definition as we will now explain. Every interpretation will be represented by a triple. For example, an interpretation $\forall a \exists b A_{SH}(a, b)$ will be represented by the triple (a, b, A_{SH}). To say that the interpretation of a formula A is given by $A^{SH} \equiv \forall a \exists b\ A_{SH}(a, b)$, we will write it as the proposition `SH_int A (a,b,A_SH)`. With this in mind, the herbrandized Shoenfield functional interpretation (Definition 2.1) is implemented as follows:

```
1  -- Herbrandized Shoenfield functional interpretation
2  inductive SH_int: Formula → (List String × List String × Formula) → Prop
3  | base:  (h : isBase A) →                    -- Interpretation of base formulas
4           SH_int A ([],[],A)
5  | disj:  (h : disj_lists_four a b c d) →     -- Interpretation of a disjunction
6           SH_int A (a,b,A_SH) →
7           SH_int B (c,d,B_SH) →
8           SH_int (A∨₁B) (a++c,b++d,(A_SH ∨₁ B_SH))
9  | neg {f a': List String}:
10          (h : disj_lists_two a b) →          -- Interpretation of a negation
11          SH_int A (a,b,A_SH) →
12          SH_int (¬₁A)
13                (f,a',(b∃₁ a (a'.tt) (¬₁A_SH)).subst (b⟹((f.tt)⊙(a.tt))))
14 ...
```

For the interpretation of formulas of the form $\forall x\ A(x)$ and $\forall x \in t\ A(x)$, refer to the file *SHFunctInterp* on Github. As mentioned above, the definition `SH_int` takes in a formula and a triple and outputs a proposition. This is why its type is given by

```
Formula → (List String × List String × Formula) → Prop
```

Notice that a triple such as (a, b, A_{SH}) has type `List String × List String × Formula` because a and b represent tuples of variables and must therefore have type `List String`. Although for technical reasons it is easier to work with triples, we can always rewrite triples as formulas by using the definition `Recreate` which we have defined in the file *SHFunctInterp*. For example, by applying `Recreate` to the triple (a, b, A_{SH}) we obtain $\forall a \exists b A_{SH}$. We now discuss each of the cases displayed in the above formalization.

Interpretation of base formulas (lines 3 to 4). To state that a formula A is a base formula, we introduce the hypothesis `h : isBase A` in line 3. From this hypothesis, Lean outputs the proposition `SH_int A ([],[],A)`, where `[]` represents the empty list, *i.e.* an empty tuple of variables (line 4). Hence, the interpretation of a base formula A_b is given by the triple $([\,], [\,], A_b)$, which simply represents the formula A_b.

Interpretation of disjunctions (lines 5 to 8). Let A and B be two formulas. In the mathematical definition, we assume to have their interpretations, respectively given by $\forall a \exists b A_{SH}(a, b)$ and $\forall c \exists d A_{SH}(c, d)$. In Lean, line 6 is to be read as "if the interpretation of A is given by (a, b, A_{SH})" and line 7 as "if the interpretation of B is given by (c, d, B_{SH})". The hypothesis that a, b, c, d are all disjoint tuples of variables is given by the hypothesis `h : disj_lists_four a b c d` in line 5. Line 8 then states that the interpretation of $A \vee B$ is given by the triple $(ac, bd, A_{SH} \vee B_{SH})$, where ac represents the concatenation of the lists a and c (analogously for bd). In Lean, the concatenation of lists is written as `++`.

Interpretation of negations (lines 9 to 13). Since in the negation case there are new tuples of variables that are created, we must introduce their strings at the beginning. This is done by writing `f a' : List String`. Similarly to the disjunction case, we introduce an hypothesis to say that the tuples of variables are disjoint (line 10) together with the interpretation of A (line 11). Line 13 represents the interpretation of $\neg A$ that we now explain. Line 13 states that the universal variable is given by f and the existential variable is given by a'. We now explain that `(b∃₁ a (a'.tt) (¬₁A_SH)).subst (b⟹((f.tt)⊙(a.tt)))` represents the lower *SH*-formula $\exists a \in a' \neg A_{SH}(a, fa)$. To do so, we first need to explain certain symbols.

We previously mentioned that we had defined the bounded existential quantifiers `b∀₁` and `b∃₁` to accept tuples of variables and tuples of terms. In fact, they accept tuples of strings (*i.e.* lists of strings that can easily be turned into variables) and tuples of terms (*i.e.* lists of terms). Hence, to represent a bounded existential quantification we write `b∃₁ x t` where `x : List String` and `t : List Term`. However, we cannot yet write `b∃₁ a a'` because `a'` has type `List String` instead of `List Term`. To overcome this, we created the definition `tt` that upgrades lists of strings into lists of terms. By writing `a'.tt`, Lean now interprets a' as a list of terms, that is, a tuple of terms, and we can write `b∃₁ a (a'.tt)` to represent $\exists a \in a'$.

Next, we discuss `(f.tt)⊙(a.tt)`. Just as was done with a' to turn a tuple of strings into a tuple of terms, we turn f and a into tuples of terms by writing `f.tt` and `a.tt`. However, due to the way the application of terms was defined (refer to the definition `Term`), we cannot perform an application between `f.tt` and `a.tt`. Our definition of term application is not valid for tuples of terms. For this reason, we extended the application of terms to the application of term tuples and used the notation $\odot$ for it. Now, the application fa can be represented by `(f.tt)⊙(a.tt)`.

Finally, since the substitution operator had been defined to also deal with simultaneous substitutions, we can use this same operator to replace the existential variable b by the application fa in $\neg A_{SH}(a, fa)$. In conclusion, $\exists a \in a' \neg A_{SH}(a, fa)$ really is represented by `(b∃₁ a (a'.tt) (¬₁A_SH)).subst (b⟹((f.tt)⊙(a.tt)))`. Notice that, for technical reasons, we applied the substitution of b by fa to the whole lower *SH*-formula instead of just to `¬₁A_SH`.

With this definition, we were able to formalize all the examples and results from Section 2.2. These, together with the above definition, are in the file *SHFunctInterp*. For a brief overview of the formalization of the Soundness Theorem as future work, refer to Appendix D.

4.3.5 Notable differences

We will now discuss some of the differences between the mathematical content of the dissertation and its formalization.

Rigour. Probably the biggest difference is the extra rigour we needed to include to formalize results in Lean. Even though this was not always problematic, it often created tedious and time consuming extra workload. On the bright side, at the end of the formalization process we created reliable and computer-verifiable proofs. An example is the Modus Ponens Rule. From a mathematical standpoint, it is not worth proving that we can retrieve this rule from our axiomatic, since we know that Shoenfield's Calculus is complete for first-order logic (refer to section 1.2.2). However, in Lean we need to formalize the Modus Ponens Rule to use it.

Type checking. As discussed, the way types were implemented in the current dissertation was slightly different in comparison to the mathematical definition. For example, the Lean definition of terms did not directly involve finite types. Rather, we added an extra definition to check that the types were all correct. This did not affect the formalization of the definitions, results and proofs. In the code, every time it was necessary to explicitly use the types of the terms we referred to this definition.

Auxiliary lemmas. During the formalization process, we had to add some auxiliary lemmas to deal with certain Lean related technicalities. Usually, these are not worth proving from a mathematical point of view. An example of such a result was `helper_cc4` that was used in the proof of the Combinatorial Completeness Theorem. The reason it is not worth proving this specific result is that it depends on how the Lean type `HashMap` is internally defined.

4.3.6 Saved by the proof assistant

During the formalization process, Lean was able to detect some small draft errors we had failed to notice in the mathematical part of the dissertation. The first concerned an example related to type checking. In Example 1.4, the types in one of the applications were wrong. By referring to the errors displayed by Lean, we were able to quickly correct the mistake. The second concerned Example 2.2. While we were proving the interpretation of the example in Lean, at first we were not able to conclude the proof in Lean. From Lean's infoview, we were then able to notice a missing negation symbol.

4.4 State of affairs: the future of proof assistants

As can be seen, proof assistants such as Lean are growing on mathematicians all around the world in a collaborative effort to formalize mathematics. But up to now we have only discussed the current state of affairs. We conclude the second part of the dissertation by making some cautious predictions on the future of mathematics when it comes to proof assistants. We will do so from three different perspectives.

Research. As mentioned at the beginning of this section, since formalizations involve such a high degree of rigor and precision, some mathematicians already believe that the future of mathematical research will involve formalizations of results. On another note, it is also probable that the time needed to peer-review a paper will become significantly reduced, since formalizations already ensure the correctness of the proofs.

Education. The need to provide every detail in a proof assistant such as Lean forces the user to truly understand every step of a proof. Various universities are therefore already introducing the study of mathematics through the use of proof assistants on an undergraduate level. For example, Prof. Dr. Kevin Buzzard from Imperial College London has not only taught such classes, but has also supervised PhD theses which actively use Lean to formalize results. Imperial College London is currently ranked as the 4th best university in Europe in the field of mathematics.(20) Universities that fail to involve proof assistants in the teaching of mathematics might begin to fall behind on universities that choose to adopt this proof assistant based approach. The mathematician of tomorrow needs the tools of today.

Artificial intelligence. Finally, with the emergence of large language models associated with artificial intelligence, it becomes urgent to reconsider the way mathematics is looked upon and researched. Mathematicians will need to become acquainted with proof verification and use artificial intelligence as a tool to do so. Even though artificial intelligence is still flawed in terms of mathematics, this might very soon change and create a gap between mathematicians who embrace proof assistants and those who do not.

(20) According to QS World University Rankings of 2024 [35].

Conclusion

In the present dissertation, we succeeded to prove a new and generalized version of Herbrand's Theorem in the broader context of higher order logic with finite types PL^{ω}_{*}, instead of the usual setting of first-order logic PL. It became noteworthy that, by doing so, a new principle appeared, a principle that is not even visible (nor can it be stated) when remaining in PL. When exploring higher types, the Bounded Axiom of Choice $\mathsf{bAC}^{\omega}_{*}$ revealed itself in the hypothesis of Herbrand's Theorem. And what was even more fascinating was that the original statement of Herbrand's Theorem, together with its generalized version, were direct consequences of our functional interpretation and the Soundness Theorem associated to it.

In order to reach these final conclusions, the study of Herbrand's Theorem required us to first study the Star Combinatory Calculus together with a herbrandized version of Shoenfield's functional intepretation. After having defined and discussed the most relevant aspects of the Star Combinatory Calculus such as its confluency and strongly normalizing properties, we shifted our attention to functional interpretations. Using as the main reference the article *A herbrandized functional interpretation of classical first-order logic* [21] by Ferreira and Ferreira where they defined a herbrandized version of Shoenfield's functional interpretation to interpret PL into PL^{ω}_{*}, we extended their work to interpret PL^{ω}_{*} into PL^{ω}_{*} by adapting their functional interpretation. And it is precisely from this adaptation that we were able to prove the Soundness Theorem and, ultimately, the original and generalized versions of Herbrand's Theorem.

After having given a mathematical framework to the dissertation, we delved into the world of the proof assistant Lean with the final goal of formalizing the Combinatorial Completeness Theorem and our functional interpretation. In order to do so, it became necessary to formalize many underlying notions. Some of these notions were straightforward to implement, while others required a fascinating new approach to mathematics: we had to abolish the word *obvious* and replace it with a very careful treatment of a significant amount of notions that mathematicians usually take for granted when discussing more advanced topics such as those discussed in the present dissertation. As a result, we had to be much more rigorous in our handling of the concepts in Lean compared to what is usually required when writing mathematical texts. But, after the initial shock related to this newly required rigour, it became possible to formalize the relevant definitions and results to then prove the Combinatorial Completeness Theorem.

Future work

The discussion presented in this dissertation has laid the foundation for avenues for future work. From a mathematical point of view, the Star Combinatory Calculus appears naturally in the context of Diller-Nahm interpretations. There is recent ongoing work by Fernando Ferreira and Paulo Oliva on new functional interpretations in which this calculus is used. These developments could be an opportunity for further research. From a proof assistant perspective, it would be interesting to expand the work that has already been developed in Lean. There are still many aspects and proofs related to the Star Combinatory Calculus that can be formalized in Lean and that have not yet been formalized in any other proof assistant. In particular, it would be of great interest to formalize strong results such as the confluency and strong normalization theorems of the Star Combinatory Calculus. On another note, formalizing the generalized version of Herbrand's Theorem in Lean, one of the original contributions of the present dissertation, would serve as a valuable next step for further exploration.

Final considerations

Throughout the last section of this dissertation, we have mentioned various benefits of the use of proof assistants and this from different perspectives, namely from a research and an education point of view. With all these benefits at hand, we could ask why not more mathematicians use proof assistants such as Lean? What is the main hurdle nowadays? Unfortunately, there are many reasons explaining why the use of proof assistants is taking so much time to be seen as a standard practice. On the one hand, learning to use a proof assistant requires time and effort. The learning curve for proof assistants is still very steep. Before being able to translate one's work into the language of a proof assistant, it is necessary to invest a significant amount of time to learn the basics. It is also necessary to make a significant amount of mistakes to become at ease with the proof assistant. However, once this is overcome it becomes very rewarding to know that our work is actively contributing to a new era of mathematics.

In the year 2024, proof assistants are certainly not for everyone. There are still too many hurdles and inconveniences which cannot be ignored. However, these are becoming less and less with the ever growing community of mathematicians (such as Peter Scholze and Terence Tao) and universities (such as Imperial College in London) adhering to the use of proof assistants. Additionally, computer programs in general have long helped mathematicians develop proofs of all kinds (such as the first attempt at a computational proof of the Four Color Theorem).

In the 21st century, it is therefore no longer possible to ignore the impact that computers have had and will have on the development of mathematics. And, with the emergence and current development of artificial intelligence, it becomes even more urgent to embrace the use of new technologies when teaching or researching in mathematics. Almost a century ago, Kurt Gödel, Alonzo Church and Alan Turing set new standards for the future of mathematics. However, nearly a century later, a large part of the mathematical community still indulges in a certain *status quo* on how mathematics should be practiced.

How much longer should we accept this *status quo*? And how much longer can most of us continue to disregard the benefits of using technologies such as proof assistants? We, as a community, have to act. We have to change, or at least adapt, our ways. Ultimately, we don't have to stop using pen and paper to do mathematics, but the time has come to truly start *lean*ing into the future of mathematics.

References

[1] Jesse Alama and Johannes Korbmacher. “The Lambda Calculus”. In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta and Uri Nodelman. Winter 2023. Metaphysics Research Lab, Stanford University, 2023. URL: `https://plato.stanford.edu/archives/win2023/entries/lambda-calculus/`.

[2] Jeremy Avigad and Solomon Feferman. “Gödel’s Functional ("Dialectica") Interpretation”. In: *Handbook of Proof Theory*. Ed. by Samuel R. Buss. Elsevier, 1998, pp. 337–405.

[3] Jeremy Avigad and Patrick Massot. *Mathematics in Lean*. 2024. URL: `https://www.leanprover-community.github.io/mathematics_in_lean/mathematics_in_lean.pdf`.

[4] Jeremy Avigad and Patrick Massot. *Theorem Proving in Lean*. 2024. URL: `https://leanprover.github.io/theorem_proving_in_lean4/print.html`.

[5] Jeremy Avigad and Leonardo de Moura. *The Lean Theorem Prover (system description)*. 2015. URL: `http://www.contrib.andrew.cmu.edu/~avigad/Papers/lean_system.pdf`.

[6] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[7] Andrej Bauer. *The Dialectica interpertation in Coq*. 2011. URL: `https://math.andrej.com/2011/01/03/the-dialectica-interpertation-in-coq/`.

[8] Paul Bernays. “Über den Zusammenhang des Herbrandschen Satzes mit den neueren Ergebnissen von Schütte und Stenius”. In: *Proceedings of the International Congress of Mathematicians 1954*. Groningen and Amsterdam: Noordhoff and North-Holland, 1957.

[9] Katalin Bimbo. *Combinatory Logic: Pure, Applied and Typed*. Chapman and Hall/CRC, 2011.

[10] Ana Borges. “On the herbrandised interpretation for nonstandard arithmetic”. master’s thesis. Instituto Superior Técnico, 2016.

[11] Nicolas G. de Bruijn. “The Mathematical Language Automath, its Usage, and Some of its Extensions”. In: *Studies in Logic and the Foundations of Mathematics*. Ed. by R.P. Nederpelt, J.H. Geuvers, and R.C. de Vrijer. Vol. 133. 1994, pp. 73–100. URL: `https://www.sciencedirect.com/science/article/pii/S0049237X08702003`.

[12] Horatiu Cheval. *Formalized proof mining*. 2021. URL: `https://github.com/hcheval/formalized-proof-mining/tree/master`.

[13] Alonzo Church. “A formulation of the simple theory of types”. In: *Journal of Symbolic Logic* 5.2 (1940), pp. 56–68.

[14] Alonzo Church. “A set of postulates for the foundation of logic”. In: *Annals of Mathematics* 33.2 (1932), pp. 346–366.

[15] Alonzo Church. “A set of postulates for the foundation of logic”. In: *Annals of Mathematics* 34.4 (1933), pp. 839–864.

[16] Alonzo Church. “An unsolvable problem of elementary number theory”. In: *American Journal of Mathematics* 58.2 (1936), pp. 345–363.

[17] Burton Dreben and Stål Aanderaa. “Herbrand analyzing functions”. In: *Bulletin of the American Mathematical Society* 70 (5 1964), pp. 697–698.

[18] Burton Dreben, Peter Andrews, and Stål Aanderaa. “False lemmas in Herbrand”. In: *Bulletin of the American Mathematical Society* 69.5 (1963), pp. 699–706.

[19] Burton Dreben and John Denton. “A supplement to Herbrand”. In: *The Journal of Symbolic Logic* 31.3 (1966), pp. 393–398.

[20] Fernando Ferreira. "A most artistic package of a jumble of ideas". In: *Dialectica* 62.2 (2008), pp. 205–222.

[21] Fernando Ferreira and Gilda Ferreira. "A herbrandized functional interpretation of classical first-order logic". In: *Archive for Mathematical Logic* 56.5-6 (2017), pp. 523–539.

[22] Fernando Ferreira and Paulo Oliva. "Bounded functional interpretation". In: *Annals of Pure and Applied Logic* 135.1-3 (2005), pp. 73–112.

[23] Jaime Gaspar. "Em torno das interpretações funcionais da aritmética". master's thesis. University of Lisbon, Lisbon, Portugal, 2007.

[24] Philipp Gerhardy and Ulrich Kohlenbach. "Extracting Herbrand Disjunctions by Functional Interpretation". In: *Archive for Mathematical Logic* 44 (2005), pp. 633–644.

[25] Kurt Gödel. "On a hitherto unutilized extension of the finitary standpoint". In: *Journal of Philosophical Logic* 9.2 (1980), pp. 133–142.

[26] Kurt Gödel. "Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes". In: *Dialectica* 12.3 (1958), pp. 280–287.

[27] Jacques Herbrand. *Recherches sur la théorie de la démonstration*. 1930.

[28] Jan W. Klop. "Term Rewriting Systems". In: *Handbook of Logic in Computer Science* 2 (1993), pp. 1–116.

[29] Jan W. Klop. "Term Rewriting Systems: From Church-Rosser to Knuth-Bendix and beyond". In: *Automata, Languages And Programming*. CWI, 1990, pp. 350–369.

[30] Georg Kreisel. "Interpretation of analysis by means of constructive functionals of finite types". In: *Constructivity in Mathematics* (1959). Ed. by Arend Heyting, pp. 101–128.

[31] Max Newman. "On Theories with a Combinatorial Definition of "Equivalence"". In: *Annals of Mathematics* 43.2 (1942), pp. 223–243.

[32] Tobias Nipkow, Lawrence C. Paulson, and Makarius Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*. 2024. URL: `https://www.isabelle.in.tum.de/doc/tutorial.pdf`.

[33] Christine Paulin-Mohring. *Inductive Definitions in the Coq Proof Assistant*. Course notes from the 2007 LASER Summer School. 2007. URL: `https://www.lri.fr/~paulin/LASER/course-notes.pdf`.

[34] PlanetMath. *Propositions as Types*. URL: `https://www.planetmath.org/111propositionsastypes`.

[35] QS World University Rankings. *QS World University Rankings by Subject 2024: Mathematics (Region: Europe)*. URL: `https://www.topuniversities.com/university-subject-rankings/mathematics?region=Europe`.

[36] Joseph R. Shoenfield. *Mathematical Logic*. Addison-Wesley, 1967.

[37] William Tait. "Intensional Interpretations of Functionals of Finite Type I". In: *The Journal of Symbolic Logic* 32.2 (1967), pp. 198–212.

[38] William Tait. "Normal Form Theorem for Bar Recursive Functions of Finite Type". In: *Studies in Logic an the Foundations of Mathematics*. Ed. by Jens E. Fenstad. Vol. 63. 1971, pp. 353–367.

[39] Anne Troelstra. *Basic Proof Theory*. Ed. by Helmut Schwichtenberg. New York: Cambridge University Press, 2000.

[40] Philip Wadler. "Propositions as types". In: *Communications of the ACM* 58.12 (2015), pp. 75–84. URL: `https://dl.acm.org/doi/pdf/10.1145/2699407`.

Appendices

Here we will present the appendices that were mentioned throughout the dissertation. The idea of these appendices is to complement the content of the dissertation by providing extra information to give a broader context to some of the studied notions.

A A digression on Lambda Calculus

A.1 A short overview

As mentioned in Remark 1.17, the notation $\lambda x.t$ is not a primitive symbol of the language $\mathscr{L}^{\omega}_{*}$, but a way of representing the term q in the Combinatorial Completeness Theorem. Nevertheless, it is possible to take a different approach by letting $\lambda x.t$ be a primitive symbol. This is the case when, instead of studying the Combinatory Calculus, we study the *Lambda Calculus*. In a plain and concise manner, and following the approaches of [9] and [1], the language $\mathscr{L}^{\lambda}$ of the Lambda Calculus consists of countably many variables, a binary operation *application* (similar to the application encountered in $\mathscr{L}^{\omega}_{*}$) and the *abstraction* operator λ. The terms of this language are defined as follows.

Definition A.1. The *terms* of the (Untyped) Lambda Calculus are defined inductively as follows:

a) All variables are terms.
b) If x is a variable and t is a term, then $\lambda x.t$ is a term.
c) If t and q are terms, then so is their application tq.

Remark A.1. Although the syntax of the Lambda Calculus is captured by this definition, there remains much more to elaborate about this calculus. As the Lambda Calculus is an extensive topic, our intention here is not to describe all of its facets. Instead, it is to consider some interesting remarks about the calculus, focusing on those that are relevant for this dissertation. A more developed and detailed approach can be found in [9] and [1]. ⊠

Interestingly enough, the Lambda Calculus and the Combinatory Calculus are intertwined. This can be seen through various illustrations. First, the notions of λ-abstraction and β-conversion in the Lambda Calculus are analogous to those of the Combinatory Calculus: λ-abstraction remains responsible for defining functions and β-conversion for evaluating said functions. Secondly, the combinators Π and Σ from the Combinatory Calculus can also be defined in the Lambda Calculus. As a matter of fact, a combinator in $\mathscr{L}^{\lambda}$ is defined to be a term with no free variables. Relying on the above definition (Definition A.1), the combinators Σ and Π are defined by

$$\Sigma :\equiv \lambda x, y, z.(xz(yz)) \tag{S}$$

$$\Pi :\equiv \lambda x, y.x. \tag{P}$$

Using this definition, we are able to retrieve the Axioms AxC_1 and AxC_2. Indeed, applying (S) to x, y, z, yields Σxyz and thus $xz(yz)$. Analogously, applying (P) to x and y yields Πxy and, in turn, x.

It is important to make a disclaimer at this point. As of now, the Lambda Calculus was discussed as if the Lambda Calculus was not typed. In reality, there are typed and untyped lambda calculi with very different properties. Here, we gave an introduction to the Untyped Lambda Calculus. We will now discuss how the Lambda Calculus emerged, why the untyped version was not always satisfactory and why, under certain circumstances, types became a requirement.

A.2 To type or not to type: a historical approach

From the previous discussion, it would seem that the Lambda Calculus is simply a formal system. In reality, lambda calculi are, in fact, a *family* of formal systems with a common foundational origin that emerged in the early 1930s when Alonzo Church published two papers both entitled *A set of postulates for the foundation of logic* (the first part [14] being published in 1932 and the second [15] in 1933). Straight from the title, it is possible to understand what the Lambda Calculus was supposed to be: a set of postulates for the foundation of logic. During the first years after the publications of both papers, Church's formal system seemed, not only to be fruitful, but also reliable. That was until 1935.

Unfortunately, the formal system described in those papers had a deficit which two students of Church, named Stephen C. Kleene and John B. Rosser, were able to identify in 1935: Church's original system was inconsistent. The inconsistency of the system is condensed in what is now known as the Kleene-Rosser paradox. Leaving the system as it was would make every formula provable in the system. To amend this, Church modified his original system by focusing only on functions, removing all other matters. By doing so, in 1936, Church was able to prove the consistency of his revised system in his article *An unsolvable problem of elementary number theory* [16]. This revised system is what is still known today as the Untyped Lambda Calculus and is what was just concisely described above (refer to Definition A.1).

Even though the *Untyped Lambda Calculus* was a breakthrough, there was an unsatisfactory and objectionable matter which led Church to extend his Untyped Lambda Calculus to create the Simply-typed Lambda Calculus in 1940 in his paper *A Formulation of the Simple Theory of Types* [13]. The problem resided in the fact that Untyped Lambda Calculus allowed for *self-application*, which means it was always possible to apply a term to itself. To better understand this concept, we first need to understand how terms are defined in the *Simply-typed Lambda Calculus*.

Just as in the Combinatory Calculus, in the Simply-typed Lambda Calculus every term receives a type. The types introduced by Church included a ground type and function types $\sigma \to \tau$ for types σ and τ (similarly to Definition 1.1, except for the star type). How terms are typed is again very similar.

Definition A.2. The *terms* of the Simply-typed Lambda Calculus are defined inductively as follows:

a) Any variable x^σ is a term.
b) If x^σ is a variable and t^τ is a term, then $(\lambda x^\sigma.t^\tau)^{\sigma\to\tau}$ is a term.
c) If $t^{\sigma\to\tau}$ and q^σ are terms, then so is their application $(tq)^\tau$.

The calculus based on these terms is called the Simply-typed Lambda Calculus. For sake of simplicity we will refer to it as typed Lambda Calculus. One of the main reasons to expand the Untyped Lambda Calculus to the Typed Lambda Calculus was to avoid self-application. In fact, in the Untyped Lambda Calculus, without the restriction of types, terms could be applied to any argument. The application didn't have to be meaningful. To give an easy to understand example, it is as if a function from the naturals to the naturals could have any real number as an argument.

Example A.1. To give a more technical example, consider the lambda term $(\lambda x.x)(\lambda x.x)$, where x is a variable. Notice that the term $\lambda x.x$ represents the identity function $x \mapsto x$ and the term $(\lambda x.x)(\lambda x.x)$ is the application of the identity function to itself. When no types are involved, self-application does not give rise to any issue. In fact, the application $(\lambda x.x)(\lambda x.x)$ would yield the result $\lambda x.x$. However, in the Typed Lambda Calculus, if σ is a type and x^σ

is a variable, then the above application would be typed as $(\lambda x.x)^{\sigma\to\sigma}(\lambda x.x)^{\sigma\to\sigma}$, which is not an admissible application anymore. A term such as $(\lambda x.x)^{\sigma\to\sigma}$ can only be applied to terms of type σ. For the application to be admissible, it would have to be of the form $(\lambda x.x)^{\sigma\to\sigma}(\lambda x.x)^{\sigma}$, which is not possible. $\boxtimes$

Hence, the Typed Lambda Calculus avoids situations of this nature, as the third clause of definition A.2 creates a restriction on how terms can be formed by ensuring that terms can only be applied to other terms of appropriate types.

B More on term rewriting systems

In Section 1.2.5, we saw that the Star Combinatory Calculus is a term rewriting system with four rules called conversions. Moreover, we studied various notions that were not only valid for the Star Combinatory Calculus but for any term rewriting system. In said section we informally stated what such a term rewriting system is, but, if we wish to be more precise, term rewriting systems can be defined as follows.

Definition B.1. A *term rewriting system (TRS)* is a pair $(\mathscr{S}, \mathscr{C})$ where $\mathscr{S}$ is some signature, listing the function symbols (together with their arities) and the constant symbols, and where $\mathscr{C}$ is a set of *conversions* of the form $t \rightsquigarrow q$, for terms t and q.

A very *natural* example of this definition is the following term rewriting system.

Example B.1. The natural numbers together with the addition and multiplication can be defined as the following term rewriting system. Let $\mathscr{S} = \{+, \cdot, succ, 0\}$ with respective arities of $2, 2, 1, 0$. As in the beginning of Section 1.1, let $\mathscr{L}_{nat}$ be the language of Arithmetic, where $+$ stands for the addition, $\cdot$ for the multiplication, *succ* for the successor function and 0 for the zero of the natural numbers. Let x, y be variables and let $\mathscr{C}$ be the set consisting of the following conversions

1. $+(x, 0) \rightsquigarrow x$,
2. $+(x, succ(x)) \rightsquigarrow succ(+(x, y))$,
3. $\cdot(x, 0) \rightsquigarrow 0$,
4. $\cdot(x, succ(y)) \rightsquigarrow +(\cdot(x, y), x)$.

The pair $(\mathscr{S}, \mathscr{C})$ is an example of a term rewriting system. ⊠

We now consider a more interesting and very well-known example of a term rewriting system: the *Untyped Combinatory Calculus*. As the Untyped Combinatory Calculus is an extensive topic, our intention here is to provide a general overview, not a detailed one.

B.1 The Untyped Combinatory Calculus vs. the Star Combinatory Calculus

Example B.2. *(Untyped Combinatory Calculus)* The *Untyped Combinatory Calculus* originally consisted of a first-order language with a binary function symbol *app*, three constant symbols Σ, Π, I (the combinators) and variables $x, y, z, ...$ Just like for the Star Combinatory Calculus we write tq instead of $app(t, q)$ for the application of terms t and q and applications are again written associated to the left. Contrary to the Star Combinatory Calculus, the Untyped Combinatory Calculus does not involve types. The axioms for the combinators together with their respective conversions in the Untyped Combinatory Calculus are given in the following table.

Combinator axioms	Conversions
$\Sigma xyz = xz(yz)$	$\Sigma xyz \rightsquigarrow xz(yz)$
$\Pi xy = x$	$\Pi xy \rightsquigarrow x$
$Ix = x$	$Ix \rightsquigarrow x$

Table 4: Combinator axioms and conversions for the Untyped Combinatory Calculus

The first two axioms and the first two conversions are the same as for the Star Combinatory Calculus. In fact, the combinators that were present in the Star Combinatory Calculus were originally defined for the Untyped Combinatory Calculus. ⊠

As can be seen, the Untyped Combinatory Calculus has one more combinator compared to the Star Combinatory Calculus. To understand why the combinator I exists, it is necessary to understand that the Untyped Combinatory Calculus was created as a way to describe functions and hence it was more than natural to include a combinator to describe the identity function.

In Definition 1.2 we briefly mentioned that some authors prefer to use the letters S and K instead of Σ and Π, respectively. At first, the combinators were defined to be terms that do not contain variables. The letters S, K, I were used in the original definition of combinators and the choice of the letters was not arbitrary. Combinators were first published in 1920 by Russian logician Moses Shönfinkel in his paper *Über die Bausteine der mathematischen Logik.* Since Shönfinkel published his work related to the Untyped Combinatory Calculus mainly in German, the letters of the combinators were the first letters of certain German words:

- S for *Verschmelzungsfunktion*, from the German verb *schmelzen*, i.e. to fuse.
- K for *Konstanzfunktion* which translates to *constancy function.*
- I for *Identitätsfunktion* which translates to *identity function.*

But why stop here with the combinators? We can consider new combinators in order to express other important notions around functions.

Remark B.1. *(More and more combinators)* Depending on the term rewriting system we want to study, we can consider other combinators such as $C, B, D, Y, U, ...$ For example, the combinator B was defined to be an abbreviation of $S(KS)K$. This particular combinator is of great interest. In fact, Untyped Combinatory Calculus was created by Shönfinkel (and later further developed by American mathematician Haskell Curry) to be a theory about functions. Hence, it was also necessary to be able to define function composition using combinators. And, in fact, the combinator B, together with its axiom $Bxyz = x(yz)$ and conversion $Bxyz \rightsquigarrow x(yz)$, is related to function composition. The application Bxy corresponds to the usual composition $x \circ y$. For the more familiar readers, it can be noted that is possible to express any partial recursive function on the natural numbers in the term rewriting system of Untyped Combinatory Calculus. ⊠

A big difference between the Star Combinatory Calculus and the Untyped Combinatory Calculus is that the latter allows for *self-application.* For example, in Untyped Combinatory Calculus, the application ΣIIx is always valid and ΣIIx reduces to xx by means of

$$\Sigma IIx \overset{\Sigma IIx}{\succ_1} Ix(Ix) \overset{Ix}{\succ_1} Ixx \overset{Ix}{\succ_1} xx.$$

If I were to be a combinator of the Star Combinatory Calculus, it would not be possible to perform an application such as ΣII. If we type the combinator I, the corresponding conversion is given by $I_\sigma x^\sigma \rightsquigarrow x^\sigma$, for some type σ. But then the reduction $\Sigma IIx \succcurlyeq xx$ is not admissible anymore, namely because xx would need to be typed as $x^{\sigma \to \tau} x^\sigma$ (for some type τ), which is not possible.

Finally, it is worth mentioning that the Untyped Combinatory Calcululs is, just like the Star Combinatory Calculus, confluent. Moreover, it also has the Unique Normal Form Property and Combinatory Completeness. However, contrary to the Star Combinatory Calculus, it is not strongly normalising. For example, if we try to reduce the term $(\Sigma II)(\Sigma II)$ we get the following *infinite* reduction sequence

$$(\Sigma II)(\Sigma II) \overset{(\Sigma II)(\Sigma II)}{\succ_1} I(\Sigma II)(I(\Sigma II)) \overset{I(\Sigma II)}{\succ_1} (\Sigma II)(\Sigma II) \overset{(\Sigma II)(\Sigma II)}{\succ_1} ...$$

For further developments on the Untyped Combinatory Calculus, including the proofs of the above results, refer to [6], [29] and [9].

B.2 Abstract reduction systems

As mentioned in Section 1.2.5, the notions we studied for the Star Combinatory Calculus are also valid for other term rewriting systems. Term rewriting systems allow us to be more abstract about results such as the Strong Normalization Property. But it is possible *to do better*. All the definitions and results that were given in Section 1.2.5 for the Star Combinatory Calculus can be stated in an even more abstract context by introducing the notion of *abstract reduction systems*, in short *ARS*. Abstract reduction systems allow us to state results independent of the term structure of the term rewriting system. In fact, the Star Combinatory Calculus and the Untyped Combinatory Calculus can both be discussed as abstract reduction systems. As it would exceed the scope of this dissertation, we will not further develop this notion. However, as the notions introduced in Section 1.2.5 are also valid for abstract reduction systems it is interesting to mention that for any abstract reduction system $\mathscr{A}$ we have, in particular, the following results that were studied in Section 1.2.5:

1. If $\mathscr{A}$ is confluent, then it has the Unique Normal Form Property.
2. *Newman's lemma.* If $\mathscr{A}$ is locally confluent and strongly normalizing, then it is confluent.

The curious reader may refer to *Term rewriting systems* [28] for the more formal definition of abstract reduction systems, as well as various examples and results.

C Proof of Proposition 2.2

To prove Proposition 2.2, we will rely on the proof of Proposition 2.1 and extend it.

Proof of Proposition 2.2. We first rewrite the interpretations that will be needed for the proof. Let A and B be formulas such that

$$A^{SH} \equiv \forall a \exists b\ A_{SH}(a, b), \tag{C.1}$$

$$B^{SH} \equiv \forall c \exists d\ B_{SH}(c, d). \tag{C.2}$$

Applying Definition 2.1.c) to A and B gives

$$(\neg A)^{SH} \equiv \forall f \exists a'\ [\exists a \in a'\ \neg A_{SH}(a, fa)], \tag{C.3}$$

$$(\neg B)^{SH} \equiv \forall g \exists c'\ [\exists c \in c'\ \neg B_{SH}(c, gc)]. \tag{C.4}$$

By applying 2.1.d) to (C.3) gives

$$(\forall x\ \neg A(x))^{SH} \equiv \forall x, f \exists a'\ [\exists a \in a'\ \neg A_{SH}(x, a, fa)]. \tag{C.5}$$

For later purposes notice that

$$(\neg A)_{SH}(f, a') \equiv \exists a \in a'\ \neg A_{SH}(a, fa), \tag{C.6}$$

$$(\neg A(x))_{SH}(f, a') \equiv \exists a \in a'\ \neg A_{SH}(x, a, fa), \tag{C.7}$$

$$(\forall x\ \neg A(x))_{SH}(x, f, a') \equiv \exists a \in a'\ \neg A_{SH}(x, a, fa), \tag{C.8}$$

$$(\neg B)_{SH}(g, c') \equiv \exists c \in c'\ \neg B_{SH}(c, gc). \tag{C.9}$$

Interpretation of the implication. Since $A \to B \equiv \neg A \vee B$, applying Definition 2.1.b) to $\neg A$ and B using (C.3) and (C.2), we obtain

$$\begin{aligned}(A \to B)^{SH} &\equiv (\neg A \vee B)^{SH} \\ &\equiv \forall f, c \exists a', d\ [\exists a \in a' \neg A_{SH}(a, fa) \vee B_{SH}(c, d)] \\ &\equiv \forall f, c \exists a', d\ [(\neg A)_{SH}(f, a') \vee B_{SH}(c, d)] \\ &\equiv \forall f, c \exists a', d\ [\neg(\neg A)_{SH}(f, a') \to B_{SH}(c, d)]. \end{aligned} \tag{C.10}$$

Interpretation of the conjunction. Since $A \wedge B \equiv \neg(\neg A \vee \neg B)$, we apply Definition 2.1.b) to (C.3) and (C.4) to have

$$(\neg A \vee \neg B)^{SH} \equiv \forall f, g \exists a', c'\ [\exists a \in a'\ \neg A_{SH}(a, fa) \vee \exists c \in c'\ \neg B_{SH}(c, gc)]. \tag{C.11}$$

Then, by Definition 2.1.c),

$$\begin{aligned}(A \wedge B)^{SH} &\equiv (\neg(\neg A \vee \neg B))^{SH} \\ &\equiv \forall \Phi, \Psi \exists f', g'\ [\exists f \in f' \exists g \in g'\ \neg \big(\exists a \in \Phi fg\ \neg A_{SH}(a, fa) \vee \exists c \in \Psi fg\ \neg B_{SH}(c, gc)\big)] \\ &\equiv \forall \Phi, \Psi \exists f', g'\ [\exists f \in f' \exists g \in g'\ \neg \big((\neg A)_{SH}(f, \Phi fg) \vee (\neg B)_{SH}(g, \Psi fg)\big)] \\ &\equiv \forall \Phi, \Psi \exists f', g'\ [\exists f \in f' \exists g \in g'\ \big(\neg(\neg A)_{SH}(f, \Phi fg) \wedge \neg(\neg B)_{SH}(g, \Psi fg)\big)]. \end{aligned} \tag{C.12}$$

Interpretation of the unbounded existential quantification. Since $\exists x A(x) \equiv \neg(\forall x \neg A(x))$, we apply Definition 2.1.c) to (C.5) and we obtain

$$\begin{aligned}(\exists x A(x))^{SH} &\equiv (\neg(\forall x\ \neg A(x)))^{SH} \\ &\equiv \forall \Phi \exists x', f'\ [\exists x \in x' \exists f \in f'\ \neg \big(\exists a \in \Phi xf\ \neg A_{SH}(x, a, fa)\big)] \\ &\equiv \forall \Phi \exists x', f'\ [\exists x \in x' \exists f \in f'\ \neg(\neg A(x))_{SH}(f, \Phi xf)], \end{aligned} \tag{C.13}$$

where the last step can be seen by comparing (C.8) with (C.7) as follows

$$\begin{aligned}\exists a \in \Phi xf\ \neg A_{SH}(x, a, fa) &\equiv (\forall x \neg A(x))_{SH}(x, f, \Phi xf) \\ &\equiv (\neg A(x))_{SH}(f, \Phi xf). \end{aligned} \tag{C.14}$$

Interpretation of the bounded existential quantification. Similarly to the interpretation of $\exists x A(x)$, since $\exists x \in t\ A(x) \equiv \neg(\forall x \in t\ \neg A(x))$, we first apply Definition 2.1.d) to (C.3)

$$(\forall x \in t\ \neg A(x))^{SH} \equiv \forall f \exists a'\ [\forall x \in t\ \exists a \in a'\ \neg A_{SH}(x, a, fa)] \tag{C.15}$$

and then, by applying Definition 2.1.c) to (C.15), we obtain

$$\begin{aligned}
(\exists x \in t\ A(x))^{SH} &\equiv (\neg(\forall x \in t\ \neg A(x)))^{SH} \\
&\equiv \forall \Phi \exists f'\ [\exists f \in f'\ \neg \big(\forall x \in t\ \exists a \in \Phi f\ \neg A_{SH}(x, a, fa)\big)] \\
&\equiv \forall \Phi \exists f'\ [\exists f \in f'\ \exists x \in t\ \neg \big(\exists a \in \Phi f\ \neg A_{SH}(x, a, fa)\big)] \\
&\equiv \forall \Phi \exists f'\ [\exists f \in f'\ \exists x \in t\ \neg(\neg A(x))_{SH}(f, \Phi f)],
\end{aligned} \tag{C.16}$$

where the last step holds again by comparison with (C.7). This concludes the proof. $\square$

D Future work on Lean

In this appendix our goal is not to explain every definition that is involved in the upcoming results, but rather to give a general overview on our future Lean related work.

The next natural step in the formalization process is the Soundness Theorem (Theorem 3.1) that can be formalized as follows:

```
1 -- Soundness Theorem
2 theorem SoundnessTheorem
3 (A B : Formula) (z : String) (x y f a' : List String)
4 (hG : Γ₁ = insert (bAC x y f c d B) Γ) (h : Γ₁ ⊢ A):
5   ∃a b A_SH, SH_int A (a,b,A_SH) ∧
6   ∃(t:List Term), (Γ ⊢(∀₁a(A_SH.subst (b ⟹ (t ⊙ (a.tt))))))
```

In line 3 we added the variables and the arguments that are needed throughout the proof. Line 4 represents the mathematical assumption that A is provable from PL^ω_*+bAC^ω_* as we now explain. Just as it was the case with the formalization of the Combinatorial Completeness Theorem, PL^ω_* is represented in Lean by the set of formulas Γ. To add the Bounded Axiom of Choice to PL^ω_* we created hypothesis `hG`. In this hypothesis, we use `insert` to add the formula of the Bounded Axiom of Choice, given by `bAC x y f c d B`, to the set of formulas Γ. This way we are able to define a new set of formulas Γ_1 corresponding precisely to PL^ω_*+bAC^ω_*. Assumption `h` then corresponds to the mathematical assumption PL^ω_*+$\mathsf{bAC}^\omega_* \vdash A$.

Now for the conclusion of the theorem (lines 5 and 6). Since for every case in the proof of the Soundness Theorem, we always fix an interpretation, we had to write the conclusion of the theorem as the conjunction in lines 5 and 6. Line 5 states that there exists an interpretation for A. This is done by saying that there exists a, b and A_{SH} such that `SH_int A (a,b,A_SH)`. Line 6 corresponds to the more mathematical conclusion of the theorem. It states that there exists a (possibly empty) list of terms t such that the formula $\forall a A_{SH}(a, ta)$ is provable from Γ. Once again, we represent this by substituting the existential variable `b` by the term tuple application `t ⊙ (a.tt)`. In Lean this is then written as `A_SH.subst (b ⟹ (t ⊙ (a.tt)))`, which explains line 6 in the conclusion.

While working on the Soundness Theorem, we noticed that we needed (at least) the following three results. First, in order to be able to use a specific interpretation for each case of the Soundness Theorem (that is, to prove the left-hand side of the conjunction in line 5), we had to add an auxiliary lemma stating that every formula has an interpretation. Although this is clear from a mathematical point of view, it still needs to be formalized in Lean. We therefore added the following auxiliary lemma:

```
-- Auxiliary lemma
lemma all_formulas_have_an_interpretation {f a' : List String}:
  ∀ A, ∃ a b A_SH, SH_int A (a, b, A_SH)
```

This result states that given strings f and a', then for every formula A, there exist lists of strings a and b and a formula A_{SH} such that we can obtain the proposition `SH_int A (a, b, A_SH)`.

Second, we needed to extend the Combinatorial Completeness Theorem so that it could be used for tuples. This is a crucial point since the terms that are constructed throughout the proof of the Soundness Theorem all involve tuples and all rely on the lambda notation. The Combinatorial Completeness Theorem for tuples can be generalized as follows:

```
-- Combinatorial Completeness Theorem for tuples
theorem CombinatorialCompleteness_tuples (x: List String) (s: List Term):
  ∀(t:List Term), (Γ ⊢ (((λ₁ x t) ⊙ s) =_t (t.subst (x ⟹ s))))
```

For this formalization we defined λ_1 to be the lambda notation for tuples and `=_t` to be the equality between tuples. The way λ_1 was defined does not require the explicit use of `to_term` anymore because the definition of λ_1 already uses it implicitly. The other symbols were already explained previously and the result is a straightforward generalization of the Combinatorial Completeness Theorem for individual terms and variables.

Third, just like in the mathematical discussion of the Soundness Theorem, we had to formalize the Monotonicity Lemma (Lemma 3.1). We did so as follows:

```
-- Monotonicity Lemma
lemma MonotonicityLemma
(A : Formula) (hAbase : isBase A)
(intA : SH_int A (a,b,A_SH)) (b b' : List String):
  Γ ⊢ (((b ⊆₁ b') ∧₁ A_SH) →₁ (A_SH.subst (b ⟹ b'.tt)))
```

For this lemma we assume to have a formula A that is a base formula (hypothesis `hAbase`). We then assume to have an interpretation for A (hypothesis `intA`) and two lists of strings b and b'. The conclusion of the theorem then states that we want to show that the formula $b \subseteq b' \wedge A_{SH}(a,b) \to A_{SH}(a,b')$ is provable from Γ, that is, from PL^{ω}_{*}. Once again, in order to represent $A_{SH}(a,b')$, we used the substitution operator applied to A_{SH}. We did so by writing `A_SH.subst (b ⟹ b'.tt)`. As can be seen, the only new aspect of the statement of the Monotonicity Lemma is the inclusion symbol $\subseteq_1$ which is defined analogously to the mathematical symbol used in the original result.

In general, it seems that these will not be the only results required to prove the Soundness Theorem in Lean. From what has been done up to now, it seems that it will become necessary to add a decent amount of small auxiliary lemmas. Although these do not seem to be of great complexity, they still represent an important extra workload when compared to the traditional mathematical proof of the Soundness Theorem.

Index