Developing proof theory for proof exchange

Matteo Manighetti

INSTITUT POLYTECHNIQUE DE PARIS

NNT : 2023IPPAX003

Thèse de doctorat

ÉCOLE POLYTECHNIQUE

IP PARIS

Developing proof theory for proof exchange

Thèse de doctorat de l'Institut Polytechnique de Paris
préparée à École Polytechnique

École doctorale n°626 École doctorale de l'Institut Polytechnique de Paris (EDIPP)
Spécialité de doctorat : Informatique

Thèse présentée et soutenue à Palaiseau, le 9 février 2023, par

Matteo Manighetti

Composition du Jury :

Ekaterina Komendantskaya Professeure, Heriot-Watt University	Rapporteur
Didier Galmiche Professeur, Université de Lorraine	Rapporteur
Gilles Dowek Directeur de recherche, Inria & ENS Paris-Saclay	Président
Chantal Keller Maîtresse de Conférences, Université Paris-Saclay	Examinatrice
Dale Miller Directeur de Recherche, Inria & Ecole Polytechnique	Directeur de thèse

Contents

Aknowledgements

Working under the direction of Dale Miller was for me a privilege. This sentence won't surprise anyone, and indeed it is something that I knew right from the start of my PhD. Indeed, it can be stated based on his scientific merits alone. What I discovered during these years, however, was that what made it a privileged experience were different things. His kindness and his steady, non-judgmental support were fundamental for me. His positive attitude and availability for us students made our discussions some of the most enjoyable. One of my predecessors in being advised by him said: *the nicest person you will ever meet*. As simple as that, I couldn't agree more.

Thanks to Ekaterina Komendantskaya and Didier Galmiche, who agreed to review the manuscript and be part of the jury, and thanks to Gilles Dowek and Chantal Keller for agreeing to be part of it.

The, at the time, *Parsifal* team was full of marvelous people that enriched these years. In particular, Roberto, Ulysse, and Sonia welcomed me among the PhD students and showed me the path, its joys and perils. Maico and Marianela took on the adventure with me, and we shared unforgettable moments. Matteo and Marianna, Marco showed me that there is life after the PhD defense And then, thanks go to all the permanent members of the now Partout team, Kaustuv, Gabriel, Lutz and Beniamino.

Thanks to the people who made Bologna such a nice place to stay. Claudio who offered me a great opportunity. Riccardo, Paolo, Andrea, Melissa, Cecilia, Gabriele, Davide and Francesco who kept such a nice ambiance in the recondite office of Via Malaguti. Finally, thanks to Andrea, who as customary transcends all categories by being a friend and a colleague, be it in Paris, in Bologna, or in one of the countless cities we shared memories in.

My family has been throughout all of my studies a solid foundation, a calm and reassuring environment where I could always go back and find some peace of mind. During these last years, as difficulties arose, the solidness just got stronger. I will forever be grateful to my parents for supporting me in my adventures, and to my sister for always being there for me.

Finally, to the one who had a PhD thesis dedicated to her at birth, and therefore knew at times what was going on better than me. Thanks to Gwenaëlle for being there and giving me all of her strength, and for all of those little moments where this made the difference.

Ponme aceite y sal
que aún hay mucho pan
y esto va pa' largo

Introduction (en Français)

Cette thèse concerne le développement d'outils fondamentaux basés sur la théorie de la démonstration structurelle pour traiter les langages formels pour la représentation de la démonstration. L'objectif est d'améliorer la communication de documents mathématiques formels d'origines différentes, produits soit par des humains avec l'aide de logiciels, soit produits uniquement par des logiciels. Par fondamentaux, nous entendons que nous cherchons à faire le moins d'engagement possible envers une certaine vision philosophique dans les concepts que nous introduisons, ce qui nous permettra de capturer plus de formes de raisonnement dans notre cadre. Nous élucidons certaines de nos opinions sur le sujet et présentons brièvement les résultats présentés dans cette thèse.

Le tournant linguistique. La logique en tant que champ d'étude se situe à la croisée des chemins de la mathématique, de la philosophie et de l'informatique. Elle concerne, à sa base, l'analyse des éléments fondamentaux qui constituent les éléments de discours dans les trois disciplines. Une telle attention aux éléments de base de leur discours est particulièrement importante lorsqu'on considère la tendance notoire à l'obscurantisme de nombreux praticiens de ces domaines.

Ce que nous entendons par logique de nos jours est cependant le résultat de certains changements dramatiques survenus au cours du XIXe siècle. Tout d'abord, la mathématisation des principes logiques par Boole a posé les bases pour la logique mathématique, où pour la première fois l'objet d'étude de la logique avait une présentation de leur propre droit ainsi que des méthodes mathématiques puissantes pour raisonner à leur sujet. Ensuite, le développement du langage plus riche de la logique propositionnelle par Frege [Fre80] est à l'origine de ce qui est devenu connu sous le nom de "tournant linguistique" dans l'histoire de la philosophie. Frege est parvenu à la conclusion que pour discuter de la pensée, il fallait discuter du sens des propositions. Cela est dû au fait qu'il n'y a peu de sens à analyser le sens des mots et des concepts en isolation, et c'est seulement dans le contexte des propositions que la question a du sens. Le rôle de la logique est donc central, car la logique est l'étude des propositions et de leurs significations. Ces vues ont révolutionné la philosophie occidentale au XXe siècle, et ont causé ce qui ressemble presque à un schisme entre l'école analytique, se conformant à cette attention à la logique et au langage, et la tradition continentale.

Logicisme et formalisme L'objectif principal de Frege était d'étudier le sens des entités mathématiques, dans le but de donner une explication purement

fondée sur la logique pour l'ensemble des mathématiques. Dans ce processus, Frege a entamé de féroces débats avec Russell et Hilbert. Ce dernier, cependant, a discuté de la nature purement logique des mathématiques et a proposé en alternative la "méthode finie", qui suppose l'existence d'entités extralogiques primitives que nous pouvons représenter dans la pensée; une collection d'opérations sûres et finies est alors autorisée sur ces entités. Le célèbre et malheureux programme de Hilbert était de sécuriser la consistance de toutes les mathématiques sur ces méthodes.

Ce qui a déterminé le sort malheureux du programme de Hilbert est la découverte des théorèmes d'incomplétude de Gödel, qui ont déclaré l' de prouver la consistance de l'arithmétique par des méthodes finies. Bien qu'il puisse être discuté en détail si cela a réellement été la fin pour le programme de Hilbert, il est certain que cela a convaincu beaucoup que la sécurisation des fondements des mathématiques n'était pas possible, et donc que l'activité fastidieuse de mathématiques formelles que Frege et Russell avaient initiée n'était finalement pas si importante.

Il sera peut-être suffisant de citer le travail de Bourbaki, une tentative de construire les mathématiques à partir de zéro. Malgré la renommée de ce projet, il semble en lisant leur travail que Bourbaki a ignoré la plupart de ces développements en logique, y compris le théorème d'incomplétude de Gödel lui-même. Le membre de Bourbaki, Dieudonné, a si peu pensé à la logique mathématique que, notamment en 1980, il a déclaré : "Si tout ce que les logiciens ont fait après 1925 disparaissait, on ne remarquerait même pas". Même sans recourir à de telles positions extrêmes, après l'échec du programme de Hilbert, l'intérêt des mathématiciens pour les mathématiques formelles et la métamathématique est resté minimal.

Formalisation et formalisme Lorsque le débat sur le formalisme en mathématiques a ralenti, la question a été repoussée par l'informatique. La formalisation mathématique aidée par ordinateur a commencé à devenir une possibilité concrète, et en 1976, le premier grand-échelle expérience (bien que menée par une seule personne) a été effectuée avec la formalisation d'un livre sur l'analyse mathématique dans le système Automath. Quelques années auparavant, Hoare avait proposé une fondation logique pour la vérification de programmes, conduisant à l'idée que les programmes informatiques pouvaient être vérifiés pour la correction par des machines.

Ce n'est qu'à la fin des années 80 que plusieurs systèmes commencent à apparaître et que des processus plus importants peuvent être réalisés en collaboration. Vers ce moment, le manifeste influent QED [Boy94] présente un plan pour le développement d'un grand corps de mathématiques formellement vérifiés. Dans celui-ci, l'auteur aborde dans une série de réponses aux objections possibles, la première étant: quelle fondation logique devrions-nous choisir? L'auteur fournit deux réponses, la première assez déflationnaire ("*on peut souvent transférer presque toutes les techniques développées dans une logique à une logique ultérieure, mieux*") et la seconde beaucoup plus subjective:

> Ce sont des controverses en Philosophie des Mathématiques. Qui s'en soucie? La très grande majorité des mathématiciens contemporains croient qu'il n'y a pas de doutes quant à ce que signifie qu'une preuve soit correcte.

Ce qui manque dans cette réponse, c'est le fait que le choix des fondements pour un système informatique détermine non seulement l'expressivité (mathématique) de l'outil, mais aussi la structure de ses arguments et leur présentation. En rejetant la question, un point crucial pour la communication formelle de la preuve est également rejeté. Alors que les mathématiciens travaillant souvent ignorent la question des fondements car ils peuvent lire des travaux indépendamment des fondements dans lesquels ils tombent, ce n'est pas le cas pour les mathématiques informatisées : ici, le fondement choisi est directement reflété dans le formalisme, et pour savoir qu'une preuve est correcte, il est nécessaire d'exécuter le bon morceau de logiciel. À titre d'exemple, le prouveur de théorème Lean est en train de publier sa version 4, qui ne sera pas compatible avec la plupart des développements réalisés dans ce même système jusqu'à présent.

Dans une section ultérieure du manifeste, il y a une discussion plus modérée des fondements logiques, et comporte des observations que nous sommes très d'accord : parmi celles-ci, l'auteur identifie le piège de la recherche d'une seule logique idéale qui devrait sous-tendre tout développement et pousse l'idée que de nombreuses fondations devraient coexister, en soulignant comment il est important que ces fondations différentes puissent être facilement implémentées et que l'une puisse servir de base à l'autre.

Cependant, il semble que au fil du temps, ces observations aient été lues pour le pire, et à la lumière de la négligence philosophique suggérée par la citation. Dans une évaluation de 20 ans de QED [HUW16], nous lisons que "la balkanisation a encore empiré". Une communication fructueuse entre différents assistants à la preuve semble être loin.

Certificats de Preuve Fondamentaux Le programme de recherche des Certificats de Preuve Fondamentaux (FPC en abrégé) propose une réponse aux problèmes de communication de preuves formelles en se concentrant sur l'idée que le démontreur devrait être capable d'expliquer son langage de preuve dans un langage minimal. À son tour, ce langage minimal est celui où le sens de la preuve est connu pour être clair, mais aucun choix particulier n'est fait sur ce qu'il devrait être. Le dispositif technique permettant cela est celui du Calcul des Séquents, une représentation de preuves introduite par Gentzen dans le contexte de l'analyse finitiste de Hilbert. Nous avons un esprit similaire en fondant le sens de l'inférence sur le sens atomique des règles du calcul; en plus de cela, des résultats récents sur ce qu'on appelle le calcul des séquents *axés* permettent une façon claire de composer ces opérations atomiques en opérations complexes, de telle sorte que le fonctionnement de ces plus grandes règles peut correspondre extérieurement à celui d'un démonstrateur de théorèmes, tandis que son sens est toujours fourni par le système fondamental.

Le credo est que ces petits fondements robustes fournissent à la fois les outils théoriques et techniques pour construire des explications formelles de langages logiques plus riches.

Contenu de la thèse Le cadre des Certificats de Preuve Fondamentaux a été présenté, à son introduction, comme capable d'exprimer des preuves formelles présentées dans une variété de systèmes formels [Chi15]. Subséquemment, il a été montré comment il pourrait gérer les preuves provenant de démon-

strateurs de théorèmes automatisés [Bla17]. Dans cette thèse, nous proposons deux nouvelles avancées, ainsi que des résultats théoriques visant à augmenter l'expressivité du cadre:

- Dans le chapter 1, nous fournissons un aperçu sur la théorie des preuves et la certification de preuves. Nous introduisons les systèmes de preuves axés et définissons formellement le cadre des Certificats de preuve fondamentaux.
- Dans le chapter 2, nous considérons deux transformations courantes qui sont utilisées comme étapes de prétraitement dans la démonstration automatique de théorèmes, la Skolemisation et la transformation de Tseitin. Nous montrons comment elles peuvent être expliquées comme des opérations sur le calcul de séquents axé, obtenant ainsi une manière de les capturer à l'intérieur des Certificats de preuve fondamentaux.
- Dans le chapter 3, nous considérons des systèmes de logique linéaire et classique étendus avec un opérateur de point fixe, et nous obtenons certaines propriétés pour eux qui seront utiles pour la description de formats de certificats pour l'arithmétique.
- Dans le chapter 4, nous présentons deux implémentations prototypes de composants basés sur les Certificats de preuve fondamentaux dans l'assistant de preuve Coq. Le premier prototype permet d'importer des preuves externes dans Coq, et le second de générer des contre-exemples pour des conjectures.

Publications La chapter 2 est basée sur [CMM19], la chapter 3 est basée sur les brouillons [MM22a; MM22b] et la chapter 4 est basée sur [MMM20] et les soumissions à des ateliers [BMM20a; BMM20b].

Code Le code discuté dans la thèse se trouve dans plusieurs dépôts

- Le code général pour FPC mentionné dans la chapter 1 peut être trouvé à `https://github.com/manmatteo/thesis-code`
- Le développement principal est constitué de deux tactiques intégrant un contrôleur pour FPC dans un Coq. Le code pour eux ainsi que quelques exemples peuvent être trouvés à `https://github.com/proofcert/fpc-elpi`
- Le code pour la procédure de déskolémisation décrite dans la chapter 2 se trouve à `https://github.com/chaudhuri/proofcert-deskolemize`

Introduction

This thesis concerns the development of foundational tools based on structural proof theory to treat formal languages for proof representation. The aim is to improve the communication of formal mathematical documents of different origins, either produced by humans through the assistance of software or produced by software alone. By foundational we mean that we strive to make as minimal a commitment as possible to a certain philosophical view in the concepts we introduce; this will allow us to capture more flavours of reasoning in our framework. We elucidate some of our views on the subject and briefly introduce the results presented in this thesis.

The linguistic turn. Logic as a field of study lies at the crossroads of mathematics, philosophy, and computer science. It concerns, at its core, the analysis of the fundamental elements that constitute the bits and pieces of the discourse in the three disciplines. Such an attention to the basic elements of their discourse is particularly important when one considers the notorious tendency to obscurantism of many of the practitioners in these fields.

What we mean by logic nowadays, however, is the result of some dramatic changes occurred over the course of the 19th century. First, the mathematization of logical principles by Boole laid the ground for *mathematical logic*, where for the first time the object of study of logic had a presentation of their own right as well as powerful mathematical methods to reason about them. Then, the development of the richer language of predicate logic by Frege [Fre80] is at the root of what became known as the *linguistic turn* in the history of philosophy. Frege came to the conclusion that in order to discuss thought, one had to discuss the meaning of propositions. This is because there is little point in analyzing the meaning of words and concepts in isolation, and it is only in the context of propositions that the question makes sense. The role of logic is therefore a central one, since logic is the study of propositions and their meanings. These views revolutionized western philosophy in the 20th century, and caused what has almost been a schism between the analytic school, abiding to this attention to logic and language, and the continental tradition.

Logicism and formalism The main aim of Frege, however, was that of studying the meaning of the mathematical entities, with the aim of giving for the entire of mathematics an explanation that was purely grounded on logic. In this process, Frege embarked in fierce debates with Russell and Hilbert. The former, after discovering the well-known paradox in Frege's system, worked on a logicist system of his own. The latter, on the other side, disagreed on the purely logical nature of mathematics and proposed as an alternative the *fini-*

tary method, which presupposes the existence of primitive, extralogical entities that we can represent in thought; then, a collection of secure, finitary operation is allowed on these entities. Hilbert's celebrated and ill-fated program was to secure the consistency of all mathematics based on these methods.

What determined the ill fate of Hilbert's program is the discovery of the incompleteness theorems by Gödel, which stated the impossibility of proving the consistency of arithmetic by finitary methods. Although it can be discussed at length whether this was really the end for Hilbert's program, what is certain is that it convinced many that there was no hope to secure the foundations of mathematics, and thus the tedious activity of formal mathematics that Frege and Russell had pioneered was in the end not so important.

Perhaps it will be sufficient to cite the work of Bourbaki, an attempt to build mathematics from the ground up. Despite the fame of this project, it seems from reading their work that Bourbaki ignored most of these developments in logic, including Gödel's incompleteness theorem itself. Bourbaki member Dieudonné thought so little of mathematical logic that he notably said in 1980 "Were everything logicians did after 1925 to disappear, one wouldn't even notice". Even without resorting to such extreme positions, after the failure of Hilbert's program the interest of mathematicians in formal mathematics and metamathematics remained minimal.

Formalization and formalism As the debate over formalism in mathematics slowed down, the matter was to be picked up by computer science. Computer-aided mathematical formalization started becoming a concrete possibility, and in 1976 the first large-scale experiment (although done by a single person) was carried out with the formalization of a book on mathematical analysis within the Automath system. A few years before, Hoare had proposed a logical foundation for program verification, leading to the idea that computer programs could be checked for correctness by machines.

It is not until the end of the 80s that multiple systems start to appear, and larger, collaborative processes appear to be possible. Around this time, the influential QED Manifesto [Boy94] presents a plan for the development of a large body of formally verified mathematics. In it, the author embraces in a series of answers to possible objections, the first of which is: which logical foundations should we choose? The author provides two answers, the first quite deflationary ("*one can often transfer almost all of the technique developed in one logic to a subsequent, better logic*") and the second much more opinionated:

> These are controversies in the Philosophy of Mathematics. Who cares? The overwhelming majority of contemporary mathematicians believe that there are no doubts about what it means for a proof to be correct.

What is missing in this answer is the fact that the choice of foundations for a software system determines not only the (mathematical) expressiveness of the tool, but also the structure of its arguments and their presentation. By dismissing the question, a crucial point for formal proof communication is also dismissed. Whereas working mathematicians often ignore the matter of foundations since they can read works independently of the foundations they happen into, this is not the case for computer-formalized mathematics: here which

foundation one choses is directly reflected on the formalism, and in order to know that a proof is correct one needs to execute the correct piece of software. Just as an example, the Lean theorem prover is now in the process of releasing its version 4, which will not be compatible with most developments carried out in that same system so far.

In a latter section the manifesto contains a milder discussion of logical foundations, and contains observations that we very much agree with: among these, the author identifies the trap of looking for a single, ideal logic that should underlay any development, and pushes the idea that many foundations should coexist, stressing the importance that these different foundation can be easily implemented and are such that one can form the basis for the other.

However, it seems that over the course of time these observations have been read for the worse, and in the light of the philosophical neglection suggested by the quote. In a 20-year evaluation of QED [HUW16], we read that "Balkanization got even worse". Fruitful communication between different proof assistants seems to be far away.

Foundational Proof Certificates The research program of Foundational Proof Certificates (FPC for short) proposes an answer to the issues in the communication of formal proofs that is centered around the idea that the prover should be able to explain their proof language in a minimal language. In turn, this minimal language is one where the meaning of proof is known to be clear, but no particular choices are made on what it should be. The technical device allowing this is that of the Sequent Calculus, a representation of proofs introduced by Gentzen in the context of Hilbert's finitist analysis. We have a similar spirit in grounding the meaning of inference on the atomic meaning of the rules of the calculus; in addition to this, recent results on what is known as *focused* sequent calculus allow a clear way to compose these atomic operations into complex ones, such that the operation of these bigger rules can externally match that of a theorem prover, while its meaning is still provided by the foundational system.

The tenet is that these small, robust foundations provide the meaning-theoretical and technical tools to build formal explanations of richer logical languages.

Contents of the thesis The framework of Foundational Proof Certificates has been presented, at its introduction, as capable of expressing proof evidence presented in a variety of formal systems [Chi15]. Subsequently, it has been shown how it could handle proof evidence coming from automated theorem provers [Bla17]. In this thesis we propose two new advances, as well as some theoretical results aimed at increasing the expressivity of the framework:

- In chapter 1 we provide some background on proof theory and proof certification. We introduce focused proof systems, and we define formally the framework of Foundational Proof Certificates.

- In chapter 2 we consider two common transformations that are used as preprocessing steps in automated theorem proving, Skolemization and the Tseitin transformation. We show how they can be explained in terms

of operations on the focused sequent calculus, thus obtaining a way to capture them inside Foundational Proof Certificates.

- In chapter 3 we consider systems of linear and classical logic extended with a fixed-point operator, and we obtain some properties for them that will be useful for the description of certificate formats for arithmetic.
- In chapter 4 we present two prototype implementations of components based on Foundational Proof Certificates into the Coq proof assistant. The first prototype allows importing external proof evidence in Coq, and the second to generate counterexamples for conjectures.

Publications Chapter 2 is based on [CMM19], chapter 3 is based on the drafts [MM22a; MM22b] chapter 4 is based on [MMM20] and the workshop submissions [BMM20a; BMM20b].

Code The code discussed in the thesis is located in several repositories

- The general code for FPC that is mentioned in chapter 1 can be found at `https://github.com/manmatteo/thesis-code`
- The main development are two tactics embedding a checker for FPC into a Coq. The code for them along with some examples can be found at `https://github.com/proofcert/fpc-elpi`
- The code for the deskolemization procedure described in chapter 2 is at `https://github.com/chaudhuri/proofcert-deskolemize`

Chapter 1

Background material

The developments of this thesis are carried out in the setting of structural proof theory. Our reference framework is the research on Foundational Proof Certificates, as proposed in [CMR17]. Structural proof theory is centered on the study of tree-like presentations of proofs and their properties, and Foundational Proof Certificate propose a refinement of these structure as a protocol for the communication between simple checkers building Gentzen-style proofs and formal proofs in a range of different proof structures.

This chapter presents the background technical material on logic and proof certification, outlines our view on these matters and introduces the technologies used for the implementations of tools and prototypes related to this work.

1.1 Structural proof theory

1.1.1 Syntax of formulas

We will use the λ-tree approach to syntax [MP99], where we use Church's Simply Typed λ calculus [Chu40] to represent the formulas. We consider two basic types ι and o, and build more complex types with the $\to$ right-associative constructor.

Definition 1. Terms and types are built according to this grammar:

$$\begin{array}{ll} \text{Types:} & \alpha, \beta := \iota \mid o \mid \alpha \to \beta \\ \text{Terms:} & t, u := x \mid \lambda x.\, t \mid t\ u \end{array}$$

The terms are related to types by means of the typing relation, denoted by the semicolon and built with the following rules:

$$\frac{\Sigma, x : \alpha \vdash t : \beta}{\Sigma \vdash \lambda x.\, t : \alpha \to \beta}\ \text{abs} \qquad \frac{\Sigma \vdash t : \alpha \to \beta \quad \Sigma \vdash u : \alpha}{\Sigma \vdash t\ u : \beta}\ \text{app} \qquad \frac{}{\Sigma, x : \alpha \vdash x : \alpha}\ \text{ax}$$

The type o is the type of formulas. We extend the system with a number of typed *logical constants*, that we will write infix:

$$\top : o \quad \bot : o$$

$$\wedge : o \to o \to o \qquad \vee : o \to o \to o \qquad \supset : o \to o \to o$$

As usual we write the connectives infix, and we use capital letters for atomic formulas. An additional unary constant $\neg : o \to o$ could be included, but we treat it instead $\neg A$ as a definition for $A \supset \bot$. Atomic formulas and negated atomic formulas will be called literals. Finally, we have the typed quantifiers:

$$\forall : (\iota \to o) \to o \quad \exists : (\iota \to o) \to o$$

Finally, we assume the existence of a countable set of constants of type o that are the propositional constants, or propositional atoms, and of a countable set of constants of types $\tau \to o$, where τ is an arbitrary type not containing o that are the predicate constants. We denote propositional atoms with $p, q, r \ldots$ and predicate constants with $P, Q, R \ldots$. A term $P\bar{t}$ of type o is a first-order atom.

The typing of the quantifiers makes it so that the binders usually associated with them are handled in the same way as term-level binders by the λ notation. For example, a universally quantified formula is written in this notation as $\forall\,(\lambda x.\, Px)$. However, since there is no ambiguity, we will use the usual concrete syntax $\forall x.\, Px$. Additionally, substitution is handled by β-reduction: given a formula $\forall x.\, A$ we define $[t/x]A$, the substitution of t for x in its body, as the β-normal form of $(\lambda x.\, P)\; t$.

1.1.2 Sequent calculus

The basic toolkit of structural proof theory builds on the works of Gentzen [Gen35], and in particular his systems of Natural Deduction and Sequent Calculus. Natural Deduction and its reading through the Curry-Howard isomorphism are usually presented as the underlying proof system for many proof assistants (such as Coq, Matita). However, we find Natural Deduction to be too opinionated as a proof formalism, and thus unsuitable as a foundational language. In particular, it lacks explicit primitives to handle the information in the context, and its handling of classical logic can feel unnatural. On the other side, Sequent Calculus provides a clear encoding of both classical and intuitionistic logic, and Natural Deduction proofs can be easily expressed in a fragment of the Sequent Calculus. The finer rules available in the Sequent Calculus make it more usable as a foundation for syntax. We shall dwell more on these arguments later, but let's now move on to the technical definitions.

The basic objects of the system are *sequents* of the form $\Sigma; \Gamma \vdash \Delta$ where both Γ and Δ are multisets of formulas, and Σ is a set of typing judgements. A sequent is meant to represent a certain point in an inference, its left-hand side is called antecedent and contains the premises, and the right-hand side is called consequent and contains the conclusions. Asserting that a sequent is valid amounts to saying that from asserting all the premises, we can correctly draw one of the conclusions. The way to show that a sequent is valid is by subsequent application of proof rules.

The rules for *LK*, the sequent calculus for first order classical logic, are shown in fig. 1.1. We call the rules in the first two groups *introduction rules*, and this is traditionally motivated by their top-down reading, where we see one connective being introduced on one side for each rule. Introduction rules are named after the side they act on and the connective they introduce, for example $\wedge$-left and so on. The rules in the *Identity* group handle the overall organization of the deduction: the init rule concludes the reasoning, while the

Right introduction rules

$$\frac{\Sigma;\Gamma\vdash\Delta,A\quad\Sigma;\Gamma\vdash\Delta,B}{\Sigma;\Gamma\vdash\Delta,A\wedge B}\qquad\frac{\Sigma;\Gamma\vdash\Delta,A,B}{\Sigma;\Gamma\vdash\Delta,A\vee B}\qquad\frac{\Sigma;\Gamma,A\vdash\Delta,B}{\Sigma;\Gamma\vdash\Delta,A\supset B}$$

$$\frac{\Sigma;\Gamma\vdash\Delta}{\Sigma;\Gamma\vdash\Delta,\bot}\qquad\frac{}{\Sigma;\Gamma\vdash\Delta,\top}\qquad\frac{\Sigma\vdash t:\iota\quad\Sigma;\Gamma\vdash\Delta,[t/x]A}{\Sigma;\Gamma\vdash\Delta,\exists x.\,A}\qquad\frac{\Sigma,y:\iota;\Gamma\vdash\Delta,[y/x]A}{\Sigma;\Gamma\vdash\Delta,\forall x.\,A}\;y\notin\Sigma$$

Left introduction rules

$$\frac{\Sigma;\Gamma,A,B\vdash\Delta}{\Sigma;\Gamma,A\wedge B\vdash\Delta}\qquad\frac{\Sigma;\Gamma,A\vdash\Delta\quad\Sigma;\Gamma,B\vdash\Delta}{\Sigma;\Gamma,A\vee B\vdash\Delta}\qquad\frac{\Sigma;\Gamma\vdash\Delta,A\quad\Sigma;\Gamma,B\vdash\Delta}{\Sigma;\Gamma,A\supset B\vdash\Delta}$$

$$\frac{\Sigma;\Gamma\vdash\Delta}{\Sigma;\Gamma,\top\vdash\Delta}\qquad\frac{}{\Sigma;\Gamma,\bot\vdash\Delta}\qquad\frac{\Sigma,y:\iota;\Gamma,[y/x]A\vdash\Delta}{\Sigma;\Gamma,\exists x.\,A\vdash\Delta}\qquad\frac{\Sigma\vdash t:\iota\quad\Gamma,[t/x]A\vdash\Delta}{\Sigma;\Gamma,\forall x.\,A\vdash\Delta}\;y\notin\Sigma$$

Identity rules

$$\frac{\Sigma;\Gamma\vdash\Delta,A\quad\Sigma;\Gamma,A\vdash\Delta}{\Sigma;\Gamma\vdash\Delta}\;\text{cut}\qquad\frac{}{\Sigma;\Gamma,A\vdash\Delta,A}\;\text{init}$$

Structural rules

$$\frac{\Sigma;\Gamma\vdash\Delta,A,A}{\Sigma;\Gamma\vdash\Delta,A}\;\text{Contraction-R}\qquad\frac{\Sigma;\Gamma\vdash\Delta,A}{\Sigma;\Gamma\vdash\Delta}\;\text{Weakening-R}$$

$$\frac{\Sigma;\Gamma,A,A\vdash\Delta}{\Sigma;\Gamma,A\vdash\Delta}\;\text{Contraction-L}\qquad\frac{\Sigma;\Gamma\vdash\Delta}{\Sigma;\Gamma,A\vdash\Delta}\;\text{Weakening-L}$$

Figure 1.1: Rules of *LK*

cut rule handles the integration of lemmas inside a proof. The rules in the final group are called *structural rules* and are used to handle the context.

What led Gentzen into investigating the *LK* calculus was the search for normal form deductions in the calculus of Natural Deduction. Instead of the introduction rules for both sides of a sequent, Natural Deduction consists of pairs of introduction and elimination rules for each connective, such as:

$$\frac{\Gamma,A\vdash B}{\Gamma\vdash A\supset B}\;\supset\text{-intro}\qquad\frac{\Gamma\vdash A\supset B\quad\Gamma\vdash A}{\Gamma\vdash B}\;\supset\text{-elim}$$

These rules can be combined in a way that presents a *detour* in the argument when an introduction rule for a connective is immediately followed by an elimination rule on that same occurrence of a connective. Gentzen's program to prove that any theorem is provable without detours went through for intuitionistic logic, but didn't work for classical logic. That's what led Gentzen to develop sequent calculi as a framework where it was possible to encode provability in natural deduction; the generalization to having multiple formulas on the right-hand side and the discovery of *LK* allowed him to study classical logic. The normalization theorem then becomes what has since been known

as Gentzen's *Hauptsatz*: a formula is provable in *LK* if and only if it is provable without the use of the Cut rule. As a consequence of this, by inspection on the rules of *LK* one gets that every formula can be proved by only appealing to subformulas, and this in turn means that no detour is present in the corresponding natural deduction proof. A sequent calculus for intuitionistic logic is obtained by restricting the consequent to only contain one formula: the cut elimination theorem goes through for this system with minimal modifications, and provides therefore another normalization theorem for intuitionistic natural deduction.

By inspecting the rules of *LK*, we quickly realize that there is some redundancy: left introduction rules for $\wedge, \vee, \forall, \exists$ can be seen as being the De Morgan dual of right introduction rules. We see for example that when treating the formula $(A \wedge B) \supset C$ on the right-hand side with the $\supset$-right, the subformula $A \wedge B$ is moved to the left-hand side, while C remains on the right-hand side. Then the conjunction is treated on the left side with a rule that resembles the rule for disjunction on the right. Conversely, if the formula $(A \wedge B) \supset C$ is in the antecedent of a sequent, the $\supset$-left rule originates a premise where C is left in the antecedent, and another premise where $A \wedge B$ has been moved to the right, where it is going to be treated by an analogue of the $\vee$-left rule. Thus, the rules for $\supset$ have the main effect of moving their left subformula to the opposite side of the sequent. Recall moreover that we defined negation of a formula F as $F \supset \bot$: when this is treated by the $\supset$ rules, the $\bot$ is immediately eliminated by the relevant $\bot$ rules, and the only effect is that of moving F to the opposite side of the sequent.

We can make this observations concrete by including an orthogonality operator $\cdot^\bot$, that embodies the idea that formulas in the left-hand side can equivalently stay on the same side provided they are treated with the rule of their dual connective. Then, we get rid of the $\supset$ logical constant and replace it with a combination of $\vee$ and $\cdot^\bot$. We keep the $\neg$ constant around to denote the orthogonal of an atom. This is also well known as the *Negation Normal Form*, since it can be seen as pushing $\neg$ inside formulas by repeatedly applying the De Morgan dualities, so that it is only applied to atoms.

Definition 2 (Negation Normal Form)**.** The Negation Normal Form of a formula is

$$nnf(A) = A \text{ if } A \text{ is atomic, } \bot \text{ or } \top \quad nnf(A \supset B) = A^\bot \vee nnf(B)$$

$$nnf(A \wedge B) = nnf(A) \wedge nnf(B) \quad nnf(A \vee B) = nnf(A) \vee nnf(B)$$

$$nnf(\forall x.\, A) = \forall x.\, nnf(A) \quad nnf(\exists x.\, A) = \exists x.\, nnf(A)$$

The operator $\cdot^\bot$ is defined as follows:

$$(A)^\bot = \neg A \quad \text{if } A \text{ is atomic}$$

$$(A \wedge B)^\bot = A^\bot \vee B^\bot \quad (A \vee B)^\bot = A^\bot \wedge B^\bot \quad (A \supset B)^\bot = nnf(A) \wedge B^\bot$$

$$\top^\bot = \bot \quad \bot^\bot = \top \quad (\forall x.\, A)^\bot = \exists x.\, A^\bot \quad (\exists x.\, A)^\bot = \forall x.\, A^\bot$$

When Γ is a multiset, $\Gamma^\bot$ and $nnf(\Gamma)$ are obtained by applying $\cdot^\bot$ or nnf to all formulas in Γ.

Logical rules

$$\frac{\Sigma \vdash \Gamma, A \quad \Sigma \vdash \Gamma, B}{\Sigma \vdash \Gamma, A \wedge B} \qquad \frac{\Sigma \vdash \Gamma, A, B}{\Sigma \vdash \Gamma, A \vee B}$$

$$\frac{\Sigma \vdash \Gamma}{\Sigma \vdash \Gamma, \bot} \qquad \frac{}{\Sigma \vdash \Gamma, \top} \qquad \frac{\Sigma \vdash t : \iota \quad \Sigma \vdash \Gamma, [t/x]A}{\Sigma \vdash \Gamma, \exists x.\, A} \qquad \frac{\Sigma, y : \iota \vdash \Gamma, [y/x]A}{\Sigma \vdash \Gamma, \forall x.\, A}\ y \notin \Sigma$$

Identity rules

$$\frac{\Sigma \vdash \Gamma, A \quad \Sigma \vdash \Gamma, A^{\perp}}{\Sigma \vdash \Gamma}\ \text{cut} \qquad \frac{}{\Sigma \vdash \Gamma, A^{\perp}, A}\ \text{init}$$

Structural rules

$$\frac{\Sigma \vdash \Gamma, A, A}{\Sigma \vdash \Gamma, A}\ \text{contraction} \qquad \frac{\Sigma \vdash \Gamma}{\Sigma \vdash \Gamma, A}\ \text{weakening}$$

Figure 1.2: Single sided version of *LK*

Figure 1.2 contains the single sided version of *LK*, where there is no antecedent. It is easy to see that a sequent $\Gamma \vdash \Delta$ is provable in the two-sided version of *LK* if and only if $\vdash \Gamma^{\perp}, nnf(\Delta)$ is provable in the single sided version.

1.1.3 Implementations in Higher Order Logic Programming

The implementation of the tools and prototypes related to the current work has been carried out using the λProlog language, and more specifically the ELPI implementation [Dun+15]. The foundation of λProlog is the theory of Hereditary Harrop Formulas in intuitionistic Higher Order logic. The main extensions with respect to traditional logic programming are

- Terms are built with λ-tree syntax and are typed with a type system analogous to the one of definition 1, where atoms are those terms of a specific type (`prop` in ELPI).
- Clauses can contain implications in their bodies with the $\Rightarrow$ operator, which are interpreted as dynamic, scoped extensions of the program.
- Similarly, clauses can contain universal quantifications in their bodies with the `pi` operator, which are interpreted as dynamic, scoped extension of the program's signature.

The mechanisms of dynamic scoped extensions provide a uniform treatment for binders, and binders in the syntax are easily treated by means of binders in the program. A complete reference for λProlog is the book [MN12], while more examples and tutorials can be found in the webpage of the ELPI project.

In listing 1.1 a type checker for the typing relation defined in definition 1 is implemented. The implementation technique is standard in Higher Order Logic Programming. The first step is defining the types `tm, ty` to represent terms and types of the object language, and by introducing the constructors

for them according to the grammar we provided. The `abs` constructor corresponds to the λ term constructor, and handles the binder: this is obtained by typing `abs` with a higher order type, that takes a `tm` abstraction and returns a `tm`. Thanks to this, we don't need to include variables with an explicit constructor.

The next code block introduces the constants for terms and types, and then we move on to the definition and implementation of the typing relation `of`. The two clauses for `app` and `abs` directly correspond to the rule of definition 1. The clause for `abs`, in particular, exploits the mechanism of scoped extensions to encode the extension to the context Σ, by introducing a new term `x` together with its typing judgement and making it available to the subsequent call to the type checking predicate.

Listing 1.1: λProlog implementation of the typechecker for definition 1

```
kind ty, tm type.
type (⟶) ty→ ty→ ty.
type abs (tm→ tm)→ tm.
type app tm→ tm→ tm.

type i,o ty.
type arr tm.
type all tm.

type of tm→ ty→ prop.

of arr (o⟶o⟶o).
of all ((i⟶o)⟶o).

of (app T U) B :-
  of T (A⟶B),
  of U A.

of (abs T) (A⟶B) :-
  pi x\ of x A⇒
  of (T x) B.
```

We can test the program in ELPI with a query like:

```
?- pi p\ of p (i⟶o)⇒
   pi q\ of q (i⟶o)⇒
     of (app all (abs x\ app (app arr (app p x)) (app q x))) o.
```

This query extends the program's signature with two constants `p` and `q` and the assertions that their types are `i` $\longrightarrow$ `o`[1], and then checks that the term $\forall\,(\lambda x.\, Px \to Qx)$ has type o. In the implemented prototypes we use a simpler encoding of formulas, where we directly use a λProlog type as the type of formulas, instead of having a type `ty` of object types, and a λProlog term `o` that is the type of propositions in the encoded system. In this way typeckeching will be performed by the type checker of ELPI.

1.2 Polarities, focusing and proof certificates

Structural proof theory is a well-studied discipline that precisely serves the task of giving an account of formal provability. However, proofs from automated theorem provers come in many formats and employ specialized optimizations and transformations. There's often a big gap between them and the languages of proof checkers and proof assistants. We present now some more recent advancements in proof theory that lead to a communication protocol

[1] This could have equivalently been done by extending the body of the program by including the declarations `type p tm. type q tm.` as well as the assertions for `of`.

where what one means by a proof is left to the proof author, who gets in exchange automated tools for checking the proof.

1.2.1 Focusing the sequent calculus

The advantages of the rules of *LK* due to their locality are easily seen. However, it is not nearly as intuitive how they can serve as a foundation for proof search and proof reconstruction tasks. There are some arguments that are usually raised against the idea that proofs in this formalism contain an adequate amount of information:

1. Neither the single nor the double-sided sequents are amenable to a clear explanation of logical inference, since there is in principle no particular formula that is the site of the inference. Rather, the entire right-hand side contains formulas that could in principle be part of the inference but won't necessarily be.

2. Contraction and weakening have no constraints, could happen anywhere and multiple times. Their role is useful for expressiveness, but they lack a meaning-theoretical interpretation.

3. Many rules can be permuted between each other. Consider for example

$$\dfrac{\dfrac{\Sigma\vdash\Gamma, A, B, C \quad \Sigma\vdash\Gamma, A, B, D}{\Sigma\vdash\Gamma, A, B, C\wedge D}}{\Sigma\vdash\Gamma, A\vee B, C\wedge D} \qquad \dfrac{\dfrac{\Sigma\vdash\Gamma, A, B, C}{\Sigma\vdash\Gamma, A\vee B, C} \quad \dfrac{\Sigma\vdash\Gamma, A, B, D}{\Sigma\vdash\Gamma, A\vee B, D}}{\Sigma\vdash\Gamma, A\vee B, C\wedge D}$$

   Here the choice of which rule to use first is irrelevant. This is not always the case, if we consider on the other side

$$\dfrac{\dfrac{\dfrac{\Sigma\vdash\Gamma, At_1, At_2}{\Sigma\vdash\Gamma, At_1, \exists x.\, A}}{\Sigma\vdash\Gamma, \exists x.\, A, \exists x.\, A} \quad \dfrac{\Sigma\vdash\Gamma, At_1, At_1}{\Sigma\vdash\Gamma, At_1}}{\Sigma\vdash\Gamma, \exists x.\, A} \qquad \dfrac{}{\Sigma\vdash\Gamma, \exists x.\, A}$$

   By reading the proof trees bottom-up, here the contraction rule has to be used before the existential rule, if we want to be able to create the two distinct instances for t_1 and t_2.

An additional problem comes from the analysis of cut elimination. Although the study of proof reductions is not the center of this work, a major source of concern about *LK* was the fact that the set of reductions given by Gentzen for his *Hauptsatz* was not confluent, and thus choices made during the simplification phase could lead to totally different arguments as result of the transformation. The foundational status of sequent calculus was greatly contested by these criticisms.

Several analyses in the beginning of the nineties, especially after Girard's Linear Logic [Gir87a], led to discoveries of a finer structure of the sequent calculus. We can trace the origins of what became known as *focused* proof systems to a few authors:

- Uniform proofs [Mil+91] are a study on goal-directed proofs, that are composed of an alternation of deterministic goal-reduction phase and a non-deterministic phase of clause selection. They were introduced as a proof-theoretic foundation for logic programming languages (and constitute the foundation of λProlog).

- Andreoli's work on focused proofs for linear logic [And92] introduced the general notion of focused proof, and provided an analysis of full linear logic based on a system akin to uniform proofs.

- Herbelin's LJT sequent calculus and $\bar{\lambda}$ calculus [Her95] linked the ideas of focusing to concepts in functional programming languages.

The first substantial realization behind the original works on focused proof theory is that we should classify the rules of *LK* into *invertible* and *non-invertible*. Rules of the first type are those where knowledge of the end sequent of the inference guarantees knowledge of the premise sequent. This is contrasted by non-invertible rules, where knowledge of the end sequent of the inference does not suffice to guarantee knowledge of the premise sequent. Consider the two inferences:

$$\frac{\Sigma \vdash \Gamma, t : \iota \quad \Sigma \vdash \Gamma, [t/x]A}{\Sigma \vdash \Gamma, \exists x.\, A} \qquad \frac{\Sigma \vdash \Gamma, A, B}{\Sigma \vdash \Gamma, A \vee B}$$

In the inference on the left, merely knowing $\exists x.\, A$ is not enough to know that it is t for which A holds. This additional knowledge was part of the inference that allowed us to conclude $\exists x.\, A$. On the contrary, in the inference on the right knowing that $A \vee B$ holds is enough to conclude that one of A, B holds given Γ. Thus, the rule we gave for $\exists$ is not invertible, while the rule for $\vee$ is. When dealing with invertible rules, no choice has been made during the inference: the focusing properties of proof systems show that these rules can be chained, and their permutation is irrelevant. Conversely, when dealing with non-invertible rules, sequences of choices can be made in succession, so that an entire sequence of choices can be undone if we are searching for a proof and hit a dead end. Notice however that we could give non-invertible rules for $\vee$ as well:

$$\frac{\Sigma \vdash \Gamma, A}{\Sigma \vdash \Gamma, A \vee B} \qquad \frac{\Sigma \vdash \Gamma, B}{\Sigma \vdash \Gamma, A \vee B}$$

In this version, information is needed to know from which one of the two disjuncts $A \vee B$ has been concluded. The early results on focusing showed that rules of each category can be chained, so that the possible backtracking points are greatly reduced.

The second substantial realization is the discovery of polarity. In addition to classifying rules, we can classify formulas based on whether they are to be inferred by an invertible or non-invertible rule. Consider formula occurrences in the single-sided sequent calculus (or equivalently, as the consequent in a dual sided sequent calculus):

1. Formulas which are built from a connective that is introduced with a non-invertible rule should be classified as positive. This captures the idea that the meaning of the formula is given by its introduction rule.

2. Formulas which are built from a connective that is introduced with an invertible rule should be classified as negative. This captures the idea that the meaning of the formula is given by how we draw consequences from it.

3. This same division also applies to atomic constants. The inference that characterizes atoms is init, and thus we get that an atom is negative if we can conclude something from knowing it, while it is positive if in order to prove it we need to know it is among the premises.

Since the $\wedge$ and $\vee$ connectives can be presented both with invertible and non-invertible rules, they should be split into two different connectives that we denote $\dot{\wedge}, \dot{\vee}$ and $\bar{\wedge}, \bar{\vee}$. The quantifiers have a fix polarity, namely $\exists$ is positive while $\forall$ is negative. The intuitionistic implication has a negative polarity. Atoms have an assigned polarity, but we only mention it at the meta-level. Finally, the $\cdot^{\perp}$ operator flips the polarity of the connectives. The concepts of polarity and focusing phases proved fundamental in the study of functional languages as well, where negative types are understood as computations and positive types are understood as values [CH00; Zei], and different polarization strategies can represent different evaluation strategies.

Ultimately *focused* calculi for classical and intuitionistic logics, akin to Andreoli's calculus for linear logic, have been presented by Liang and Miller in [LM09]. Their system *LKF* of focused, single sided classical logic is presented in fig. 1.3, and is the system upon which our work is carried out. Sequents in *LKF* extend those in *LK*, and come in two types: sequents of the form $\Sigma \vdash \Gamma \Downarrow F$ are called synchronous, those of the form $\Sigma \vdash \Gamma \Uparrow \Theta$ are called asynchronous. As before, Σ is a typing context and Γ is a multiset of formulas. We now call Γ the *storage* area. The new additions are Θ, a list of formulas, and F, a select formula which is said to be under focus.

The rules of *LKF* are divided into four blocks that are a refinement of the blocks in fig. 1.2. Asynchronous rules treat asynchronous sequents, and are invertible, while synchronous rules treat synchronous sequents and are non-invertible. The system is *fully focused*: asynchronous rules continue decomposing the goals on the right of $\Uparrow$, until positive subformulas or literals are exposed and moved to the storage area; then, the decide rule places one positive formula under focus, and the branches in the synchronous phase continue decomposing the formula until either they end by init, or a negative subformula is found and a release reinstates the asynchronous phase.

Given a classical formula F, we call $\hat{F}$ a polarization of F if it consists of F where all the connectives have been replaced by polarized versions, and where the atoms have polarity information (thus, there is an exponential amount of possible polarizations). Then we have

Theorem 1 (Soundness and completeness)**.** *If F is a formula in negation normal form and $\hat{F}$ is any polarization of F, F is provable in LK if and only if $\hat{F}$ is provable in LKF.*

Asynchronous rules

$$\frac{\Sigma \vdash \Gamma \Uparrow \Theta, A \quad \Sigma \vdash \Gamma \Uparrow \Theta, B}{\Sigma \vdash \Gamma \Uparrow \Theta, A \mathbin{\bar{\wedge}} B} \qquad \frac{}{\Sigma \vdash \Gamma \Uparrow \Theta, \bar{\top}} \qquad \frac{\Sigma \vdash \Gamma \Uparrow \Theta, A, B}{\Sigma \vdash \Gamma \Uparrow \Theta, A \mathbin{\bar{\vee}} B}$$

$$\frac{\Sigma \vdash \Gamma \Uparrow \Theta}{\Sigma \vdash \Gamma \Uparrow \Theta, \bar{\bot}} \qquad \frac{\Sigma, y \vdash \Gamma \Uparrow \Theta, [y/x]A}{\Sigma \vdash \Gamma \Uparrow \Theta, \forall x.\, A}\ y \notin \Sigma$$

Synchronous rules

$$\frac{\Sigma \vdash \Gamma \Downarrow A \quad \Sigma \vdash \Gamma \Downarrow B}{\Sigma \vdash \Gamma \Downarrow A \mathbin{\hat{\wedge}} B} \qquad \frac{}{\Sigma \vdash \Gamma \Downarrow \hat{\top}} \qquad \frac{\Sigma \vdash \Gamma \Downarrow A}{\Sigma \vdash \Gamma \Downarrow A \mathbin{\hat{\vee}} B} \qquad \frac{\Sigma \vdash \Gamma \Downarrow B}{\Sigma \vdash \Gamma \Downarrow A \mathbin{\hat{\vee}} B}$$

$$\frac{\Sigma \vdash t \quad \Sigma \vdash \Gamma \Downarrow [t/x]A}{\Sigma \vdash \Gamma \Downarrow \exists x.\, A}$$

Identity rules

$$\frac{}{\Sigma \vdash \Gamma, \neg p \Downarrow p}\ \text{init} \qquad \frac{\Sigma \vdash \Gamma \Uparrow A \quad \Sigma \vdash \Gamma \Uparrow A^{\perp}}{\Sigma \vdash \Gamma \Uparrow \cdot}\ \text{cut}$$

Structural rules

$$\frac{\Sigma \vdash \Gamma, R \Uparrow \Theta}{\Sigma \vdash \Gamma \Uparrow R, \Theta}\ \text{store} \qquad \frac{\Sigma \vdash \Gamma, P \Downarrow P}{\Sigma \vdash \Gamma, P \Uparrow \cdot}\ \text{decide} \qquad \frac{\Sigma \vdash \Gamma \Uparrow N}{\Sigma \vdash \Gamma \Downarrow N}\ \text{release}$$

In the store rule, R is a positive formula or a literal

Figure 1.3: Rules of *LKF*. Γ is a multiset of positive formulas or literals, and Θ is a list of formulas.

At the opposite extremes of the focusing discipline we find the *purely positive* or *purely negative* polarizations. The former terminology is for formulas whose connectives and atoms are all positive (and thus can't contain $\forall$), while the latter is all negative (and thus can't contain $\exists$). Proofs of formulas of the first kind will only use the storage for negated literals; since the only way to end a proof is by having the complement of a stored literal under focus, they will need to be steered through many decisions on $\hat{\vee}$ rules and instantiations on $\exists$ rules in order to find exactly the two opposite literals in each branch. On the opposite, proofs of purely negative formulas proceed by decomposing the subproof into lots of literals that are stored and only at the end a decide rule can be used, on one of the stored literals for which its complementary has been found.

If we restrict to propositional logic, we see that the same formula could be polarized either fully negatively or fully positively. Then, if the formula is provable, both polarizations are provable; however their proofs may look completely different.

1.2.2 Foundational proof certificates

The focused sequent calculus for classical logic provides answers to all criticisms that were brought out in subsection 1.2.1: now we have a notion of for-

mula under focus that can explain inference; irrelevant rule permutations are hidden by abstracting focused phases; contraction is precisely controlled by means of finer structural rules; cut reductions become deterministic. Moreover, we already got a hint at how different polarizations can lead to proof requiring different amounts of information: in general, performing an inference on a positive connective requires external information, while performing an inference on a negative connective can be reconstructed automatically by a proof checker. At the same time we are just refining Gentzen's system, as proofs in *LK* can be retrieved from proofs in *LKF* by simply ignoring the focusing and polarity information. This provides a much better ground where to anchor a formal semantics of proof languages.

The framework of *Foundational Proof Certificates* is centered around the division between a *kernel* and a *client*. The kernel implements a version of *LKF* geared towards proof-search, that should be trusted and performs the proof certification. The client provides a proof object, together with a description of how it should be used to build a full proof. Since the asynchronous rules don't need external information in order to be performed by the kernel, the corresponding external evidence just needs to be transformed so that it is suitable to be dispatched to the premises (which might be two distinct branches); we call these dispatchers *clerks*. In the case of the asynchronous rules, instead, the kernel wishes to obtain information from the proof evidence. This information can be seen as coming in three kinds:

1. In the $\dot{\vee}$ rule, a left/right bit is needed in order to decide which disjunct to choose.
2. In the $\exists$ rule, a well-formed term is needed in order to create an instance.
3. In the decide and init rules, an index is needed in order to know which formula should be selected. This index should be assigned to the formula in correspondence with the store rule.

The framework we are introducing is developed in the context of logic programming. This programming paradigm is particularly well suited for the encoding of this kind of checkers, since the combination of unification and backtracking proof search make it easy to treat partial proof evidence: any of the components specified above might be left out in an evidence, and the checker will try its best to fill the gaps. Therefore, the natural consequence is to implement the clerks and experts as relational specifications, denoting relations between the proof evidence available in the end sequent of a rule application and that which will be handed to the premises. The full system is built by asking that at each inference the corresponding clerk or expert holds of the available proof evidence. We call the system LKF^a, for Augmented LKF, and the full rules are presented in fig. 1.4

We fix the types $cert$ of certificates, $index$ of formula index and $choice$ of left/right choices. The sequents of LKF^a have the forms $\Xi; \Sigma \vdash \Gamma \Uparrow \Theta$ and $\Xi; \Sigma \vdash \Gamma \Downarrow F$, where Σ, F and Θ are like in *LKF*, Ξ is a certificate, and Γ is now an indexed storage, that is a multiset of pairs of the form $i : F$ where i is an index and F is a formula. The rules of LKF^a mimic those of *LKF*, but add the condition that the certificate satisfies the clerks and experts at each rule.

Asynchronous rules

$$\frac{\Xi_1; \Sigma \vdash \Gamma \Uparrow A, \Theta \quad \Xi_2; \Sigma \vdash \Gamma \Uparrow B, \Theta \quad \wedge_c(\Xi_0, \Xi_1, \Xi_2)}{\Xi_0; \Sigma \vdash \Gamma \Uparrow A \mathbin{\bar{\wedge}} B, \Theta}$$

$$\frac{}{\Xi_0; \Sigma \vdash \Gamma \Uparrow \bar{\top}, \Theta}$$

$$\frac{\Xi_1; \Sigma \vdash \Gamma \Uparrow A, B, \Theta \quad \vee_c(\Xi_0, \Xi_1)}{\Xi_0; \Sigma \vdash \Gamma \Uparrow A \mathbin{\bar{\vee}} B, \Theta} \qquad \frac{\Xi_1; \Sigma \vdash \Gamma \Uparrow \Theta \quad \bot_c(\Xi_0, \Xi_1)}{\Xi_0; \Sigma \vdash \Gamma \Uparrow \bar{\bot}, \Theta}$$

$$\frac{(\Xi_1\ y); \Sigma, y : i \vdash \Gamma \Uparrow [y/x]A, \Theta \quad \forall_c(\Xi_0, \Xi_1)}{\Xi_0; \Sigma \vdash \Gamma \Uparrow \forall x.\, A, \Theta}\ y \notin \Sigma$$

Synchronous rules

$$\frac{\Xi_1; \Sigma \vdash \Gamma \Downarrow A \quad \Xi_2; \Sigma \vdash \Gamma \Downarrow B \quad \wedge_e(\Xi_0, \Xi_1, \Xi_2)}{\Xi_0; \Sigma \vdash \Gamma \Downarrow A \mathbin{\hat{\wedge}} B} \qquad \frac{\top_e(\Xi_0)}{\Xi_0; \Sigma \vdash \Gamma \Downarrow \hat{\top}}$$

$$\frac{\Xi_1; \Sigma \vdash \Gamma \Downarrow A_i \quad \vee_e(\Xi_0, \Xi_1, i)}{\Xi_0; \Sigma \vdash \Gamma \Downarrow A_1 \mathbin{\hat{\vee}} A_2}\ i \in \{1, 2\}$$

$$\frac{\Sigma \vdash t : i \quad \Xi_1; \Sigma \vdash \Gamma \Downarrow [t/x]A \quad \exists_e(\Xi_0, \Xi_1, t)}{\Xi_0; \Sigma \vdash \Gamma \Downarrow \exists x.\, A}$$

Identity rules

$$\frac{\mathsf{init}_e(\Xi_0, l)}{\Xi_0; \Sigma \vdash \Gamma, l{:}\neg p \Downarrow p}\ \mathsf{init}$$

$$\frac{\Xi_1; \Sigma \vdash \Gamma \Uparrow A \quad \Xi_2; \Sigma \vdash \Gamma \Uparrow A^{\perp} \quad \mathsf{cut}_e(\Xi_0, \Xi_1, \Xi_2, A)}{\Xi_0; \Sigma \vdash \Gamma \Uparrow \cdot}\ \mathsf{cut}$$

Structural rules

$$\frac{\Xi_1; \Sigma \vdash \Gamma, l{:}P \Downarrow P \quad \mathsf{decide}_e(\Xi_0, \Xi_1, l)}{\Xi_0; \Sigma \vdash \Gamma, l{:}P \Uparrow \cdot}\ \mathsf{decide}$$

$$\frac{\Xi_1; \Sigma \vdash \Gamma \Uparrow N \quad \mathsf{release}_e(\Xi_0, \Xi_1)}{\Xi_0; \Sigma \vdash \Gamma \Downarrow N}\ \mathsf{release}$$

$$\frac{\Xi_1; \Sigma \vdash \Gamma, l{:}R \Uparrow \Theta \quad \mathsf{store}_c(\Xi_0, \Xi_1, l)}{\Xi_0; \Sigma \vdash \Gamma \Uparrow R, \Theta}\ \mathsf{store}$$

In the store rule, R is a positive formula or a literal

Figure 1.4: Rules of LKF^a, an augmented version of *LKF*. Γ is a multiset of pairs of the form $l{:}R$ where l is an index and R is a positive formula or literal, and Θ is a list of formulas.

```
type dd     int → cert.
type indx   index.
% The decide expert reduces the depth by one
decide_ke (dd D) (dd D') indx :- D > 0, D' is D - 1.
% All other clerks and experts thread through the certificate
andPos_ke  (dd D) (dd D) (dd D).
andNeg_kc  (dd D) (dd D) (dd D).
initial_ke (dd _D) indx.
orNeg_kc   (dd D) (dd D).
orPos_ke   (dd D) (dd D) _Dec.
all_kc     (dd D) (x\ dd D).
false_kc   (dd D) (dd D).
release_ke (dd D) (dd D).
some_ke    (dd D) (dd D) _Term.
store_kc   (dd D) (dd D) indx.
true_ke    (dd _D).
```

Figure 1.5: The decide-depth FPC

1.2.3 Some examples of FPC definitions

Decide depth A very simple definition for an FPC is the one that simply limits the number of bipoles that compose the proof: this is done by bounding the number of decide rules on every branch of the proof. Since no polarization is assumed, proofs thus bounded can vary wildly: if a negative polarization is chosen for propositional classical logic, in particular, any formula can be proven with just one usage of the decide rule, just below the initial rule at every leaf. More than the actual size of the proof, decide depth limits the amount of external guidance that the proof is allowed to take. A presentation of the clerks and experts is given in fig. 1.5.

Resolution refutations A more interesting example, that we will use in section 2.2 is the FPC for resolution refutations, presented in [CMR17] and extended in [Bla17].

Resolution is a method for building refutations of formulas that are in conjunctive normal form, that is formulas of the form $C_1 \wedge \dots C_n$ where each of the C_i (called a *clause*) is of the form $L_1 \vee \dots L_n$ and each of the L_i is a literal, that is either an atom, a negated atom or $\bot$.

Definition 3 (Resolution refutation)**.** The resolution rule is:

$$\frac{L_1 \vee \dots L_n \vee a \quad O_1 \vee \dots O_n \vee \neg a}{L_1 \vee \dots L_n \vee O_1 \vee \dots \vee O_n}$$

A resolution refutation for a formula $C_1 \wedge \dots C_n$ is built out of the following three lists:

- The list of initial clauses $C_1, \dots C_n$
- A subsequent list of clauses $C_{n+1}, \dots C_k$, where the clause $\bot$ appears

```
% Deduced clauses
type lemma int → form → o.
% Stored clauses
type idx int → index.
% Stored literals
type lit index.
% Initial indexing
type start int → list cert →
    cert.
% List of resolution triples
kind triple type.
type rlist list triple →
    cert.
% Extract index of new clause
type rlisti int → list
    triple → cert.
% Resolvent triple
type resolve int → int → int
   → triple.
% Check resolution step
type dlist list int → cert.
% End with initial
type rdone cert.
% End of left cut premise
type done cert.
```

Figure 1.6: Signature of the resolution FPC

- A list of triples $\langle i, j, k\rangle$, indicating that C_k has been obtained by the resolution rule from C_i and C_j

From a model theoretic perspective refuting a set of clauses amounts to showing that no model satisfies the set, and in many cases model-theoretic reasoning is used (see [RV01]). However, most theorem provers can be instructed to output a trace in a format akin to that of definition 3.

In the single-sided calculus *LKF*^{a} we can interpret the refutation of a clause normal form formula $C_1 \wedge \ldots \wedge C_n$ as a direct proof of the co-clause normal form formula ${C_1}^\perp \vee \ldots \vee {C_n}^\perp$, where each of the co-clauses ${C_i}^\perp$ is a conjunction of literals. Following the approach of [CMR17], the choice of polarization is $\dot{\wedge}$ for the conjunction and $\bar{\vee}$ for the disjunction in the formula we prove: this means that the disjunction between co-clauses is automatically decomposed, while the conjunction that makes up the co-clauses requires naming that will be used throughout the proof.

The signature for the FPC is in fig. 1.6. Indexes have three constructors: the main one builds an index from an integer, and is meant to directly replay the indexing of clauses in the refutation; the other two are used in the replication of the resolution rule. Correspondingly, the code for the FPC in fig. 1.7 can be divided into three zones: a starting zone that indexes the clauses, and two other that are used to replicate the resolution rule.

The implementation of the clerks and experts is in fig. 1.7, and consists of three phases: the first phase where the initial clauses of the goal formula are indexed; a second phase where all subsequent resolution steps are translated to cut inferences; a third phase where the left-hand sides of all resolution steps are checked.

The constructor `start` gives rise to the following synthetic inference rule:

$$\frac{\texttt{start (n + 1) R} \vdash \texttt{idx 1} : {C_1}^\perp, \ldots, \texttt{(idx n)} : {C_n}^\perp \Uparrow \cdot}{\texttt{start 1 R} \vdash \cdot \Uparrow {C_1}^\perp \mathbin{\bar{\vee}} \ldots \mathbin{\bar{\vee}} {C_n}^\perp}$$

Since we chose the negative polarity for disjunction, this is forcefully used right away. Once there is nothing left on the right-hand side of the arrow, we can use either cut or decide. Since the certificate we have at the end of this

synthetic rule is built with `start`, the only expert that can be used is cut: this switches to the second certificate constructor, `rlist`, that directly manipulates the trace of the resolution refutation given as triples:

$$\dfrac{\texttt{dlist [i,j]} \vdash \Gamma \Uparrow C_k \qquad \dfrac{\texttt{rlist R} \vdash \Gamma, \texttt{idx k} : {C_k}^{\perp} \Uparrow \cdot}{\texttt{rlisti k R} \vdash \Gamma \Uparrow {C_k}^{\perp}}\ \text{store}}{\texttt{start (n + 1) (resolve i j k :: R)} \vdash \Gamma \Uparrow \cdot}\ \text{cut}$$

Thus begins the second phase, where all resolution steps are translated to cuts, in a right-branching fashion. The index k for the newly built clause is passed to the `lemma` predicate, which is provided by the user and should contain the clause that is built by the resolution step; it is then also shipped to the intermediate certificate constructor `rlsti`, so that the new co-clause is stored at the right index. This continues until the list of triples is empty: at this point, the resolution refutation has been completely processed, and a clause $\perp$ must have been encountered; therefore, a co-clause $\top$ is present in the sequent and the decide expert non-deterministically looks for it so that the `trueE` expert can conclude the proof.

The left-hand branch of each of these cut inferences is what is left to check in the third phase. Here, we want to use the knowledge that clauses C_i and C_j resolve to give clause C_k in order to build a proof of C_k in a context where the co-clauses ${C_i}^{\perp}$ and ${C_j}^{\perp}$ are available. Since we chose to polarize conjunctions positively in the co-clause, the clauses like C_k are built with the dual of $\barwedge$ and are therefore negative disjunctions of literals. The code for the FPC allows the blind decomposition of these disjunctions, and instructs the kernel to store all the uncovered literals at a same address, resulting in the following:

$$\dfrac{\dfrac{\dfrac{\texttt{dlist [j]} \vdash \Gamma, \Lambda \Downarrow a_1 \quad \texttt{dlist [j]} \vdash \Gamma, \Lambda \Downarrow a_2 \quad \texttt{dlist [j]} \vdash \Gamma, \Lambda \Downarrow a_2}{\texttt{dlist [j]} \vdash \Gamma, \texttt{lit} : L_1, \ldots, \texttt{lit} : L_q \Downarrow C_i^{\perp}}}{\texttt{dlist [i,j]} \vdash \Gamma, \texttt{lit} : L_1, \ldots, \texttt{lit} : L_q \Uparrow \cdot}}{\texttt{dlist [i,j]} \vdash \Gamma \Uparrow C_k}$$

The certificate `dlist [i,j]` contains the indexes of the two co-clauses to be resolved, among which `decideE` picks one nondeterministically, and leaves the other in the resulting certificate. Say the expert picks i, resulting on a decide on the co-clause which will be of the form ${C_i}^{\perp} = a_1 \barwedge a_2 \barwedge_3$. Then, several branches corresponding to the decomposition of $\barwedge$ are created. In each branch the proof will be concluded either by immediately finding a complementary stored literal arising from the decomposition of C_k, or by storing the literal under focus and deciding on co-clause ${C_j}^{\perp}$ to finding the complementary literals for the second co-clause.

1.2.4 Hosting calculi

The declarative essence of Foundational Proof Certificates allows for a great variety of proof formats that can be specified, and an equivalently great variety of programming methodologies that can be used to encode them. For example, the user can craft a modular specification where common preprocessing steps that appear inside proofs are specified by means of simple macros or more

```
orC      (start Ct Certs) (start Ct Certs).
falseC   (start Ct Certs) (start Ct Certs).
storeC   (start Ct Certs) (start 'Ct Certs) (idx Ct) :-
         inc Ct 'Ct.
cutE     (start _ Certs) C1 C2 Cut :-
         cutE (rlist Certs) C1 C2 Cut.

cutE     (rlist (resolve I J K ::Certs))
         (dlist [I,J]) (rlisti K Certs) Cut :-
         lemma K Cut.
falseC   (rlist Rs) (rlist Rs).
storeC   (rlisti K Rs) (rlist Rs) (idx K).
decideE  (rlist []) rdone (idx I).
trueE    rdone.

% Left premise
allC     (dlist L) (x\ dlist L).
orC      (dlist L) (dlist L).
falseC   (dlist L) (dlist L).
storeC   (dlist L) (dlist L) lit.
decideE  (dlist L) (dlist [J]) (idx I) :- L = [I,J] ; L =
    [J,I].
decideE  (dlist [I]) (dlist []) (idx I).
decideE  (dlist L) (dlist []) lit :- L = [I] ; L = [].
initialE (dlist L) lit.
trueE    (dlist _L).
andE     (dlist L) (dlist L) (dlist L).
someE    (dlist L) (dlist L) T.
releaseE (dlist L) (dlist L).
```

Figure 1.7: Implementation of the resolution FPC

complex subroutines. Thus, we can have certificates that are themselves interpreters for some domain-specific certificate format, that automatically benefits of the soundness provided by the general framework. But what if we couple this with a logical translation of another object logic inside the client logic? Then what we obtain is a device that allows for a faithful encoding of proofs in different user-provided systems on top of a kernel, unbeknownst to the kernel itself.

This has been called *hosting* of kernels [Chi15; CIM16], and is based on ideas that have been used in presentations of focused proof systems since their inception [LM07].

The task of the client in these cases includes an additional step. First of all, they must provide adequate meaning for the clerks and experts of the object logic they wish to encode by composing the meaning of the kernel's clerks and experts. In addition to this, they must also provide a logical transformation from the object logic into the logic of the kernel. The fact that this logical transformation preserves meaning is not checked by the kernel, but has to be

taken care of by the user. Therefore, it lies out of the trusted zone. Some useful examples of hosted systems are:

- The two-sided version of *LKF* can be reconstructed on top of LKF^a by following the same argument that was presented in subsection 1.1.2. The logical transformation here is the transformation to negation normal form of definition 2. In the two-sided classical logic we have left-introduction rules that have the dual role of right-introduction rules; every negative connective has therefore an expert on the left, and every positive connective has a clerk on the left. The object logic encoding contains clerks and experts for the $\supset$ negative connective that keep track of the fact that the proof evidence of the antecedent needs to be treated by the dual connective.
- Double negation embeddings of classical logic into intuitionistic logic can be given an account based on polarity [CIM16; FO12]. This made it possible to implement LKF^a as a hosted kernel on top of the LJF^a calculus for intuitionistic logic.

1.3 Proof Certificates and syntactic foundation

So far we have argued for the foundational status of foundational proof certificates and focused proof theory mainly on the grounds of simplicity and expressivity. More points have been made about this program, and more programs were built on similar concepts with a more distinctly foundational flavor.

In the work of Nigam and Miller [NM10] the focused proof theory of linear logic has been used as the metatheory for describing proof systems. Thanks to this, they discovered that some proof systems (such as Natural Deduction and the sequent calculus) can be described as making different meta-level choices about polarization: that is, whether the antecedents and consequents of sequents are themselves to be treated as positive or negative entities, whether they should be eagerly decomposed or only externally inspected. Henriksen later showed that an intuitionistic focused metatheory already suffices for this characterization[Hen10], and active research continues in this field, trying to uniformly establish meta-theoretic results about proof systems expressed in a focused metatheory.

The work of Nigam and Miller proposes three levels of adequacy when discussing the encoding of proof systems:

- Relative completeness is established when the provable formulas in the encoding are the same as in the codified system.
- Full completeness of proofs is established when the completed proofs are in one-to-one correspondence.
- Full completeness of derivations requires in addition that open derivations (such as inference rules themselves) are also in one-to-one correspondence.

The FPC framework does not require a focused meta-theory to be carried out, and it does not target meta-theoretic results per se. However, it has first

class handling of polarities, and it can be seen as achieving full completeness since the user specification have a granularity that can be constrained enough that it can mimic single rules. What handling open derivations means in this context is still not fully explored; however, we had to take this into account when designing the integration of FPC checkers into elaborators for proof assistants. The nature of derivation in proof assistants is naturally that of being incomplete, since for the most part of their development the user is working on them; despite this, the user might want to get access to external information in the process of proving. This includes the cases where the user attempting to prove something that turns out not to be provable, but asks an external prover to solve a goal that happens to be solvable nonetheless. Or the more tricky case where the incautious user is again attempting to prove a falsehood, and asks for a counterexample (that is, a proof of the negation of their claim): then, it would be much more useful if instead of discarding the current work the user could interact with the counterproof in order to find the precise spot where the argument is fallacious.

The search for a foundation of syntax by including partial derivation as first class elements is reminiscent of the project of J.-Y. Girard (from the early works on focusing [Gir91] to the work on Ludics [Gir01] up until the most recent program of Transcendental Syntax [Gir17; Eng]). These projects start from the realization that there is always some sort of circularity in the grounding of logical systems, whereby when trying to explain inference one has to resort, in some form or another, to a metatheoretic inference that is taken as granted. The innovation that the project of Transcendental Syntax aims to bring, in particular, is also presented by a system of levels that partly relates to Miller and Nigam's. In Girard's view, the levels are:

- Level -1 is the analysis of truth embodied by Tarskian Semantics. Here truth is defined as a meta-level mathematical construction, and theoremhood is checked against it. If the test passes we are fine, however nothing is known about distinct propositions except for the fact that they are claimed to be true.

- Level -2 takes proof into account, and is exemplified by the Curry-Howard view, the Brouwer-Heyting-Kolmogorov semantics for intuitionistic logic, or realizability. A proof formalism is presented that has a particularly good behavior, and meaning is rooted upon this formalism. Proof become useful objects, and they are also used as computational objects.

- Level -3 is represented by the projects of Geometry of Interaction and Ludics. The proof formalism disappears and is replaced by more abstract constructions. In particular, constructions that are possibly not proofs are allowed. These constructions come with a notion of interaction (that is the precursor of the reductions of a cut elimination procedure), and objects that can give rise to proofs are characterized by how they interact with counter-proofs.

- Level -4 where the new Transcendental Syntax project is placed. The need to reach this level is due to the fact that Level -3 does not provide a finitary test for validity of a proof: in principle one would have to test

> a candidate proof-object with any possible counterproof. At this level, instead, one wishes to find finite tests for validity of the pre-proof objects. An example of this is the correctness criteria for Proof Nets in (fragments of) Linear Logic. The main technical development that is being carried out for this new project is *stellar resolution* [Eng].

It should be clear from the discussion we had so far that the project of Foundational Proof Certificates easily gives an account for concepts at least up until Level -2. In particular, in chapter 2 we will present work that tries to free some optimizations in automated theorem proving from being stuck at Level -1, and in chapter 4 we try to improve the situation of proof assistants being at Level -2.

The general program of FPC however aims at level -3, where a language is provided for the generic representation of proofs, and interaction with a (possibly incomplete) object of this language is possible. How and if FPCs can be brought to level -4 is unclear: at the moment, since not even level -3 is completed, the only test we have for a certificate is at level -2, that is the reconstruction of a full proof. We believe this discussion to be relevant nonetheless, given that the notions of focusing and polarity are central to both projects; ultimately, one could say that Girard's project aims at using these to rebuild logic from inside out, while the FPC project aims at further refining the structures of sequent calculus from outside in.

Another presentation of a proof-theoretic foundation that takes the views of polarized logic at its core is the work of Zeilberger [Zei08]. Zeilberger revisits the pragmatist and verificationist meaning theories of Dummett [Dum93] using the judgmental method of Martin-Löf [ML96]. Dummett proposed two approaches to the foundations of Natural Deduction: in the verificationist meaning theory the meaning of a connective is given by its introduction rule, while its elimination rule of Natural Deduction is justified by it[2]; in the pragmatist meaning theory the meaning of a connective is given by its elimination rule, and the introduction rule is determined by it.

Zeilberger showed that the alternative views of pragmatist and verificationist meaning theories can be reconciled, and argues that this distinction is coincident with the distinction between positive and negative polarities. A verificationist meaning theory serves to determine the meaning of positive connectives, while a pragmatist theory serves to determine the meaning of negative ones. Echoes of this exposition have been used in subsection 1.2.1 to introduce the concepts of polarity. However, the foundational work of Zeilberger (as well as its references in Dummett, Prawitz and Martin-Löf) is centered around natural deduction, and inherits from that system some technical difficulties for our scopes.

[2]This is a view that is also found in comments made by Gentzen [Gen35], and is held among others by Prawitz [Pra06]

Chapter 2

Names for terms and subproofs

The format of FPC operates a clear separation into the information that is provided by the user and the part of deduction that an automated checker can deal with in all cases. It enables also a finer distinction of the different kinds of provided information, as well as their different structure and their interaction with the proving process. In particular, we can note that experts need to obtain three kinds of information:

- Bits for choices such as left or right in the $\vee$ rule
- Terms for the instantiation of quantifiers
- Indexes for the store, decide and initial rules
- Formulas to be introduced by the cut rule

The structure for the first kind of data is rapidly explained: the information to be exchanged amounts to nothing more than an indication on which premise to choose; client and kernel should therefore agree on such a format that contains two alternatives, and nothing more[1]. The treatment of formulas, on the other side, is well explained by the theory of polarization. The two remainig kinds of information, terms and indexes, can be explored further.

Terms make part of the language of first order logic; however, the client need not use the same naming structures as the kernel. This is in particular the case when automated theorem provers introduce new terms in order to increase their performance. Indexes are a concept that derives from the analysis of focused proof systems, and they are not present in standard presentations of logic; however, they model a concept that is a natural part of the description of a deductive process that is the act of keeping track of the hypothesis or goals that are either under analysis or postponed. Indexes provide a viable tool to further understand the different ways by which knowledge is accessed or made available. In this chapter, we will see in more detail how by a careful analysis of these two structures we can recover a pure proof-theoretic understanding of two common optimization techniques in theorem proving.

[1] Nevertheless, the amount of information that a prover decides to share may vary considerably: for example, the most recent draft for the Alethe format to be used by `veriT` and `cvc5` omits completely these bits and only records that a projection rule was used.

2.1 Communicating terms

In addition to the logical constants, first order logic comes with two kinds of non-logical symbols: predicate symbols and term (or function) symbols. The structure of predicate symbols is rather clear: they are used to build the atoms from which formulas are built, and this atom building process depends on their arity. The way the atoms are then organized in formulas depends more on our view on the logical symbols (for example, we have presented the λ-tree syntax in section 1.1, but other choices could have been taken) rather than on the predicate symbols themselves. The only addition we require to the common presentation of the syntax of predicate symbols in FPCs is that they are to be divided into two categories, namely the positive and the negative.

The situation is slightly more complicated for term symbols. We can start by observing that the usual presentation of first order terms as being built up of function symbols and variables needs to be augmented with the notions of free and bound variable, both of which depend on the particular occurrence of the term. This induces the notions of scoping and dependency over the variables that occur in a formula, and common tasks in theorem proving such as unification need to deal with them in a way that can lead to different implementations of the same concepts. Tightly related to this distinction is the necessity of identifying during the proof process those occurrences of variables that should denote new, generic individuals (as opposed to those individuals that are denoted by terms and variables that occur elsewhere in the proof's assumptions).

Reasoning with quantifiers requires the notion of a generic name. For example, when we want to conclude *"all philosophers are mortal"*, we will try to infer for an arbitrary $person$ that *"if* $person$ *is a philosopher then* $person$ *is mortal"*. Tarskian semantics is deeply unsatisfactory, as it often is, since it treats generic individual with its *weird old trick*: it replaces it with a meta-level notion. Therefore, the notion of generic name is hidden away by appealing to the meta-level quantification, where it is assumed that the reader knows how to think about a generic *person* and no real explanation for what is going on is given[2]. In particular, the notion of dependency is completely hidden away. Proof theory is left with the entire task of formalizing generic elements, and it is here that we meet concepts that might seem esoteric, since a distinction needs to be made between individuals and *generic* individuals.

In order to capture this distinction, Gentzen's proof systems introduce one additional category of terms that are called *eigenvariables*, whose role in the proof process is precisely to be generic individuals handled by one of the quantifier rules. Eigenvariables can both be presented as being in their own syntactic category or being variables with specific constraints. An alternative to eigenvariables is Hilbert's ϵ operator. Most theorem provers, however, rely on skolemization: a device that internalizes most notions we discussed into the syntax of terms by extending the language with fresh uninterpreted functions, and thus removes the need for special care in the proof search process.

Formats and systems for proof exchange make the assumption that the proof-checking kernel and the user agree entirely on the signature. However

[2]Kit Fine proposed an extension of a formalized semantics in the style of Tarski where a primitive notion of generic element is included [Fin85]. His aim was to create a model theory that could explain proofs instead of mere provability, and in particular explain the notion of eigenvariable.

$$\frac{\Xi_1; \Sigma, (\texttt{copy}\ t\ y) \vdash \Gamma \Uparrow [y/x]A, \Theta \quad \forall_c(\Xi_0, \Xi_1, t)}{\Xi_0; \Sigma \vdash \Gamma \Uparrow \forall x.\, A, \Theta}\ y \notin \Sigma$$

$$\frac{\Sigma \vdash (\texttt{copy}\ t\ s) \quad \Xi_1; \Sigma \vdash \Gamma \Downarrow [s/x]A \quad \exists_e(\Xi_0, \Xi_1, t)}{\Xi_0; \Sigma \vdash \Gamma \Downarrow \exists x.\, A}$$

Figure 2.1: LKF^a rules augmented with `copy` clauses

by using skolemization the prover extends the signature during the proof process, and it is no longer the case that they agree. In this section we elaborate on this idea and show how to handle proofs produced by a prover that uses skolemization. We interpret those proofs making use of an extended language into our kernel, which doesn't extend the original signature but employs eigenvariables.

2.1.1 Treating user terms and kernel terms

Reviewing the rules for $\forall$ and $\exists$ in fig. 1.4, we note in particular that the universal rule extends the signature Σ with a new eigenvariable; the result Ξ_1 of the $\forall$ clerk is a certificate constructor that takes an input term, and the new eigenvariable is fed to it in order to continue the proof. Thus, the certificate handles a variable that is bound directly to the reconstructed proof tree by the kernel.

In order to allow for differentiating user terms and kernel terms and translating between them, we extend the LKF^a kernel with a relation that initially puts in correspondence all user terms and their corresponding kernel term. During the proof checking process, the relation is extended with new eigenvariables on the kernel side, and what is known about the user term on the user side. Correspondingly the definition of the clerk for the universal quantifier is updated to also make it possible for the user to communicate information about the term that will reference the generic individual in the proof. Since in the case of skolemization no trace is left in the proof corresponding to these inference steps, typically the proof will omit any evidence concerning this term and the new kernel eigenvariable will be associated with an unknown term. We similarly extend the proof rules where the user provides an instantiation term: the kernel instantiates the formula with the kernel term corresponding to the user-supplied term. The resulting rules are in fig. 2.1.

The relational handling of the correspondence between user and kernel terms can be put to work in the context of logic programming by using copy clauses, as presented in [MN12]: by interpreting the `copy` relation we just sketched as a logic program, we obtain an extensible way to perform efficient substitution on user and kernel term, as well as a handler for reconstructing partially specified terms.

In the implementation, the predicate `copy: user_term → kernel_term → prop` is initialized with a rule for each symbol in the signature: constant symbols need to be copied to a constant, while for symbols with arity greater than one we state that a kernel term is the copy of a user term if all the arguments are recursively copies; we write $\mathcal{C}\Sigma$ for the set of copy clauses generated

from the signature Σ.

Example 1. Assume that the base signature for both the client and the kernel is $\mathscr{L} = \{a/0, f/1, g/2\}$. The copy clauses in $\mathcal{C}\Sigma$ are:

```
copy a a.
copy (f X)   (f U)   :- copy X U.
copy (g X Y) (g U V) :- copy X U, copy Y V.
```

In the simplest case, that we are considering here, user terms and kernel terms have the same type and their distinction only concerns their extension during proof checking (eigenvariables on the kernel side, Skolem terms on the user side). More complex behavior could be captured by means of this approach, such as user terms being less determinate or built out of a different syntax.

2.1.2 Skolemization

Skolemization is a term that is used in reference to a variety of techniques aimed at quantifier elimination, and justified by theorems of Skolem, Löwenheim and Herbrand. The core of the idea is that any quantifier that will be treated by an inference that uses a generic individual can be eliminated, and the variable that it was binding can be replaced by a fresh function symbol applied to all the variables that are visible in that scope. Since the function symbol is fresh, nothing constrains its interpretation, and we get an effective representation of a generic individual in a language that is the original language extended with these new generic individuals. For example when trying to establish that *All philosophers are mortal* we would remove the quantifier, introduce the function symbol c with arity 0 and move on to prove the formula *If c is a philosopher, then c is mortal*.

It is common to call Herbrandization the procedure where only universal quantifiers are removed, and skolemization the procedure where only existential quantifiers are removed. Traditionally one would use Herbrandization when wanting to preserve model-theoretic validity in the transformation, and skolemization when wanting to preserve model-theoretic satisfiability. However, when discussing the matter in a proof theoretic perspective we can see the two as the same procedure, applied to the right hand side or the left hand side of a sequent. We use the term skolemization even though we will work on the right-hand side of sequents, in a single-sided calculus.

In the following, we will use the following two definitions of skolemization (which are standard, see for example [RV01]):

Definition 4 (Skolemization)**.**

- An *outer skolemization step* is a pair of formulas in which
  - the first formula, say, B is such that it contains the subformula $\forall x.\, C$ that is not in the scope of any universal quantifier and which is in the scope of existential quantifiers binding the variables $x_1, \ldots, x_n$ ($n \geq 0$); and

  - the second formula results from picking an n-arity symbol f from Σ_{sk} that does not appear in B and replacing that occurrence of $\forall x.C$ in B with the instance $[f(x_1, \ldots, x_n)/x]C$.

- An *inner skolemization step* is a pair of formulas that is defined analogously with the only difference being that the Skolem term used to instantiate x in C is $f(y_1, \ldots, y_m)$ where $y_1, \ldots, y_m$ are the free variables of the occurrence of $\forall x.\, C$.

- The formula E is the result of performing *outer skolemization* on B if there is a sequence of outer skolemization steps that carries B to E and where E does not contain any strong quantifiers (i.e., universal quantifiers). Similarly, the formula E is the result of performing *inner skolemization* on B if there is a sequence of inner skolemization steps that carries B to E and where E does not contain any strong quantifiers.

Automated theorem provers use skolemization in order to simplify the unification steps in their algorithms: once quantifier alternation is removed, simple first order unification suffices. This is a standard step in theorem provers based on the Davis-Putnam algorithm [DP60] and similar. Here is an example of the definition in action:

Example 2. The formula $\exists x.\, (\neg\, D(x) \vee \forall y.\, D(y))$ can be skolemized as follows.

- Outer: $\exists x.\, (\neg\, D(x) \vee D(f(x)))$
- Inner: $\exists x.\, (\neg\, D(x) \vee D(f))$

The theorem that ensures the applicability of skolemization says:

Theorem 2 (Skolem)**.** *Let B be a first-order formula over the signature Σ_0 and let E be either an inner or outer Skolemization of B. If E is provable then so is B.*

The proof of the theorem for the case of outer skolemization is a standard result in logic, while the case of inner skolemization was proved by Andrews (for this reason it is often referenced as Andrew's skolemization). In both cases, the proof of the theorem relies on a model theoretic argument that uses the axiom of choice: this is a combination that wouldn't leave any hope of getting some understanding of the proof of a formula if we are handed a proof of its skolemization.

If we view skolemization as a naming device as we have just presented, we obtain the following:

Example 1 (continuing)**.** Assume that the client is using $h/1$ as a Skolem function and that the kernel has introduced two eigenvariables x and y and that Σ contains exactly the two associations `(copy (h a) x)` and `(copy (h (f a)) y)`. The λProlog query

$$\mathcal{C}\Sigma, \Sigma \vdash \texttt{(copy (g (h (f a)) (f (h a))) X)}$$

for some logic variable `X` will have a unique solution, namely, the one that binds `X` to `(g y (f x))`. It is this step that performs deskolemization. Note, however, that we do not necessarily assume that deskolemization is determinate. In particular, if the Γ context contained the atoms `(copy (h a) x)`

and `(copy (h a) y)`, then there are two solutions to the query `(copy (g (h a) (f a)) X)`, namely, binding `X` to either `(g x (f a))` or `(g y (f a))`. □

2.1.3 Proofs of skolemized statements

Our interest is in building the proof of a formula when the prover hands us a proof of its skolemization. We will now look into how proofs of skolemized formulas in the *LK* sequent calculus look like, in order to draw some first conclusions. We start by analyzing example 2, and consider proofs of the two skolemized formulas over a signature containing a constant symbol c

Example 3 (Outer skolemization and inner skolemization)**.**
Outer: $\exists x.\,(\neg\, D(x) \vee D(f(x)))$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \neg\, D(c), D(f(c)), \neg\, D(f(c)), D(f(f(c)))}\ \text{init}}{\vdash \neg\, D(c), D(f(c)), \neg\, D(f(c)) \vee D(f(f(c)))}\ \vee}{\vdash \neg\, D(c), D(f(c)), \exists x.\,(\neg\, D(x) \vee D(f(x)))}\ \exists f(c).}{\vdash \neg\, D(c) \vee D(f(c)), \exists x.\,(\neg\, D(x) \vee D(f(x)))}\ \vee}{\vdash \exists x.\,(\neg\, D(x) \vee D(f(x))), \exists x.\,(\neg\, D(x) \vee D(f(x)))}\ \exists c.}{\vdash \exists x.\,(\neg\, D(x) \vee D(f(x)))}\ \mathrm{C}$$

An *LK* proof of the outer skolemized form needs to start with a contraction and uses two witness terms, c and $f(c)$ (where c is the individual we included in the global signature)
Inner: $\exists x.\,(\neg\, D(x) \vee D(f))$

$$\dfrac{\dfrac{\dfrac{}{\vdash \neg\, D(f), D(f)}\ \text{init}}{\vdash \neg\, D(f) \vee D(f)}\ \vee r}{\vdash \exists x.\,(\neg\, D(x) \vee d(f))}\ \exists f.$$

The inner skolemized form, on the other hand, has a simple *LK* proof that provides the witness f for x and does not require a contraction nor an appeal to the postulated individual.

If we try to reconstruct a proof of the original, unskolemized formula $\exists x.\,(\neg\, D(x) \vee \forall y.\, D(y))$ by following the first script we are easily successful: we can start in the same way with C, $\exists c.$ and $\vee$; at this point the sequent will contain the subformula $\forall y.\, D(y)$ to which we can apply a $\forall$ rule that will introduce an eigenvariable. This eigenvariable is what we must use in the next $\exists$ rule, and

the proof is then similarly concluded.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \neg D(c), D(y), \neg D(y), \forall y.\, D(y)}\,\text{init}}{\vdash \neg D(c), D(y), \neg D(y) \vee \forall y.\, D(y)}\,\vee}{\vdash \neg D(c), D(y), \exists x.\, (\neg D(x) \vee \forall y.\, D(y))}\,\exists y.}{\vdash \neg D(c), \forall y.\, D(y), \exists x.\, (\neg D(x) \vee \forall y.\, D(y))}\,\forall}{\vdash \neg D(c) \vee \forall y.\, D(y), \exists x.\, (\neg D(x) \vee \forall y.\, D(y))}\,\vee}{\vdash \exists x.\, (\neg D(x) \vee \forall y.\, D(y)), \exists x.\, (\neg D(x) \vee \forall y.\, D(y))}\,\exists c.}{\vdash \exists x.\, (\neg D(x) \vee \forall y.\, D(y))}\,\mathrm{C}$$

The same cannot be said about the proof script involving inner skolemization. Here the existential rule is used immediately, but the only individual that is available right away in the non-skolemized world is c; when we introduce the eigenvariable corresponding to the universal subformula, we cannot match the two instances to conclude the proof.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{???}{\vdash \neg D(c), D(y)}\,???}{\vdash \neg D(c), \forall y.\, D(y)}\,\forall}{\vdash \neg D(c) \vee \forall y.\, D(y)}\,\vee}{\vdash \exists x.\, (\neg D(x) \vee \forall y.\, D(y))}\,\exists c.$$

Inner skolemization adds a new constant term to the signature instead of a function, and thereby erases all information on dependencies. The constant term can be used outside its scope.

If we look at the proof script top-down, we see that an alternative way to follow this inference would be by relaxing the eigenvariable condition, and starting with the sequent $\vdash \neg D(c), D(c)$ inferring $\vdash \neg d(c), \forall y. D(y)$. Very recently several works by Baaz and others [BL22; AB19] studied a variant of *LK* where the eigenvariable condition is relaxed in a way that allows this inference. The proof scripts translate to the reasoning below, where the new rule is used in the inference from 1 to 2:

1. That Kurt Gödel is Austrian, entails that Kurt Gödel is Austrian
2. Hence, that Kurt Gödel is Austrian entails that everybody is Austrian
3. That is, if Kurt Gödel is Austrian then all the people are Austrian
4. Therefore there exists a person such that if that person is Austrian then all the people are Austrian

The conclusion is indeed the drinker's formula, and the proof corresponds to the one we have for the inner skolemization. In the works of Baaz et al. a translation is provided from this extended calculus into *LK*, together with a proof that there is no elementary function bounding the length of the obtained *LK* proofs in terms of the proofs in the extended calculus. Therefore, there is little hope to be able to directly use proofs of inner skolemizations as certificates. Ralph [Ral20] drew a connection between Baaz's calculus and Deep

Inference, which could provide an alternative framework to understand these proofs.

It is well known that inner skolemization is problematic when it comes to proof reconstruction, and as of present we are not aware of any proposed proof checking system that allows for checking this kind of proof. The case of outer skolemization is usually seen as less problematic; however we should see that it also needs some special care. We start by noticing how in the proof script in example 3 we make the choice of interleaving the two applications of the $\exists$ rule with the $\vee$ rule. While this choice is not relevant in that proof, and we could easily exchange the two applications, this is not the case once we try to de-skolemize the proof: the $\forall$ rule needs to be applied between the two applications of the $\exists$ rule, and it needs to follow the $\vee$ rule; therefore, the $\vee$ rule must be used before the second $\exists$.

To make things clearer, we modify slightly the example and consider the formula $((\forall x.\ \neg P(x)) \wedge \neg q) \vee \exists x.\ (P(x) \vee q)$. Suppose we have proof evidence in the form of an *LK* proof for its outer skolemization $(\neg P(f) \wedge \neg q) \vee \exists x.\ (P(x) \vee q)$, with f a fresh Skolem constant. This formula has the following two proofs:

Example 4 (Different proofs with outer skolemization)**.**

$$\dfrac{\dfrac{\dfrac{}{\vdash \neg P(f), P(f), q}\ \text{init}}{\vdash \neg P(f), \exists x.\ (P(x) \vee q)}\ \exists(f), \vee \qquad \dfrac{\dfrac{}{\vdash \neg q, P(f), q}\ \text{init}}{\vdash \neg q, \exists x.\ (P(x) \vee q)}\ \exists(f), \vee}{\vdash (\neg P(f) \wedge \neg q) \vee \exists x.\ (P(x) \vee q)}\ \vee, \wedge$$

$$\dfrac{\dfrac{\dfrac{}{\vdash \neg P(f), P(f), q}\ \text{init} \qquad \dfrac{}{\vdash \neg q, P(f), q}\ \text{init}}{\vdash \neg P(f) \wedge \neg q, P(f) \vee q}\ \wedge, \vee}{\vdash (\neg P(f) \wedge \neg q) \vee \exists x.\ (P(x) \vee q)}\ \vee, \exists(f)$$

The first, longer, proof contains all the information we need to build a proof of the original formula. When reconstructing the proof, we begin by using the $\vee$ and $\wedge$ rules, at which point we have two branches where the sequents we are proving contain the subformula $\forall x.\ \neg P(x)$. An application of the $\forall$ rule in both branches will add to the contexts the eigenvariables we need to subsequently use the $\exists$ rule and successfully conclude with the $\vee$ and init rules.

The second proof looks problematic, in a way that reminds us of the previous problem with inner skolemization. However now we are facing a formula where inner and outer skolemization coincide, so the phenomena is slightly different. Indeed, it was already noted in [BHW12] that outer skolemization allows for proofs that can be exponentially shorter than the proofs of the original statement; in particular, the example we chose comes directly from that paper, and is part of a class of formula whose shortest proofs are proved to be exponentially longer than the proofs of their skolemization.

By analyzing these proofs through polarization and focused proof systems, we can uncover more of what's going on. We start by noting that in all the examples so far we chose to use the $\vee$ rule of *LK* that corresponds to the $\bar{\vee}$ rule of *LKF*; in a focused proof the $\bar{\vee}$ connective must be eagerly decomposed during the asynchronous phase and, if under focus, will stop the synchronous phase. The proof we considered in example 3 contains precisely this interruption of

the synchronous phase, where the disjunction is decomposed between the two existential instantiations; we had indeed noticed that this had to be the case if we wanted to understand this as a proof of the original formula. In a similar spirit, the difference between the two proofs in example 4 lies in whether we are using the positive or negative conjunction; only the negative can be used to get insight on a proof of the original formula, and the positive on the other side corresponds to the shorter proof (as is usually the case).

Consider LKF^a with the new rules in fig. 2.1. Assume an FPC specification for a format that uses skolemization. We add to it the following specification for the $\forall$ clerk (which is obviously not specified if skolemization is assumed):

```
allC Cert Cert _T.
```

That is, no user term is provided as generic name, and a fresh logic meta-variable is handed to the kernel instead. We have the following theorem, which is a generalization of Theorem 6.12 in [Mil87].

Theorem 3. *Let F be a formula and F' be the outer skolemization of F. Let $\hat{F}'$ and $\hat{F}$ be their purely negative polarizations, and let Ξ be a certificate for $\hat{F}'$. Then, Ξ is a certificate for $\hat{F}$.*

Consider for example the first proof of example 3. Its initial set of copy clauses is $\mathcal{C}\Sigma = \{\texttt{copy c c}\}$. Its reconstruction in LKF^a looks like this:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{copy (f c) y, copy Y z}, \mathcal{C}\Sigma \vdash \exists x.\, (\neg\, D(x) \,\bar{\vee}\, \forall y.\, D(y)), \neg\, D(c), D(y), D(z) \Downarrow \neg\, D(y)}}{\texttt{copy (f c) y, copy Y z}, \mathcal{C}\Sigma \vdash \exists x.\, (\neg\, D(x) \,\bar{\vee}\, \forall y.\, D(y)), \neg\, D(c), D(y), \neg\, D(y), D(z) \Uparrow \cdot}}{\texttt{copy (f c) y, copy Y z}, \mathcal{C}\Sigma \vdash \exists x.\, (\neg\, D(x) \,\bar{\vee}\, \forall y.\, D(y)), \neg\, D(c), D(y), \neg\, D(y) \Uparrow D(z)}\,\forall}{\texttt{copy (f c) y}, \mathcal{C}\Sigma \vdash \exists x.\, (\neg\, D(x) \,\bar{\vee}\, \forall y.\, D(y)), \neg\, D(c), D(y), \neg\, D(y) \Uparrow \forall y.\, D(y)}}{\texttt{copy (f c) y}, \mathcal{C}\Sigma \vdash \exists x.\, (\neg\, D(x) \,\bar{\vee}\, \forall y.\, D(y)), \neg\, D(c), D(y) \Downarrow \neg\, D(y), \forall y.\, D(y)}\,\text{rel, store}}{\texttt{copy X y}, \mathcal{C}\Sigma \vdash \exists x.\, (\neg\, D(x) \,\bar{\vee}\, \forall y.\, D(y)), \neg\, D(c), D(y) \Downarrow \exists x.\, (\neg\, D(x) \,\bar{\vee}\, \forall y.\, D(y))}\,\exists\,\texttt{X},\bar{\vee}}{\texttt{copy X y}, \mathcal{C}\Sigma \vdash \exists x.\, (\neg\, D(x) \,\bar{\vee}\, \forall y.\, D(y)), \neg\, D(c), D(y) \Uparrow \cdot}\,\text{decide}}{\texttt{copy X y}, \mathcal{C}\Sigma \vdash \exists x.\, (\neg\, D(x) \,\bar{\vee}\, \forall y.\, D(y)) \Uparrow \neg\, D(c), D(y)}\,\text{store}}{\mathcal{C}\Sigma \vdash \exists x.\, (\neg\, D(x) \,\bar{\vee}\, \forall y.\, D(y)) \Uparrow \neg\, D(c), \forall y.\, D(y)}\,\forall}{\mathcal{C}\Sigma \vdash \exists x.\, (\neg\, D(x) \,\bar{\vee}\, \forall y.\, D(y)) \Uparrow \neg\, D(c) \,\bar{\vee}\, \forall y.\, D(y)}\,\vee}{\mathcal{C}\Sigma \vdash \exists x.\, (\neg\, D(x) \,\bar{\vee}\, \forall y.\, D(y)) \Downarrow \exists x.\, (\neg\, D(x) \,\bar{\vee}\, D(f(x)))}\,\text{release}}{\mathcal{C}\Sigma \vdash \exists x.\, (\neg\, D(x) \,\bar{\vee}\, \forall y.\, D(y)) \Downarrow \exists x.\, (\neg\, D(x) \,\bar{\vee}\, D(f(x)))}\,\exists c.}{\mathcal{C}\Sigma \vdash \exists x.\, (\neg\, D(x) \,\bar{\vee}\, \forall y.\, D(y)) \Uparrow \cdot}\,\text{decide}}{\mathcal{C}\Sigma \vdash \cdot \Uparrow \exists x.\, (\neg\, D(x) \,\bar{\vee}\, \forall y.\, D(y))}\,\text{store}$$

The new eigenvariables are associated with non-determinate user names. Upon attempting to instantiate an existential with the user-provided Skolem term, that is unknown to the kernel, the new name $f(c)$ is unified with the available logic variable `X`, and the user term is translated to the kernel term y. A new logic variable is created for the second eigenvariable instantiation, but that turns out to be not relevant for the proof. The builtin control of eigenvariables done by λProlog ensures that no free occurrence condition is violated.

While on a first sight restricting to negative polarities might seem a big restriction, it is not new. We hope that framing it in the context of focused proof theory will help further developments. Among the few approaches to

proof reconstruction and certification that treat skolemization by deskolemizing (as opposed to including skolemization in the certifier's metatheory by adding axioms or rules), most are incapable of treating proofs involving the positive sequent calculus rules, although the different formalisms don't make this limitation easy to appreciate ([FK16; EH21]). The most complete treatment of proofs with skolemization is found in the GAPT tool [Ebn+16], where the algorithms from [BHW12] and subsequent articles are used and proof are internally represented using a structure similar to the Expansion Trees introduced in [Mil87]; this means that proofs of skolemized statements involving positive rules are converted to Expansion Trees before being available to the user, and deskolemization is only performed on Expansion Trees.

Indeed, most proof-theoretic results on deskolemization ultimately rely on [Mil87], where Expansion Trees and their deskolemization are introduced. In the next section we will look at a concrete example of handling proof evidence in the form of Expansion Trees.

2.1.4 Proof evidence containing Skolem terms

We describe now the concrete proof format of Expansion Trees. This allows us to show another angle of the discussion about user terms, since expansion trees internalize the dependency conditions by means of the notion of *select variable*. Even when not talking about proofs containing skolemization, we will need to leverage the extended rules we just introduce in order to certify proofs in the form of expansion trees. We conclude the section with the treatment of skolemized expansion trees.

Definition 5 (Expansion trees)**.**

- A literal or logical constant is an expansion tree for itself.
- If Q_1 and Q_2 are expansion trees of A_1 and A_2, then $(eOr\ Q1\ Q2)$ and $(eAnd\ Q1\ Q2)$ are expansion trees for $A_1 \vee A_2$ and $A_1 \wedge A_2$ respectively
- If u is a variable (called a *select variable*) and Q is an expansion tree of $[u/x]A$, then $(eAll\ u\ Q)$ is an expansion tree for $\forall x.\ A$.
- If $t_1, \ldots, t_n$ is a list of *expansion* terms and if Q_i is an expansion tree for $[t_i/x]A$ (for $i \in 1..n$), then

  $$(eSome\ [(t_1, Q_1), \ldots, (t_n, Q_n)])$$

  is an expansion tree for $\exists x.\ A$. □

Expansion terms can certainly contain select variables. The formal, standalone definition of expansion trees requires additional correctness conditions to be assumed (that a certain propositional formula derived from the expansion tree is a tautology and that a certain relationship on select variables is acyclic) but these conditions are not needed here since they will be implicitly handled by the proof checking kernel itself.

The datatype for expansion trees can be formalized by the λProlog signature in Figure 2.2 and the more general notion of certificate based on expansion trees is given in Figure 2.3. There, proof certificates (terms of type `cert`) are

```
kind et                type.
type eTrue, eFalse     et.
type eLit              et.
type eAnd, eOr         et → et → et.
type eAll              i  → et → et.
type eSome             list (pair i et) → et.
```

Figure 2.2: The datatype for expansion trees. The `kind` declaration introduces a primitive type `et` and the `type` declarations introduce constructors for this primitive type.

```
kind address       type.
type root          address.
type lf, rg, dn    address → address.
type idx           address → index.
typeabbrev context list (pair address et).
type astate        context → context → cert.
type dstate        context → context → cert.
type sstate        context → pair address et → cert.
```

Figure 2.3: Certificate constructors for expansion trees. The primitive types `index` and `cert` are declared as part of the kernel. The type `address` is introduced for this particular FPC.

```
orC    (astate Left ((pr Add (eOr E1 E2))::Qs))
       (astate Left ((pr (lf Add) E1)::
                     (pr (rg Add) E2)::Qs)).
andC   (astate Left ((pr Add (eAnd E1 E2))::Qs))
       (astate Left ((pr (lf Add) E1)::Qs))
       (astate Left ((pr (rg Add) E2)::Qs)).
someE  (sstate Left
               (pr Add (eSome ((pr Term ET)::nil)))
       (dstate Left ((pr (dn Add) ET)::nil)) Term.
allCx  (eAll Term Cert) Cert Term.
```

Figure 2.4: Some of the clerks and experts for expansion trees. All of these λProlog clauses are simply atomic formulas that perform some pattern matching and simple transformations on certificates.

built from three constructors: `astate` is consumed during the asynchronous phase and records two contexts representing some information about the storage zone Γ and the asynchronous zone Θ; `sstate` is consumed during the synchronous phase and records the storage and the formula under focus; and `dstate` is used to break focusing on adjacent existential introductions. Formulas are paired in the certificate with the expansion trees to which they are associated. Addresses are essentially paths through the proposed theorem: they are used to uniquely describe subformulas. For example, such addresses are used to link stored formulas (note that indexes contain addresses) with expansion trees sorted within certificate terms.

The main clerks and experts are specified in Figure 2.4. Since connectives are polarized negatively, most of the work is carried out by clerks that simply consume expansion trees and reorganize internal components of certificates. When proof checking encounters a strong quantifier, the expansion-tree-cum-certificate contains the select variable associated to it: we then use the `allCx` to instruct the kernel to create a new eigenvariable and associate the client's select variable as a name for that eigenvariable. When proof checking meets an existential node, together with the list of terms by which the existential should be instantiated, we can simply communicate one of the client's expansion terms to the kernel which then proceed to translate it to a kernel term. Note that, in the code, we make the assumption that only one term is present in the list: this is because contraction is handled by the expert for the decide rule (not shown here).

Note that the mechanism we have described as deskolemization is exactly the same mechanism that can replace variable names (select variables) with eigenvariables. Note also that if the expansion tree that is being checked uses a select variable more than once to name different eigenvariables, the checker will need to deal with nondeterminism in sorting out which assignment of select variable to eigenvariable leads to a proper proof. Similar to the comment in Example 1, such non-unique naming is not a soundness problem: it can, however, raise the cost and complexity of proof checking.

We now turn our attention to the setting where the client has an expansion tree relative to a skolemized formula, but we would like to use it as proof evidence of the original, unskolemized formula. As it turns out, there is not much which is left to do here. Since there are no strong quantifiers left in the skolemized formula, the expansion tree will not contain any select variables (nor any Skolem terms). Accordingly, we modify the `allCx` to be just

```
allCx Cert Cert T.
```

Thus, when the checker finds a strong quantifier it will simply associate to the newly created eigenvariable a logic variable (here, `T`) as the name for it. This variable will ultimately be instantiated to be an actual Skolem term (through the interaction of proof checking and unification).

2.2 Communicating indexes

A central part in the theory of focused proof systems is played by the store, decide, release and initial rules. Their read back in the original Gentzen calculus are the contraction and initial rules, but when we move to focused proof theory they gain a finer meaning: here, they act as delimiters of the different phases in a proof and provide control to its geometry; moreover, by managing when to contract and which formulas to put under focus, they give rise to a notion of full-fledged storage area. The data structure these rules rely on is that of an index.

Several preprocessing steps employed in automated theorem provers involve dealing with aspects of a formula that are related to its geometry and other morphological facets. In these steps we typically see the goal formula being transformed to a different one so that the new one complies to a predetermined shape, in order to permit the application of automation techniques.

A prominent case is the transformation to Clause Normal Form, where a formula is transformed into one that is structured as a conjunction of disjunctions, in such a way that satisfiability is preserved.

2.2.1 Tseitin transformation

The most direct transformation to conjunctive normal form relies on using the distributivity property of disjunction over conjunction; however this generates formulas whose size can be exponential in the number of disjunctions: in the case of the formula $(p \wedge q) \vee r$ we obtain $(p \vee r) \wedge (q \vee r)$ with one application of distributivity, and r has been duplicated.

This is why several optimization strategies exist that provide shorter formulas in CNF. One chief example of such transformations is Tseitin's transformation, which manages to produce formulas whose size is linear in the size of the input formula and are equisatisfiable. The idea of the transformation is to introduce new propositional constants in correspondence with every subformula of the input formula, and to build the output CNF formula so that it can be treated as a set of clauses stating that each one of the newly generated propositional constants must be logically equivalent to the subformula it corresponds to. It is also called *Definitional CNF* (see [Har09]), since it involves introducing new symbols together with their definition. Formally, given a formula F, let $SF(F)$ be the set of subformulas of F, excluding the literals. The transformation proceeds as follows:

Definition 6.

- $T(F) := a_F \wedge \bigwedge_{G \in SF(F)} t(G)$
- $t(\neg G) := (\neg a_{\neg G} \vee \neg a_G) \wedge (a_{\neg G} \vee a_G)$
- $t(G_1 \wedge G_2) := (\neg a_{G_1 \wedge G_2} \vee a_{G_1}) \wedge (\neg a_{G_1 \wedge G_2} \vee a_{G_2}) \wedge (\neg a_{G_1} \vee \neg a_{G_2} \vee a_{G_1 \wedge G_2})$
- $t(G_1 \vee G_2) := (\neg a_{G_1} \vee a_{G_1 \vee G_2}) \wedge (\neg a_{G_2} \vee a_{G_1 \vee G_2}) \wedge (\neg a_{G_1 \vee G_2} \vee a_{G_1} \vee a_{G_2})$

Since we are working in negation normal form, we don't need to consider the clause for negation. Moreover, since the soundness of our system is always guaranteed by the soundness of LKF^a, we only need to consider the first half of the clauses. The reduced definition consists of only the two following cases:

- $t(G_1 \wedge G_2) := (\neg a_{G_1 \wedge G_2} \vee a_{G_1}) \wedge (\neg a_{G_1 \wedge G_2} \vee a_{G_2})$
- $t(G_1 \vee G_2) := (\neg a_{G_1 \vee G_2} \vee a_{G_1} \vee a_{G_2})$

Consider the following example:

Example 5. Say we want to prove $F := G_1 \vee G_2$, where $G_1 := (p \wedge \neg q) \vee (q \wedge \neg p)$ and $G_2 := (p \wedge \neg r) \vee ((\neg q \vee r) \wedge (\neg p \vee q))$.
We first negate it: $F^\perp := {G_1}^\perp \wedge {G_2}^\perp$, $G_1 := (\neg p \vee q) \wedge (\neg q \vee p)$ and $G_2 := (\neg p \vee r) \wedge ((q \wedge \neg r) \vee (p \wedge \neg q))$.
Then its Tseitin transformation is $T((F)^\perp) =$

$$
\begin{aligned}
&a_F \wedge (\neg a_{F^\perp} \vee a_{{G_1}^\perp}) \wedge (\neg a_{F^\perp} \vee a_{{G_2}^\perp}) \wedge (\neg a_{{G_1}^\perp} \vee \neg p \vee q) \wedge \\
&(\neg a_{{G_1}^\perp} \vee \neg q \vee p) \wedge (\neg a_{{G_2}^\perp} \vee \neg p \vee r) \wedge (\neg a_{{G_2}^\perp} \vee a_{(q \wedge \neg r)} \vee a_{(p \wedge \neg q)}) \wedge \\
&(\neg a_{q \wedge \neg r} \vee q) \wedge (\neg a_{q \wedge \neg r} \vee \neg r) \wedge (\neg a_{p \wedge \neg q} \vee p) \wedge (\neg a_{p \wedge \neg q} \vee \neg q)
\end{aligned}
$$

The formula obtained through this transformation has a size that is linear in the size of the input formula, and is equisatisfiable with it. The prover will try to refute $T((F)^{\perp})$. Such a refutation, if found, guarantees that $(F)^{\perp}$ can be refuted as well, and constitutes therefore the proof evidence for the provability of F.

The justification for this method is usually presented by appealing to model theoretic reasoning: if a model exists for the translated formula, then we can read back a model for the original formula by looking at the assignments it makes to the propositional constants that appear in the original formula.

By observing this reasoning, however, it stands out that the newly introduced constants do not bear any real logical meaning, and are rather just used as shorthand for the denoted subformula: this is highlighted by the fact that we assign and read back their valuations by making a simple meta-level observation that these should correspond to the valuation of the subformula they denote. As such, they seem misplaced as logical constructs, and should rather be explained as a metalogical device.

Just as in the previous section, we consider also the dualized transformation and obtain a validity-preserving transformation. Additionally, we note that once we are working in negation normal form we can use a reduced version of the transformation where no clause for the negation connective is present, and as a consequence only one direction of the implication is left. We get the following definition:

Definition 7 (Valitity-preserving Tseitin transformation)**.**

- $T(F) := a_F \bar{\vee} \bigvee_{G \in SF(F)} t(G)$
- $t(G_1 \vee G_2) := (\neg\, a_{G_1 \vee G_2} \wedge a_{G_1}) \vee (\neg\, a_{G_1 \vee G_2} \wedge a_{G_2})$
- $t(G_1 \wedge G_2) := (\neg\, a_{G_1 \wedge G_2} \wedge a_{G_1} \wedge a_{G_2})$

2.2.2 The structure of indexes

Our choice of using the set of subformulas itself as the set of indexes for the Tseitin constants makes this remark very evident. It is however something that is usually neglected in most presentations of the Tseitin transformation (such as [Har09]), where the set of indexes is just that of natural numbers with the effect of blurring the information that is contained in the proof.

Let's go back to the initial observation, and let's place ourselves in front of the task of certifying a proof evidence where the prover employed the Tseitin transformation. More specifically, we are looking for a way to understand the inferences involving Tseitin clauses as inferences made on the original formula, within the original language. We argue that if the meaning of a Tseitin constant is entirely determined by the subformula that was used to build it (and is, in many cases, used to name it or even somehow index it), then they should be understood and treated as indexes relative to that subformula, and we will explain this interpretation in the next section.

The indexing mechanism of *LKF*^{a} presented in subsection 1.2.2 is governed by choices in the polarization of the formula. In particular, we know that an index is built for everything positive that is encountered during an asynchronous phase. In order to be able to carry out our interpretation of proofs of Tseitin

transformations, we need to start by describing a suitable polarization that imposes a corresponding indexing strategy. We have argued that the Tseitin transformation amounts to imposing that every subformula be indexed; for a start, then, we need every subformula to be positive. Note that this is different from saying that every *connective* should be positive, as this would also impose a proving strategy. The way to proceed is thus to introduce a positive delay before every negative subformula. This is not enough, since sequences of nested positive connectives are treated as composants of a same indexed piece of information and subsequently decomposed during a single synchronous phase, without introducing further indexes. What the Tseitin indexing does is instead an indexing of each and every distinct subformula. In order to force the introduction of a new index, we need to immediately stop the synchronous phase after a connective has been treated, so that a new index is created for the uncovered subformulas. This is the dual of what we have just discussed, since we need to have negative subformulas below every positive connective, and we obtain this by including a negative delay before positive subformulas.

Definition 8 (Tseitin polarization)**.** The Tseitin polarization $T'(F)$ of a polarized formula F is defined by induction on F:

- $T'(G_1 \mathbin{\bar{\wedge}} G_2) = \dagger\bar{\partial}(T'(G_1) \mathbin{\bar{\wedge}} T'(G_2))$
- $T'(G_1 \mathbin{\bar{\vee}} G_2) = \dagger\bar{\partial}(T'(G_1) \mathbin{\bar{\vee}} T'(G_2))$
- $T'(G_1 \mathbin{\dagger\bar{\wedge}} G_2) = \bar{\partial}(T'(G_1) \mathbin{\dagger\bar{\wedge}} T'(G_2))$
- $T'(G_1 \mathbin{\dagger\bar{\vee}} G_2) = \bar{\partial}(T'(G_1) \mathbin{\dagger\bar{\vee}} T'(G_2))$
- $T'(p) = p$ for every literal.

Example 6. Consider the formula we wanted to prove in example 5. Start by choosing a polarization of F; this choice has no impact right now, so we will start by choosing a fully negative polarization out of simplicity. Name $\hat{F}$ the purely negative polarization of F, its focused Tseitin transformation is

$$T'(\hat{F}) := \dagger\bar{\partial}(\dagger\bar{\partial}(\dagger\bar{\partial}(p \mathbin{\bar{\wedge}} \neg q) \mathbin{\bar{\vee}} \dagger\bar{\partial}(q \mathbin{\bar{\wedge}} \neg p)) \mathbin{\bar{\vee}} \dagger\bar{\partial}(\dagger\bar{\partial}(p \mathbin{\bar{\wedge}} \neg r) \mathbin{\bar{\vee}} \dagger\bar{\partial}(\dagger\bar{\partial}(\neg q \mathbin{\bar{\vee}} r) \mathbin{\bar{\wedge}} \dagger\bar{\partial}(\neg p \mathbin{\bar{\vee}} q))))$$

An *LKF*^{a} proof of $T'(\hat{F}')$, as shown in fig. 2.5, has a very constrained structure in its lowermost part: as a first inference we are forced to immediately store the delayed main formula, stop the asynchronous phase and then resume it and strip off the positive delay operator. Subsequently, since we chose the negative polarization of connectives, we must resume the asynchronous phase and consume the connective that has just been unveiled; the sequent we have just unveiled has the same structure as the initial sequent, where the formulas in the focus area are surrounded by the positive delay operator introduced by T'.

The certificate that we elaborate must begin with the instructions on how to build the indexes for all the subformulas. Since we imposed delays (and thus indexing) on all subformulas and since all connectives are negatively polarized, the only way for the certificate to contain information for concluding a proof is by presenting the index for a propositional constant once its complementary is under focus.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Xi_6 \vdash i{:}\ddagger\eth(F), j{:}T'(\hat{G}[1]), k{:}T'(\hat{G}[2]) \Uparrow \cdot \quad \mathsf{store}_c(\Xi_5, \Xi_6, k)}{\Xi_5 \vdash i{:}\ddagger\eth(F), j{:}T'(\hat{G}[1]) \Uparrow T'(\hat{G}[2]) \quad \mathsf{store}_c(\Xi_4, \Xi_5, j)}}{\Xi_4 \vdash i{:}T'(\hat{F}) \Uparrow T'(\hat{G}[1]), T'(\hat{G}[2]) \quad \vee_c(\Xi_3, \Xi_4)}}{\Xi_3 \vdash i{:}T'(\hat{F}) \Uparrow T'(\hat{G}[1]) \mathbin{\bar{\vee}} T'(\hat{G}[2]) \quad \mathsf{release}_e(\Xi_2, \Xi_3)}}{\Xi_2 \vdash i{:}T'(\hat{F}) \Downarrow T'(\hat{G}[1]) \mathbin{\bar{\vee}} T'(\hat{G}[2]) \quad \mathsf{delay}}\ \mathsf{release}}{\Xi_2 \vdash i{:}T'(\hat{F}) \Downarrow \ddagger\eth(T'(\hat{G}[1]) \mathbin{\bar{\vee}} T'(\hat{G}[2])) \quad \mathsf{decide}_e(\Xi_1, \Xi_2, i)}\ \mathsf{decide}}{\Xi_1 \vdash i{:}T'(\hat{F}) \Uparrow \cdot \quad \mathsf{store}_c(\Xi_0, \Xi_1, i)}\ \mathsf{store}}{\Xi_0 \vdash \cdot \Uparrow T'(\hat{F})}$$

Figure 2.5: An LKF^a proof of the focused Tseitin transform for a polarization of the formula in example 6

If the polarization we choose is richer, the certificate can contain more information. However, in order for us to consume the information relative to a certain subformula we will need to be steered through the delays that enclose it, and therefore build the indexes for the subformulas we meet. This is how the Tseitin transformation is internalized: just like here, in a proof of the usual Tseitin transformation of a formula we need to go through a series of expansion of the definitional clauses before we can apply an insight relative to subformulas of the original formula.

2.2.3 Proof evidence for Tseitin transformation

In this section we describe how to elaborate a proof of the Tseitin transformation of a formula as a certificate for the original formula, where no Tseitin constants will appear in the formula and in the proof. Contrarily to the case of skolemization in section 2.1, here we are able to interpret proofs involving any polarization. Therefore the strategy we should use is more akin to the one of the hosted calculi in subsection 1.2.4. We consider the case where the client handles us:

- An FPC specification for a format of proofs of formulas obtained through the Tseitin transformation, composed of clerks and experts that we mark with $\cdot^T$.
- The methodology by which the transformation was built, comprising the names of the newly introduced constants, the subformula they were built from and the indexes of their definitional clauses inside a context Θ.
- A polarization for the clauses in Θ.
- The proof certificate and the Tseitin transformed sequent that it constitutes evidence for, of the form $\Xi^T \vdash^T \Theta \Uparrow a_F$.

There are some immediate constraints for the polarization of the clauses. First of all, the disjunctions linking them must be negative: this means that all clauses should be split up as soon as possible and stored at their own index; in this way we can assume that Θ already contains the indexed clauses.

A positive polarization could also have worked, as it would have signified a more bureaucratic approach where every time one of the definitional clauses is used its location inside the big disjunction is provided. More importantly, the conjunction between the literal and the body in each of the clauses must be positive polarity, since it asks that Tseitin clauses are indeed indexed. Our aim is to write a collection of clerks and experts that can interpret these as a proof of the sequent $\Xi^T \vdash^I \cdot \Uparrow F$, where $\vdash^I$ is provability in *LKF*^{a} with the interpreter's clerks and experts.

We start by inspecting how the proof looks like in *LKF*^{a} augmented with the $\cdot^T$ clerks and experts. Initially only the a_F literal is present, and since a_F does not appear in Θ the proof will store it and call the decide_e^T expert to extract the index on one of the clauses in Θ.

$$\dfrac{\dfrac{\dfrac{\dfrac{\Pi_1}{\Xi_4^T \vdash \Theta, \mathtt{t}{:}a_F \Downarrow \neg a_i} \quad \dfrac{\Pi_2}{\Xi_5^T \vdash \Theta, \mathtt{t}{:}a_F \Downarrow \varphi} \quad \dagger_e^T\,(\Xi_3^T, \Xi_4^T, \Xi_5^T)}{\Xi_3^T \vdash \Theta, \mathtt{t}{:}a_F \Downarrow \neg a_i \dagger \varphi \quad \text{decide}_e^T(\Xi_2^T, \Xi_3^T, i)}}{\Xi_2^T \vdash \Theta \mathtt{t}{:}a_F \Uparrow \quad \text{store}_c^T(\Xi^T, \Xi_2^T, \mathtt{t})}}{\Xi^T \vdash \Theta \Uparrow a_F}$$

If the proof contains a decide on the definitional axiom for the i-th Tseitin constant, its body φ will be uncovered. The body can be either a single logical (non-Tseitin) literal, a single Tseitin literal, or the conjunction of two of these. In any case, the right-hand premise of the $\dagger$ inference doesn't contain negated Tseitin literals, and the left-hand premise contains one non-negated Tseitin literal. The right branch of the proof will keep extending without negated Tseitin literals, and contains the logical part of the proof. This does not make the left branch less interesting, although its shape greatly depends on the polarity assigned to the Tseitin literals. If $\neg a_i$ is positive, then the branch must be concluded with init; since a_F is the only available non-negated literal, we conclude that if all the literals are negatively polarized we forcefully decided on the definitional clause for F. On the other side, if there are positively polarized literals we are free to chose one, regardless of which subformula location it denotes, and this might lead to much shorter proofs.

This proof clearly resembles the proof in fig. 2.5 for what concerns the index handling part; in particular, deciding on a Tseitin clause for which the Tseitin literal is negative is in direct correspondence with a decide on the denoted subformula. In order to treat the case of positive Tseitin literals, we allow cuts in the reconstructed proof. The code for the interpreter is presented in fig. 2.6, where we implement the clerks and experts for the interpreter in terms of any FPC specification that uses the Tseitin transformation; the clerks and experts for the user FPC are marked with an additional T. The two insights we illustrated are contained in the second clause for $decide_e$ and in the one for cut_e. In the former, we recognize that we should decide on a subformula when the left branch side of a decide immediately ends with initial. In the latter, we recognize that we should build a lemma if we find a release in the left branch.

We now prove the completeness of the interpreter. Let F be a formula, Θ the list of indexed, polarized Tseitin definitional clauses for F, Ω (for *open* context) a list of indexed Tseitin literals, Δ (for *delayed* context) a list of indexed negated Tseitin literals, and Λ (for *logical* context) a list of indexed logical lit-

```
accumulate mimic-lkf-fpc.

release_ke C1 C2 :-
  release_tke C1 C2.

decide_ke C C3 J :-
  decide_tke C C1 _I,
  andPos_tke C1 C2 C3,
  initial_tke C2 J,
  J = (idx _).

eta_initial C I :-
  initial_tke C I.

cut_ke C C2 (litcert C3) F'
    :-
  decide_tke C C1 I,
  andPos_tke C1 C2 C3,
  release_tke C2 _C4,
  mapsto I F, polarize_res F
      F'.

decide_ke C1 C2 I :-
  decide_tke C1 C2 I.

release_ke (litcert C)
    (litcert C).
store_kc (litcert C) (litidx
    C I) I.
decide_ke (litidx C I) C I.

store_kc C1 C2 I :-
  store_tkc C1 C2 I.

initial_ke C I :-
  initial_tke C I.

release_ke C (mimic I) :-
  initial_tke C I.

orNeg_kc C1 C2 :-
  orNeg_tkc C1 C2.

orPos_ke C C _Choice.
```

Figure 2.6: A hosted interpreter for Tseitin proofs

erals appearing in F. Moreover, let Ω' be the list obtained from Ω by replacing the literals with the subformula of F they denote and including F, and let Δ' be obtained by replacing the literals in Δ with the negations of the subformulas of F they denote (in both Ω' and Δ' we leave the same indexes as in Ω and Δ).

Theorem 4. *Let $\hat{F}$ be the Tseitin polarization of F, let a be either a (possibly negated) Tseitin literal or a logical literal, and let D be a if a is logical, or the subformula of $\hat{F}$ denoted by a if it is a Tseitin literal, or its negation if a is a negated Tseitin literal.*

If the sequent $\Xi^T \vdash^T \Theta, \Omega, \Delta, \Lambda \Uparrow a$ is provable, then $\Xi^T \vdash^I \Omega', \Delta', \Lambda \Uparrow D$ is provable. Moreover, if the sequent $\Xi^T \vdash^T \Theta, \Omega, \Delta, \Lambda \Downarrow a$ is provable, then $\Xi^T \vdash^I \Omega', \Delta', \Lambda \Downarrow D$ is provable.

Proof. We proceed by induction on the derivation and start by proving the two base cases

i If $\Xi^T \vdash^T \Theta, \Omega, \Delta, \Lambda \Downarrow a$ is proved with an initial, then $\mathrm{init}^T_e(\Xi^T, i)$ holds. If a is a logical literal, $i : \neg a \in \Lambda$ and the proof of $\Xi^T \vdash^I \Omega', \Delta', \Lambda \Downarrow a$ can be concluded with init on the same index. If a is a possibly negated Tseitin literal, then its complement is in $\Omega \cup \Delta$; therefore, the negation of D is in $\Omega' \cup \Delta'$ (including the case where the literal is $\neg a_F$, since always include F in Ω'), and the proof of $\Xi^T \vdash^I \Omega', \Delta', \Lambda \Downarrow D$ can be concluded by performing a release and then using the mimic FPC from [CMR17] on i.

ii The base case for $\Xi^T \vdash^T \Theta, \Omega, \Delta, \Lambda \Uparrow a$ is store followed by decide and then by initial. This means that we are deciding on a literal; in the interpreted

proof we store D at the same index as a, and use decide and init on the same indexes. The arising cases are treated in the same way as the previous point. This includes the case of a_F, the only literal in Θ, which requires $\neg\, a_F \in \Omega \cup \Delta$.

The inductive case for $\Xi^T \vdash^T \Theta, \Omega, \Delta, \Lambda \Downarrow a$ is quickly dealt with. If a is positive, then the only applicable rule is init and we conclude as in (i). If a is negative, the only possible rule is release; since we added negative delays to every subformula in the Tseitin polarization, the proof of $\Xi^T \vdash^I \Omega', \Delta', \Lambda \Downarrow D$ must start with a release as well. Then, we can apply the $\Uparrow$ inductive hypothesis.

We move to the inductive case for $\Xi^T \vdash^T \Theta, \Omega, \Delta, \Lambda \Uparrow a$. The bottom inferences must be store and decide, and thus it must be the case that $\mathsf{store}_e^T(\Xi^T, \Xi_1^T, k)$ and $\mathsf{decide}_e^T(\Xi^T, \Xi_1^T, i)$ for indexes k, i. If the formula stored at i is a literal, we can conclude with the $\Downarrow$ inductive hypothesis.

We are left with the case where we decide on one of the definitional clauses from Θ. Let a_G be the literal in the definitional clause, G be the subformula it denotes, and φ the body of the definitional clause. The next rule has to be $\dagger\wedge$, and thus we have two subproofs:

$$\dfrac{\dfrac{\Pi_1^T}{\Xi_2^T \vdash \Theta, \Omega, \Delta, \Lambda \Downarrow \neg\, a_G} \quad \dfrac{\Pi_2^T}{\Xi_3^T \vdash \Theta, \Omega, \Delta, \Lambda \Downarrow \varphi} \quad \dagger\wedge_e^T (\Xi_1^T, \Xi_2^T, \Xi_3^T)}{\Xi_1^T \vdash \Theta, \Omega, \Delta, \Lambda \Downarrow \neg\, a_G \mathbin{\dagger\wedge} \varphi}$$

We start by noting that from Π_2^T we can obtain a proof of $\Xi_3^T \vdash \Theta, \Omega, \Delta, \Lambda \Downarrow G$, and we do this based on the shape of G. If $G = G_1 \mathbin{\dagger\wedge} G_2$ then also $\varphi = a_1 \mathbin{\dagger\wedge} a_2$, where the two literals denote the two subformulas, so the Tseitin proof must continue the synchronous phase with the $\dagger\wedge$ rule and we can apply the $\Downarrow$ inductive hypothesis to the premises. If $G = G_1 \mathbin{\dagger\vee} G_2$ then φ is the literal denoting one of the two disjuncts, depending on which definitional clause was selected; the interpreter will extract the according left/right bit for the $\dagger\vee$ rule, and then the $\Downarrow$ inductive hypothesis can be used.

We can now use this proof of $\Xi_3^T \vdash \Theta, \Omega, \Delta, \Lambda \Downarrow G$ to obtain a proof of $\Xi_3^T \vdash \Theta, \Omega, \Delta, \Lambda \Uparrow \cdot$, by combining it with the interpretation of Ξ_2^T. There are two possible cases for Π_1: either the proof ends with initial and $a_i \in \Omega$, or the proof continues with release, store and begins another asynchronous phase.

i In the first case $\mathsf{initial}_e(\Xi_2^T, j)$ holds for an index j and $j : a_i \in \Omega$; therefore, $G \in \Omega'$, and we interpret the decide on the definitional clause with a decide on the index j, after which we continue with the interpretation of Π_2^T we just saw.

ii In the second case, we have that $\mathit{release}_e(\Xi_2^T, \Xi_4^T)$ holds and we apply the $\Uparrow$ inductive hypothesis to $\Xi_4^T \vdash^T \Theta, \Omega, \Delta, \Lambda \Uparrow \neg\, a_G$ to obtain a proof of $\Xi_4^T \vdash^I \Omega', \Delta', \Lambda \Uparrow \neg\, G$. We obtain the required proof by using cut on the formula $\dagger\partial G$. The left branch of the cut (where the goal formula is $\neg\, \dagger\partial G = \bar{\partial} \neg\, G$) immediately removes the delay and continues with the interpretation of Ξ_4^T, while the right branch stores the cut formula at a temporary index in order to immediately decide on it and continues with the interpretation of Ξ_3^T.

□

We already observed at the beginning of this section that if the Tseitin literals are polarized negatively by the client, the proof has a more constrained shape. By observing the proof of theorem 4 we can immediately state the following corollary:

Corollary 1. If $\Xi^T \vdash^T \Theta \Uparrow a_F$ is provable without cuts and all the Tseitin literals have negative polarization, then $\Xi^T \vdash^I \hat{F}$ is provable without cuts.

Note that we are not asking that the connectives are polarized negatively. Essentially, the negative polarization of Tseitin literals disallows the choice of a formula that is not an immediate subformula of one of those in the context.

2.2.4 Resolution refutations

When we introduced the Tseitin transformation in subsection 2.2.3, its chief motivation was that of creating short CNF formulas suitable for resolution theorem proving. In subsection 1.2.3 we presented an FPC specification for resolution refutations from the literature, where each resolution step is translated to a cut. Although the solution we just presented can interpret proofs certificates for arbitrary FPC specifications, it is restricted to cut-free proof evidence and as such it cannot be used straight away to interpret resolution proofs in the format from the introduction. Nevertheless, we shall now see that it is indeed possible to extend the interpretation we just gave to the case of the resolution FPC.

Consider the formula F from example 5; in order to prove it, we build the Tseitin transformation for its negation, $T((F)^\perp)$ and provide a resolution refutation for it in fig. 2.7. The resolution FPC from subsection 1.2.3 handles the interpretation of refutations of formulas as proofs of their negation; therefore, in our case, it interprets fig. 2.7 as a proof of $(T((F)^\perp))^\perp$, which is (modulo renaming the subscripts of the Tseitin literals) the dualized Tseitin transformation of F; using the interpreter from subsection 2.2.3, we can interpret this as a proof of the Tseitin polarization of F.

Our concern is that of finding a suitable treatment for the cut inferences. Recall from the description of the resolution FPC that the cuts that are introduced correspond to each resolution step, where the deduced clause is the cut formula. As we can see in the refutation trace in fig. 2.7, deduced clauses can contain any combination of logical literals, and Tseitin literals. The strategy will be that of reconstructing the cuts by using as cut formula the formula where we substitute to each literal the subformula it denotes. Operationally, we will use again `copy` clauses for performing substitutions in λProlog. We introduce a predicate that we name `detseitin` for clarity, we initialize the clauses by copying to themselves all atoms, and we take the set of Tseitin clauses and build `detseitin` clauses relating each index to the subformula it denotes. We add to fig. 2.6 the following clauses:

Listing 2.1: Cut for resolution with Tseitin literals

```
cut_ke C1 C2 C3 TF :-
  cut_tke C1 C2 C3 F,
  detseitin F DF, nnf DF PF, polarize_tseitin PF TF.
```

Input formula, from 5 $F^{\perp} := {G_1}^{\perp} \wedge {G_2}^{\perp}$, where ${G_1}^{\perp} := (\neg p \vee q) \wedge (\neg q \vee p)$ and ${G_2}^{\perp} := (\neg p \vee r) \wedge ((q \wedge \neg r) \vee (p \wedge \neg q))$
Tseitin transformation:

$a_{F^{\perp}}$	(2.1)	$(\neg a_{{G_2}^{\perp}} \vee a_{(q \wedge \neg r)} \vee a_{(p \wedge \neg q)})$	(2.7)
$(\neg a_{F^{\perp}} \vee a_{{G_1}^{\perp}})$	(2.2)		
$(\neg a_{F^{\perp}} \vee a_{{G_2}^{\perp}})$	(2.3)	$(\neg a_{q \wedge \neg r} \vee q)$	(2.8)
$(\neg a_{{G_1}^{\perp}} \vee \neg p \vee q)$	(2.4)	$(\neg a_{q \wedge \neg r} \vee \neg r)$	(2.9)
$(\neg a_{{G_1}^{\perp}} \vee \neg q \vee p)$	(2.5)	$(\neg a_{p \wedge \neg q} \vee p)$	(2.10)
$(\neg a_{{G_2}^{\perp}} \vee \neg p \vee r)$	(2.6)	$(\neg a_{p \wedge \neg q} \vee \neg q)$	(2.11)

$a_{{G_1}^{\perp}}$	$2.1, 2.2$	(2.12)	$\neg a_{p \wedge \neg q} \vee q$	$2.10, 2.14$	(2.18)
$a_{{G_2}^{\perp}}$	$2.1, 2.3$	(2.13)	$\neg a_{q \wedge \neg r} \vee p$	$2.8, 2.15$	(2.19)
$\neg p \vee q$	$2.12, 2.4$	(2.14)	$\neg a_{q \wedge \neg r} \vee r$	$2.19, 2.16$	(2.20)
$\neg q \vee p$	$2.12, 2.5$	(2.15)	$\neg a_{q \wedge \neg r}$	$2.20, 2.9$	(2.21)
$\neg p \vee r$	$2.6, 2.13$	(2.16)	$\neg a_{p \wedge \neg q}$	$2.11, 2.18$	(2.22)
$a_{(q \wedge \neg r)} \vee a_{(p \wedge \neg q)}$	$2.7, 2.13$	(2.17)	$a_{q \wedge \neg r}$	$2.17, 2.22$	(2.23)
			$\perp$	$2.23, 2.21$	(2.24)

Figure 2.7: A resolution refutation for the formula in example 6

```
detseitin (A ||- B) (A' ||- B') :-
  detseitin A A', detseitin B B'.
% Continue similarly with the detseitin clauses
```

Note in particular that there the formula F provided by the Tseitin cut expert might contain negated occurrences of Tseitin literals, and therefore after replacing them the formula we get is not in negation normal form. We use a simple predicate `nnf` to transform it to negation normal form, and then we introduce the delays as per definition 8.

We now review the three blocks of the FPC specification in fig. 1.7, through the interpreter in fig. 2.6. The first block concerns the handling of the `start` certificates; while at a first glance nothing surprising happens here, note that the resolution FPC assumes that the storage is empty at the initial state, while the interpreter expects the object proof to include the indexed Tseitin clauses in its initial state. We have seen in the introduction that the `start` certificate gives rise to a simple synthetic rule where all the starting clauses are subsequently stored. This rule is applied immediately and it only uses the store and $\bar{\vee}$ rules, which the interpreter executes right away. Therefore, we reach the initial state that is expected by the interpreter.

Next is the backbone of cuts generated by the `rlisti` certificate constructor. Here we need to translate the cut formula by using the `detseitin` clauses we introduced earlier. Consider for example the first three resolution steps

(2.12,2.13,2.14) (certificate and indexes annotation are ignored for succinctness in this tree):

$$\dfrac{\dfrac{\Pi_{12}}{\vdash T'(G_1) \mathbin{\bar{\vee}} T'(G_2) \Uparrow (T'(G_1))^{\perp}} \quad \dfrac{\dfrac{\Pi_{13}}{\vdash T'(F), T'(G_1) \Uparrow (T'(G_2))^{\perp}} \quad \dfrac{\dfrac{\Pi_{14}}{\vdash T'(F) \Uparrow \neg p \mathbin{\bar{\vee}} q} \quad \dfrac{\vdots}{\vdash T'(F) \Uparrow (p \mathbin{\bar{\wedge}} \neg q)}}{\vdash T'(F), T'(G_1) \Uparrow T'(G_2)}\,\text{res}}{\vdash T'(G_1) \mathbin{\bar{\vee}} T'(G_2) \Uparrow T'(G_1)}\,\text{res}}{\vdash \cdot \Uparrow \bar{\partial}(T'(G_1) \mathbin{\bar{\vee}} T'(G_2))}\,\text{res}$$

The situation is slightly more complex when several Tseitin or non-Tseitin are mixed. For example, in steps 2.17 and 2.18 we get the following (where Γ contains the previously stored deduced lemmas, and the second tree contains the right branch Π_r of the first tree):

$$\dfrac{\dfrac{\Pi_{17}}{\vdash \Gamma \Uparrow (q \mathbin{\bar{\wedge}} \neg r) \mathbin{\bar{\vee}} (p \mathbin{\bar{\wedge}} \neg q)} \quad \dfrac{\Pi_r}{\vdash \Gamma \Uparrow (\neg q \mathbin{\bar{\vee}} r) \mathbin{\bar{\wedge}} (\neg p \mathbin{\bar{\vee}} q)}}{\vdash \Gamma \Uparrow \cdot}$$

$$\dfrac{\dfrac{\Pi_{18}}{\vdash \Gamma, (\neg q \mathbin{\bar{\vee}} r) \mathbin{\bar{\wedge}} (\neg p \mathbin{\bar{\vee}} q) \Uparrow (\neg p \mathbin{\bar{\vee}} q) \mathbin{\bar{\vee}} q} \quad \dfrac{\vdots}{\vdash \Gamma, (\neg q \mathbin{\bar{\vee}} r) \mathbin{\bar{\wedge}} (\neg p \mathbin{\bar{\vee}} q) \Uparrow (p \mathbin{\bar{\wedge}} \neg q) \mathbin{\bar{\wedge}} q}}{\vdash \Gamma \Uparrow (\neg q \mathbin{\bar{\vee}} r) \mathbin{\bar{\wedge}} (\neg p \mathbin{\bar{\vee}} q)}$$

When building the second cut formula, the occurrence of the Tseitin literal $a_{p \wedge \neg q}$ in $\neg a_{p \wedge \neg q} \vee q$ is replaced by the denoted formula, and through the negation normal form transformation we obtain $(\neg p \mathbin{\bar{\wedge}} q) \mathbin{\bar{\vee}} q$.

Finally, the `dlist` constructor checks each resolution step on the left-hand side of the cuts we have introduced. This is where we see in action the interpreter for the Tseitin constants. In fact, a certificate of the form `dlist [i,j]` steers the proof through at least one decide on one of i or j, which are indexes for possibly Tseitin clauses. Let's analyze Π_{12}, corresponding to the resolution step 2.12. For the sake of greater clearness, fig. 2.8 shows how the argument is reconstructed by leaving Tseitin constants in, in the original resolution FPC. The `dlst` certificate determines a decide rule on the Tseitin clause stored at 2.3, which is then split in two branches; the first one is concluded by finding $a_{F^{\perp}}$ in the storage at index 1, while the second is concluded by finding the resolvant indexed at `lit`.

$$\dfrac{\dfrac{\dfrac{\overline{\texttt{dl [1]} \vdash \texttt{1} : \neg a_{F^{\perp}}, \texttt{l} : a_{G_1{}^{\perp}} \Downarrow a_{F^{\perp}}} \quad \overline{\texttt{dl [1]} \vdash \texttt{1} : \neg a_{F^{\perp}}, \texttt{l} : a_{G_1{}^{\perp}} \Downarrow \neg a_{G_1{}^{\perp}}}}{\texttt{dlist [2.1]} \vdash \texttt{2.1} : \neg a_{F^{\perp}}, \texttt{lit} : a_{G_1{}^{\perp}} \Downarrow (a_{F^{\perp}} \mathbin{\bar{\wedge}} \neg a_{G_1{}^{\perp}})}}{\texttt{dlist [2.1,2.2]} \vdash \texttt{2.1} : \neg a_{F^{\perp}}, \texttt{lit} : a_{G_1{}^{\perp}} \Uparrow \cdot}\,\text{dec 2.2}}{\texttt{dlist [2.1,2.2]} \vdash \texttt{2.1} : \neg a_{F^{\perp}} \Uparrow a_{G_1{}^{\perp}}}$$

Figure 2.8: Resolution step check for 2.12

Figure 2.9 contains the interpreted proof. We have just seen that the decide on clause 2.2 uncovers an $\bar{\wedge}$ that has a left-side branch that ends with an initial on the complementary literal stored at 2.1, an index that corresponds to a formula that is available in the non-Tseitin storage where it refers to F. Therefore,

the clause for the decide of the Tseitin interpreter is satisfied: this is mapped to a direct decide on the stored 2.1, to uncover its subformulas. The correct index for $\mathbin{\dot{\vee}}$ is nondeterministically reconstructed, and we reach the second initial from the original resolution proof. The Tseitin interpreter is now focused on a negative formula, thus performs a release and ends the proof with a `mimic` on the dual subformula stored at `lit`.

$$\dfrac{\dfrac{\dfrac{\texttt{mimic lit} \vdash 1 : \dot{\partial} F^{\perp}, \texttt{lit} : (\neg p \mathbin{\bar{\vee}} q) \mathbin{\dot{\wedge}} (\neg q \mathbin{\bar{\vee}} p) \Uparrow (p \mathbin{\dot{\wedge}} \neg q) \mathbin{\bar{\vee}} (q \mathbin{\dot{\wedge}} \neg p)}{\texttt{dlist [2.1]} \vdash 1 : \dot{\partial} F^{\perp}, \texttt{lit} : (\neg p \mathbin{\bar{\vee}} q) \mathbin{\dot{\wedge}} (\neg q \mathbin{\bar{\vee}} p) \Downarrow \bar{\partial}(T'(G_1))}\,\text{rel}}{\texttt{dlist [2.1]} \vdash 1 : \dot{\partial} F^{\perp}, \texttt{lit} : (\neg p \mathbin{\bar{\vee}} q) \mathbin{\dot{\wedge}} (\neg q \mathbin{\bar{\vee}} p) \Downarrow \bar{\partial}(T'(G_1)) \mathbin{\dot{\vee}} \bar{\partial}(T'(G_2))}}{\dfrac{\texttt{dlist [2.1,2.2]} \vdash 1 : \dot{\partial} F^{\perp}, \texttt{lit} : (\neg p \mathbin{\bar{\vee}} q) \mathbin{\dot{\wedge}} (\neg q \mathbin{\bar{\vee}} p) \Uparrow \cdot}{\texttt{dlist [2.1,2.2]} \vdash 1 : T'(F) \Uparrow (\neg p \mathbin{\bar{\vee}} q) \mathbin{\dot{\wedge}} (\neg q \mathbin{\bar{\vee}} p)}}\,\text{dec 2.2}$$

Figure 2.9: Tseitin interpretation of fig. 2.8

This case was simple: the resolution step precisely resolved the Tseitin clause against its head literal, and the literal denoted the formula F which was in the context. What we get is exactly the selection of an immediate subformula. This is the case also for the proofs trees Π_{13}, Π_{14} and Π_{17} in the partial backbones we considered before. Consider now the subproof Π_{18} verifying the resolution step 2.18, which starts with the certificate `dlist [2.10, 2.14]` and resolves a Tseitin clause with a deduced clause upon a non-Tseitin literal. Its interpretation is shown in fig. 2.10. Just like in fig. 2.9, an entire subformula is stored at `lit`. However, this is no longer the unveiled body of a definitional clause, but it is instead the head of the clause composed with a literal from another clause. Still, the decision on index 2.10 corresponds to a decision on a definitional clause for a subformula which is now stored ad `lit` (to make this clearer, see the partial derivation for the original resolution proof in fig. 2.11). The decision on the stored subformula leads to the selection of its subformula $\neg p$ and, since the other resolvant is a deduced clause that does not contain any Tseitin literal, the proof can be concluded as a usual resolution proof.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdash \Gamma, \texttt{lit} : (\neg p \mathbin{\dot{\vee}} q), \texttt{lit} : q, \texttt{lit} : \neg p \Downarrow p}{\qquad}\quad \dfrac{\vdash \Gamma, \texttt{lit} : (\neg p \mathbin{\dot{\vee}} q), \texttt{lit} : q, \texttt{lit} : \neg p \Downarrow \neg q}{\qquad}}{\texttt{dlist []} \vdash \Gamma, \texttt{lit} : (\neg p \mathbin{\dot{\vee}} q), \texttt{lit} : q, \texttt{lit} : \neg p \Downarrow p \mathbin{\dot{\wedge}} \neg q}}{\texttt{dlist [2.14]} \vdash \Gamma, \texttt{lit} : (\neg p \mathbin{\dot{\vee}} q), \texttt{lit} : q, \texttt{lit} : \neg p \Uparrow \cdot}\,\text{decide 2.14}}{\texttt{dlist [2.14]} \vdash \Gamma, \texttt{lit} : (\neg p \mathbin{\dot{\vee}} q), \texttt{lit} : q \Uparrow \neg p}}{\texttt{dlist [2.14]} \vdash \Gamma, \texttt{lit} : (\neg p \mathbin{\dot{\vee}} q), \texttt{lit} : q \Downarrow \neg p}}{\texttt{dlist [2.14]} \vdash \Gamma, \texttt{lit} : (\neg p \mathbin{\dot{\vee}} q), \texttt{lit} : q \Downarrow (\neg p \mathbin{\dot{\vee}} q)}}{\texttt{dlist [2.10,2.14]} \vdash \Gamma, \texttt{lit} : (\neg p \mathbin{\dot{\vee}} q), \texttt{lit} : q \Uparrow \cdot}\,\text{decide } \texttt{lit}}{\texttt{dlist [2.10,2.14]} \vdash \Gamma \Uparrow (\neg p \mathbin{\dot{\vee}} q) \mathbin{\bar{\vee}} q}$$

Figure 2.10: Tseitin interpretation of the resolution step check 2.18

We have seen that with a careful treatment of the cut formulas we have obtained a reconstruction procedure for resolution using the Tseitin transfor-

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{dlist [2.14]} \vdash \texttt{lit} : \neg\, a_{p\wedge\neg q}, \texttt{lit} : q \Downarrow a_{p\wedge\neg q}} \quad \dfrac{\vdots}{\texttt{dlist [2.14]} \vdash \texttt{lit} : \neg\, a_{p\wedge\neg q}, \texttt{lit} : q \Downarrow \neg p}}{\texttt{dlist [2.14]} \vdash \texttt{lit} : \neg\, a_{p\wedge\neg q}, \texttt{lit} : q \Downarrow a_{p\wedge\neg q} \wedge \neg p}}{\texttt{dlist [2.10,2.14]} \vdash \texttt{lit} : \neg\, a_{p\wedge\neg q}, \texttt{lit} : q \Uparrow \cdot}}{\texttt{dlist [2.10,2.14]} \vdash \Uparrow \neg\, a_{p\wedge\neg q} \vee q}$$

Figure 2.11: Partial derivation of the resolution step check for eq. (2.23)

mation as a pre-processing step. Moreover, the resolution FPC from subsection 1.2.3 uses a negative polarization for the literals; therefore, we do not introduce more cuts in the proof due to this additional step. One could argue that, since some resolution steps (such as inference 2.12) are only used to expand on a Tseitin clause, then one should be able to avoid completely the use of cuts in many places. While this is certainly true, our aim here was that of devising a general methodology to treat proof evidence with Tseitin definitions. A more ad-hoc solution specifically targetting resolution refutation could be more optimized.

2.3 Further references

skolemization has a long story in mathematical logic, either as a technique for theorem proving or as part of the eponymous theorem of Skolem. This theorem is usually stated in the context of model theory as saying that any first order theory that has a model has a countable model. Its proof involves constructing a model with a set of additional constants and function symbols, in a similar way as we see in the proofs of skolemized statemets. The paradoxical result is that even a theory that appearently talks about uncountable sets has a model where those sets are countable. This seems to be rooted in the fact that we can replace the actual individuals of the formula with formal placeholders, that only maintain the information on their mutual dependency (with various levels of strictness, as we have seen). Hilary Putnam (who helped popularize skolemization as a technical device for theorem proving [DP60]) used this fact at the center of one of his most notoriously difficult arguments in his philosophy of mind [Put80], whereby he refused the idea that what one talks about in logic is a mind-independent reality; however one tries to further specify what is the reference of their discourse, the result is *"just more theory"* being added, which is still subject to Skolem's theorem.

Several languages for the representation of mathematical knowledge, or inferences more generally, have faced the problem of including a construct to handle generic individuals. We cite a few, and the way their constructions are devised. This should convince us that they could be handled by an appropriate FPC with the methodology that we have introduced:

- The TSTP format, part of the TPTP benchmark for theorem provers, contains a proof directive to indicate the creation of a new name [SZS]. This

is not limited to skolemization but could in principle serve other purpose, which is still perfectly in scope of our treatment.

- The Alethe proof language, proposed for use in `cvc5`, uses choice operators instead. While different in presentation, they can be explained as naming devices as well.
- The RDF format for the semantic web has a notion of "Blank node" that doesn't have a name and depends on its location. Those nodes can be assigned a name by the user through a method that is called skolemization in that context. A similar attempt by the W3C was the Provenance Markup Language [Sem].
- The MathML format for the representation on the web of mathematical data and mathematical proofs, also by the W3c [Bou] provides a primitive notion of bound variable that needs to pass an aciclicity constraint (however at the moment of writing most of MathML is not implemented in any browser).

Tseitin transformation are present in many places in the literature of proof theory. In the same paper where he introduced focusing for linear logic, J.-M. Andreoli proposed a logic programming language called LinLog, that resembles languages based on uniform proofs. This language only uses a fragment of linear logic, and thus its search procedure is not directly a complete focused theorem proving procedure; in order to use the proof-search procedure of LinLog for full linear logic, Andreoli provides a series of transformations of linear logic formulas into this fragment. One of these transformations, which he calls skolemization, is actually a Tseitin transformation; our opinion is that he would just have needed to include delays in his system. The *geometrisation* of theories by Dyckhoff and Negri is also tightly related [DN15].

The Z3 SMT solver has some documentation about how the Tseitin transformation is presented in proof objects produced by the prover [MB08]: literals are introduced as quotations of each subformula, and the proof trace uses an axiom that is parametrized by the subformula, indicating that a definitional clause has been used. Current works on extracting proofs from Z3 for the use in proof assistants [Boh09; PSU21] build lemmas for each of these steps; however, since the proofs are a variation of resolution proofs it should be possible to reconstruct them without cuts, as per corollary 1.

Chapter 3

Linearized arithmetic

The expressiveness of first order logic is very limited, especially since its foundations eschew on purpose any mathematical principle. If we wish to formalize reasoning over a specific theory, there are several roads that can be taken.

In the realm of proof assistants it is typical to have different notions of inductive definition as a basis for the description of mathematical structures. Indeed, inductive definitions are very close to the usual style of definitions in mathematics.

David Baelde [BM07b] introduced the system μMALL, consisting of linear logic without exponentials and with fixed points, and proved completeness of a focused proof system for it. In this chapter we prove some additional results about this and related systems. We show some initial connections with axiomatic systems of arithmetic, and we propose a way to identify functional computations.

While the formal semantics of Foundational Proof Certificates has not yet been extended to these systems, the results exposed in this chapter are meant to help lay out such a semantics. Focusing has already been employed to develop theorem proving tools in [BMS10]. Hetzl and Wong started a program analyzing methods in theorem proving for arithmetic with the help of structural proof theory [HW18].

3.1 Systems of arithmetic based on fixed point definitions

The logical basis for most of this chapter will be multiplicative-additive linear logic, the subsystem of linear logic where contraction and weakening are never allowed. The $\barwedge, \bar{\wedge}$ and $\barvee, \bar{\vee}$ connectives are not provably equivalent without contraction and weakening, and we will therefore introduce the linear logic notation $\otimes, \&$ and $\oplus, \parr$. Once contraction and weakening are allowed in MALL we get full equiprovability with *LK* with the linear connectives replaced by the polarized ones.

The system μMALL adds to MALL the two unary connectives μ and ν, denoting respectively the least and greatest fixpoint of a predicate operator. By following Church's notation, as we did in 1, we have that a propositional predicate operator has type $o \to o$; the two new connectives construct a new predicate out of the operator, and therefore in the propositional case their type is $(o \to o) \to o$. When we move to first order logic, the type of n-ary predi-

cate operators is $(\overbrace{i \to \cdots \to i}^{n} \to o) \to (\overbrace{i \to \cdots \to i}^{n} \to o)$. This means that the μ and ν operators are actually a family of operators one for each arity, building predicates of arity n from operators of arity n, and whose type is $((\overbrace{i \to \cdots \to i}^{n} \to o) \to (\overbrace{i \to \cdots \to i}^{n} \to o)) \to (\overbrace{i \to \cdots \to i}^{n} \to o)$. However, there is usually no ambiguity in the usage, and we will leave out these details. In addition to these, we treat equality as a logical connective, and include the symbols $=$ and $\neq$ of type $i \to i \to o$.

The system has no undefined predicates, and therefore all predicates are built out of the μ or ν operators applied to an operator. We will explicitly use the lambda notation when building predicates, like in $((\mu\lambda P\lambda x(B\ P\ x))\ t)$: here we abstract both on a term variable x and on a predicate variable P (this is reminiscent of the fact that the inductive reasoning we are trying to capture deals with a restricted form of second order reasoning). Predicate operators, and therefore predicate variables, can only appear in fixed point constructions; the polarity of atoms built with predicate variables is therefore dependent on what kind of occurrence this is, and will be positive when occurring in a μ fixed point, negative in the ν fixed point.

If we include the constant symbols **0** of arity 0 and s of arity 1, we can now express natural numbers with the fixed point:

$$nat := \mu\lambda N\lambda n(n = \mathbf{0} \oplus \exists n'(n = (s\ n') \otimes N\ n'))$$

3.1.1 Proof rules

The rules are divided in three groups: fig. 3.1 shows the rules for the fragment of linear logic without exponentials; fig. 3.2 shows the rules for fixed points; fig. 3.3 contains the weakening and contraction rules that extend the system μMALL to be classical. We call the classical system μLK.

The ν-rule can be better understood by noting that in a two-sided sequent calculus it becomes the following two rules:

$$\frac{\Gamma, S\bar{t} \vdash \Delta \quad BS\bar{x} \vdash S\bar{x}}{\Gamma, \mu B\bar{t} \vdash \Delta}\ \textit{induction} \qquad \frac{\Gamma \vdash \Delta, S\bar{t} \quad S\bar{x} \vdash BS\bar{x}}{\Gamma \vdash \nu B\bar{t}, \Delta}\ \textit{coinduction}$$

That is, the one rule for ν yields both coinduction and induction. In general, we shall speak of the higher-order substitution term S used in both of these rules as the *invariant* of that rule (therefore we will not use the term co-invariant even though that might be more appropriate in some settings).

The left-hand side states that, when we are drawing a conclusion from the least fixed point of an operator, we need to start by finding an invariant for the operator. Then, the proof consists a proof of the conclusion from the invariant, together with a proof of the fact that the invariant really is one. In the case of the nat fixed point we just introduced, we get:

$$\frac{\Gamma, S\,x \vdash \Delta \quad \dfrac{\vdash S\,\mathbf{0} \quad \dfrac{S\ n' \vdash S\,(s\ n')}{\exists n'(x = (s\ n') \otimes S\ n') \vdash S\,x}}{x = \mathbf{0} \oplus \exists n'(x = (s\ n') \otimes S\ n') \vdash S\,x}}{\Gamma, \mathit{nat}\,x \vdash \Delta}\ \textit{induction}$$

The three premises are easily recognizable as a formalization of induction on natural numbers, where we check that S holds of 0 and that whenever S holds of n', it holds of its successor as well.

We make the following additional observations about this proof system.

1. The $\mu\nu$ rule is a limited form of the initial rule. The general form of the initial rule, namely, that the sequent $\vdash Q, (Q)^{\perp}$ is provable, is admissible: this more general rule is named init.

2. The rule for μ allows for the μ fixed point to be unfolded. This rule captures, in part, the identification of μB with $B(\mu B)$; that is, that μB is a fixed point of B. This inference rules allows one occurrence of B in (μB) to be expanded to two occurrences of B in $B(\mu B)$. In this way, unbounded behaviors can appear in μMALL where it did not occur in MALL.

3. The *unfold* rule in Figure 3.3, which simply unfolds ν-expression, is admissible in this proof system by using the ν-rule with the invariant $S = B(\nu B)$.

4. In the $\neq$-introduction rule, if the terms t and t' are not unifiable, then the premise is empty and the conclusion is immediately proved.

3.1.2 Encoding arithmetical statements

We saw how the induction rule can capture a reasoning similar to the usual induction on natural numbers. We now bring forward this argument to arithmetical statements, and show in particular that since μLK is based on classical logic it can prove an appropriate encoding of all the statements of Peano Arithmetic (PA).

Consider as a standard presentation of PA a system based on first order (non polarized) logic with equality, with a language consisting of $\mathbf{0}, s, +, \cdot$ and the following axioms:

$$\begin{array}{ll}
\forall x.\ (s\,x) \neq \mathbf{0} & \forall x.\ \forall y (x + s\,x) = s(x + y) \\
\forall x \forall y.\ (s\,x = s\,y) \supset (x = y) & \forall x.\ (x \cdot \mathbf{0} = \mathbf{0}) \\
\forall x.\ (x + \mathbf{0} = x) & \forall x.\ (x \cdot s\,y = (x \cdot y + x)) \\
(A\mathbf{0} \wedge \forall x.\ (Ax \supset A(s\,x))) \supset \forall x.\ Ax &
\end{array}$$

We start by observing that while we need to have the constants $\mathbf{0}$ and s in the language, we don't need to introduce the symbols for addition and multiplication in our language. Indeed, it is sufficient to encode them as the fixed points:

$$\begin{aligned}
\mathit{plus} :=& \mu\lambda P \lambda n \lambda m \lambda p ((n = \mathbf{0} \otimes m = p) \oplus \\
& \exists n' \exists p' (n = (s\,n') \otimes p = (s\,p') \otimes P\ n'\ m\ p')) \\
\mathit{mult} :=& \mu\lambda M \lambda n \lambda m \lambda p \big((n = \mathbf{0} \otimes p = \mathbf{0}) \oplus \\
& \exists n' \exists p' (n = (s\,n') \otimes \mathit{plus}\ m\ p'\ p \otimes M\ n'\ m\ p')\big)
\end{aligned}$$

$$\frac{\vdash \Gamma, P \quad \vdash \Delta, Q}{\vdash \Gamma, \Delta, P \otimes Q}\ \otimes \qquad \frac{}{\vdash 1}\ 1 \qquad \frac{\vdash \Gamma, P, Q}{\vdash \Gamma, P \parr Q}\ \parr \qquad \frac{\vdash \Gamma}{\vdash \Gamma, \bot}\ \bot$$

$$\frac{\vdash \Gamma, P \quad \vdash \Gamma, Q}{\vdash \Gamma, P \,\&\, Q}\ \& \qquad \frac{}{\vdash \Delta, \top}\ \top \qquad \frac{\vdash \Gamma, P_i}{\vdash \Gamma, P_0 \oplus P_1}\ \oplus$$

$$\frac{\vdash \Gamma, Pt}{\vdash \Gamma, \exists x.Px}\ \exists \qquad \frac{\vdash \Gamma, Py}{\vdash \Gamma, \forall x.Px}\ \forall$$

Figure 3.1: Multiplicative-Additive linear logic

$$\frac{\{\ \vdash \Gamma\theta : \theta = mgu(t, t')\ \}}{\vdash \Gamma, t \neq t'}\ \neq \qquad \frac{}{\vdash t = t}\ = \qquad \frac{\vdash \Gamma, S\bar{t} \quad \vdash BS\bar{x}, \overline{(S\bar{x})}}{\vdash \Gamma, \nu B\bar{t}}\ \nu$$

$$\frac{\vdash \Gamma, B(\mu B)\bar{t}}{\vdash \Gamma, \mu B\bar{t}}\ \mu \qquad \frac{}{\vdash \mu B\bar{t}, \nu\overline{B}\bar{t}}\ \mu\nu$$

Figure 3.2: Add these rules to the core set to get μMALL

$$\frac{\vdash \Gamma, B(\nu B)\bar{t}}{\vdash \Gamma, \nu B\bar{t}}\ \textit{unfold} \qquad \frac{\vdash \Gamma, Q, Q}{\vdash \Gamma, Q}\ \mathrm{C} \qquad \frac{\vdash \Gamma}{\vdash \Gamma, Q}\ \mathrm{W} \qquad \frac{\vdash \Gamma, Q \quad \vdash \Delta, \overline{Q}}{\vdash \Gamma, \Delta}\ \text{cut}$$

Figure 3.3: Additional rules for classical logic

The induction principle is now a rule rather than an axiom, and we carry out induction on the naturals by reasoning over the inductively defined nat predicate. Accordingly, we define a translation $\Box^\circ$ from formulas of PA to formulas of μLK that extends the usual polarizing translation $\hat{\Box}$ as follows:

Definition 9 (Embedding of PA in μLK)**.**

$$\begin{aligned}
(\forall x.\, A)^\circ &:= \forall x.\, ((\mathit{nat}\ x)^\perp \parr (Ax)) & \qquad (\exists x.\, A)^\circ &:= \exists x.\, (\mathit{nat}\ x \otimes (Ax)) \\
(x + y = w)^\circ &:= \mathit{plus}\, x\, y\, w & \qquad (x \cdot y = w)^\circ &:= \mathit{mult}\, x\, y\, w \\
(A \wedge B)^\circ &:= \text{polar. translation} & \qquad (A \vee B)^\circ &:= \text{polar. translation} \\
(\neg A)^\circ &:= \neg(A)^\circ
\end{aligned}$$

Checking that the translation of the axioms of PA is provable in μLK is not particularly challenging:

Theorem 5 (μLK contains Peano arithmetic)**.** *Let Q be any unpolarized formula and let $\hat{Q}$ be a polarized version of Q. If Q is provable in Peano arithmetic then $(\hat{Q})^\circ$ is provable in μLK.*

Proof. It is easy to prove that $mult$ and $plus$ describe precisely the multiplication and addition operations on natural numbers. Furthermore, the translations of the Peano Axioms can all be proved in μLK. We illustrate just one of these axioms here. In particular, a polarization of the translation of the induction scheme is

$$((A\mathbf{0} \otimes \forall x.\, ((nat\ x)^\perp \parr (Ax)^\perp \parr A(s\ x))))^\perp \parr \forall x.\, ((nat\ x)^\perp \parr Ax)$$

An application of the ν rule to the second occurrence of $(nat\ x)^\perp$ can provide an immediate proof of this axiom. Finally, the cut rule in μLK allows us to encode the inference rule of modus ponens. □

As we have seen for the definitions of $mult$ and $plus$, the mechanism of fixed points is particularly well suited for encoding definitions in the style of logic programming. We can encode the specification of primitive recursive functions as μMALL formulas in the following way:

$$succ(y, z) := z = Sy$$
$$proj_n^i(\bar{y}, z) := z = y_i$$

If F and $G_1 \dots G_n$ are the specification of the functions $f, g_1, \dots g_n$, then the specification H of their composition h is given by

$$H(\bar{y}, z) := \forall \bar{x}.G_1(\bar{y}, x_1) \multimap G_2(\bar{y}, x_2) \dots G_n(\bar{y}, x_n) \multimap F(\bar{x}, z)$$

If F and G are the specification of f of arity n and g of arity $n + 2$, then the specification H of the function h obtained by primitive recursion is

$$H := \mu\lambda H.\lambda n.\lambda \bar{y}.\lambda z.(n = 0 \otimes F(\bar{y}, z)) \\ \oplus (\exists m.\exists o.n = Sm \otimes H(n, \bar{y}, o) \otimes G(m, o, \bar{y}, z))$$

We can easily be convinced that for any primitive recursive function f there is a specification F such that $f(\bar{x}) = z$ if and only if $F(\bar{x}, z)$ holds.

3.1.3 Consistency of μ LK

We move on to show that the classical system arising from the addition of the structural rules to μMALL is consistent. By second-order linear logic, LL^2, we mean the logic of MALL with the addition of the following logical connectives: two exponentials ! and ?, negation $(\cdot)^\perp$, equality and non-equality, and first-order and second-order quantification (no occurrences of fixed points are permitted). Cut-elimination of this version of LL^2 follows from Girard's original cut-elimination proof [Gir87b] (see also [Oka99]) and the cut-elimination proofs known for equality and non-equality [Gir92; SH93].

We translate μLK formulas into LL^2 formulas by translating fixed point expressions into second-order quantified formulas, quantifying over the invariant. The least fixed point expression $\mu B\bar{x}$ should be translated to a formula roughly of the form $\forall S\Big(!(\forall \vec{y}\ .\ B S\vec{y} \multimap S\vec{y}) \multimap S\bar{x}\ \Big)$. This translation must also insert ? into formulas in order to account for the fact that in μLK, any formula can be contracted and weakened at any point in a proof. The translation is given as follows.

- $\lceil t = s \rceil = ?(t = s)$ and $\lceil t \neq s \rceil = ?(t \neq s)$
- $\lceil \forall x.Px \rceil = ? \forall x.\lceil Px \rceil$ and $\lceil \exists x.Px \rceil = ? \exists x.\lceil Px \rceil$.
- $\lceil B \otimes C \rceil = ?(\lceil B \rceil \otimes \lceil C \rceil)$, $\lceil B \parr C \rceil = ?(\lceil B \rceil \parr \lceil C \rceil)$, $\lceil B \,\&\, C \rceil = ?(\lceil B \rceil \,\&\, \lceil C \rceil)$,
  $\lceil B \oplus C \rceil = ?(\lceil B \rceil \oplus \lceil C \rceil)$
- $\lceil 1 \rceil = ? 1$, $\lceil \bot \rceil = ? \bot$, $\lceil 0 \rceil = ? 0$, $\lceil \top \rceil = ? \top$
- $\lceil \mu B \bar{x} \rceil = ? \forall S[(? \exists \vec{y} \,.\, \lceil B \rceil S\vec{y} \otimes (S\vec{y})^{\bot}) \parr S\bar{x}\,]$
- $\lceil \nu B \bar{x} \rceil = ? \exists S[(! \forall \vec{y} \,.\, \lceil B \rceil S\vec{y} \parr (S\vec{y})^{\bot}) \otimes S\bar{x}\,]$
- $\lceil A \rceil = A$ where A is an atomic formula.
- The $\lceil \cdot \rceil$ operator commutes with λ-abstraction: $\lceil \lambda x.B \rceil = \lambda x.\lceil B \rceil$. This feature of $\lceil \cdot \rceil$ permit translating invariants and the body of fixed point expressions.
- The $\lceil \cdot \rceil$ operator can be applied to a multiset of formulas: $\lceil \Gamma \rceil = \{\lceil P \rceil \mid P \in \Gamma\}$.

Note that when B is the λ-abstraction $\lambda p \lambda \bar{x}.C$, where C is a μMALL formula, p is a first-order predicate variable, and $\bar{x}$ is a list of first-order variables, then $\lceil B \rceil \lceil S \rceil \bar{t}$ is equal to $\lceil BS\bar{t} \rceil$ up to λ-conversion. We shall also need the following inference rule in LL^2, which is a kind of generalization of the cut rule.

$$\frac{\vdash \Gamma, BQ\bar{t} \quad \vdash \neg(Q\bar{x}), P\bar{x}}{\vdash \Gamma, BP\bar{t}} \text{ deep.}$$

Here, of course, the first-order variables $\bar{x}$ are new. Also, the expression B has the type that takes a first-order predicate to a first-order predicate and also *monotonic*, meaning that there are no occurrences of negated predicate variables in B. It is proved in [BM07a, Proposition 2] that this rule is admissible in LL^2. This rule essentially allows us to move from the fact that $Q \subseteq P$ and to the fact that $BQ \subseteq BP$.

Lemma 1. If $\vdash \Gamma$ is derivable in μLK then $\vdash \lceil \Gamma \rceil$ is derivable in LL^2.

Proof. We proceed by induction on the structure of cut-free μLK proofs. In particular, assume that $\vdash \Gamma$ has a cut-free μLK proof Ξ.

Case: The last inference rule of Ξ comes from Figure 3.13.2, i.e., it is an introduction rule for a propositional connective, a unit, or a quantifier. For example, assume that this last inference rule is the following $\otimes$ introduction rule.

$$\frac{\vdash \Gamma, P \quad \vdash \Delta, Q}{\vdash \Gamma, \Delta, P \otimes Q} \otimes$$

By the inductive assumption, $\vdash \lceil \Gamma \rceil, \lceil P \rceil$ and $\vdash \lceil \Delta \rceil, \lceil Q \rceil$ have LL^2 proofs. Hence, $\vdash \lceil \Gamma \rceil, \lceil \Delta \rceil, \lceil P \rceil \otimes \lceil Q \rceil$ has an LL^2 proof. By using the dereliction rule for $?$ and the definition of $\lceil \cdot \rceil$, we know that $\vdash \lceil \Gamma, \Delta \rceil, \lceil P \otimes Q \rceil$ has an LL^2 proof.

Case: The last inference rule is either weakening W or contraction C. Since the image of $\lceil\cdot\rceil$ always has a $?$ exponential as its top-level connective, the corresponding LL^2 inference rule is built with the same structural rule.

Case: The last inference rule of Ξ is one of the fixed point rules from Figure 3.2. Assume, for example, that the last rule is

$$\frac{}{\vdash \mu B\bar{t}, \nu\overline{B}\bar{t}}\ \mu\nu$$

The desired translation of this inference rule into LL^2 is

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdash \lceil B\rceil S\vec{y}, \lceil(B)^\perp\rceil(\lambda\vec{w}(S\vec{w})^\perp)\vec{y} \quad \overline{\vdash (S\vec{y})^\perp, \neg((S\bar{x})^\perp)}\ \text{init}}{\vdash \lceil B\rceil S\vec{y} \otimes (S\vec{y})^\perp, \lceil(B)^\perp\rceil(\lambda\vec{w}(S\vec{w})^\perp)\vec{y} \parr \neg((S\vec{y})^\perp)}\ \parr,\otimes}{\vdash ?\exists\vec{y}.\lceil B\rceil S\vec{y} \otimes (S\vec{y})^\perp, !(\forall\vec{y}.\lceil(B)^\perp\rceil(\lambda\vec{w}(S\vec{w})^\perp)\vec{y} \parr \neg((S\vec{y})^\perp))}\ !R, ?D, \forall, \exists \quad \dfrac{}{\vdash S\bar{x}, (S\bar{x})^\perp}\ \text{init}}{\vdash ?(\exists\vec{y}.\lceil B\rceil S\vec{y} \otimes (S\vec{y})^\perp) \parr S\bar{x}\,, !(\forall\vec{y}.\lceil(B)^\perp\rceil(\lambda\vec{w}(S\vec{w})^\perp)\vec{y} \parr \neg((S\vec{y})^\perp)) \otimes (S\bar{x})^\perp}\ \parr,\otimes}{\vdash ?(\exists\vec{y}\,.\,\lceil B\rceil S\vec{y} \otimes (S\vec{y})^\perp) \parr S\bar{x}\,, \exists S[!(\forall\vec{y}\,.\,\lceil(B)^\perp\rceil S\vec{y} \parr (S\vec{y})^\perp) \otimes S\bar{x}\,]}\ \exists\lambda\vec{w}(S\vec{w})^\perp$$
$$\dfrac{}{\vdash ? \forall S[?(\exists\vec{y}\,.\,\lceil B\rceil S\vec{y} \otimes (S\vec{y})^\perp) \parr S\bar{x}\,], ? \exists S[!(\forall\vec{y}\,.\,\lceil(B)^\perp\rceil S\vec{y} \parr (S\vec{y})^\perp) \otimes S\bar{x}\,]}\ ?D, \forall$$

An induction on the structure of the formula B provides a proof that there is an LL^2 proof of remaining open premise.

Assume instead that the last rule of Ξ is the introduction for ν, namely,

$$\frac{\vdash \Gamma, S\bar{t} \quad \vdash BS\bar{x}, (S\bar{x})^\perp}{\vdash \Gamma, \nu B\bar{t}}\ \nu$$

The higher-order quantifier that appears in the LL^2 encoding is instantiated with $\lceil S\rceil$. Thus, the desired LL^2 proof is

$$\dfrac{\dfrac{\dfrac{\vdash \lceil BS\bar{x}\rceil, \lceil(S\bar{x})^\perp\rceil \quad \vdash \neg(\lceil(S\bar{x})^\perp\rceil), \neg(\lceil S\rceil\bar{x})}{\vdash \lceil B\rceil\lceil S\rceil\bar{x}, \neg(\lceil S\rceil\bar{x})}\ \text{cut} \quad \vdash \lceil\Gamma\rceil, \lceil S\bar{t}\rceil}{\vdash \lceil\Gamma\rceil, !(\forall\vec{y}\,.\,\lceil B\rceil\lceil S\rceil\vec{y} \parr \neg(\lceil S\rceil\vec{y})) \otimes \lceil S\rceil\bar{t}}\ \otimes,\forall,\parr}{\vdash \lceil\Gamma\rceil, ? \exists S[!(\forall\vec{y}\,.\,\lceil B\rceil S\vec{y} \parr \neg(S\vec{y})) \otimes S\bar{t}\,]}\ ?D, \exists\ S \mapsto \lceil S\rceil$$

By the inductive hypothesis, the leftmost and rightmost premises have LL^2 proof. Induction on first-order abstractions such as S shows that the middle premise also has an LL^2 proof.

Assume instead that the last rule of Ξ is the introduction for μ, namely,

$$\frac{\vdash \Gamma, B(\mu B)\bar{t}}{\vdash \Gamma, \mu B\bar{t}}\ \mu$$

We first show that $\vdash \lceil B\rceil\lceil\mu B\rceil\bar{t} \multimap \lceil\mu B\bar{t}\rceil$ has an LL^2 proof for all B and $\bar{t}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\overline{\vdash (\lceil B\rceil\lceil\mu B\rceil\bar{t})^\perp, \lceil B\rceil\lceil\mu B\rceil\bar{t}}\ \text{init} \quad \vdash ?(\exists\vec{y}\,.\,\lceil B\rceil S\vec{y} \otimes \overset{\Xi}{(S\vec{y})^\perp}), (\lceil\mu B\rceil\bar{x})^\perp, S\bar{x}}{\vdash (\lceil B\rceil\lceil\mu B\rceil\bar{t})^\perp, ?(\exists\vec{y}\,.\,\lceil B\rceil S\vec{y} \otimes (S\vec{y})^\perp), \lceil B\rceil S\bar{t}}\ \text{deep} \quad \dfrac{}{\vdash (S\bar{t})^\perp, S\bar{t}}\ \text{init}}{\vdash (\lceil B\rceil\lceil\mu B\rceil\bar{t})^\perp, ?(\exists\vec{y}\,.\,\lceil B\rceil S\vec{y} \otimes (S\vec{y})^\perp), \lceil B\rceil S\bar{t} \otimes (S\bar{t})^\perp, S\bar{t}}\ \otimes}{\vdash (\lceil B\rceil\lceil\mu B\rceil\bar{t})^\perp, ?(\exists\vec{y}\,.\,\lceil B\rceil S\vec{y} \otimes (S\vec{y})^\perp), S\bar{t}}\ C, D, \exists$$
$$\dfrac{}{\vdash (\lceil B\rceil\lceil\mu B\rceil\bar{t})^\perp, \lceil\mu B\bar{t}\rceil}\ ?, \forall, \parr$$

Here, Ξ is a straightforward LL^2 proof. Finally, using this proof of $\vdash (\lceil B \rceil \lceil \mu B \rceil \bar{t})^\perp, \lceil \mu B \bar{t} \rceil$ and the cut rule for LL^2, we have shown the soundness of the μ rule in Figure 3.2. □

Theorem 6. *μLK is consistent*

Proof. Assume that $\vdash B$ and $\vdash (B)^\perp$ have μLK proofs. By Lemma 1, we know that $\vdash \lceil B \rceil$ and $\vdash \lceil (B)^\perp \rceil$ have LL^2 proofs. While it is not the case that $\lceil (B)^\perp \rceil = (\lceil B \rceil)^\perp$, a simple induction on the structure of B shows that $\lceil (B)^\perp \rceil \vdash (\lceil B \rceil)^\perp$ is provable in LL^2. Since LL^2 has a cut rule, we know that there is an LL^2 proof of $\vdash \cdot$ (the empty sequent). By the cut-elimination theorem of LL^2, this sequent also has a cut-free LL^2 proof, which is impossible. □

3.2 Linearly and classically provable statements

The combined action of linearity on one side and the availability of inductive predicates on the other, renders the task of understanding the class of formulas that are provable in μMALL particularly difficult. The matter has been extensively analyzed in the λ-calculus literature, where it corresponds to analyzing the behavior of a recursion operator in absence of the contraction rule (see [DL05; Alv+10]), as well as more recently in the context of cyclic proof systems [KPP21].

In the original work on μMALL by David Baelde [Bae08], it was shown how one can obtain primitive recursive functions in a Curry-Howard interpretation of μMALL; the first order counterpart to that encoding was presented in subsection 3.1.2. Establishing an upper bound remains an open problem. Our attempt at tackling the question has been by comparison with μLK. We present here some results on a fragment were the two systems coincide.

When trying to compare μMALL with μLK, we find that there are a number of statements that are provable in μLK but not in μMALL since their proofs require contraction. Take for example the formula $\bot \multimap \bot \otimes \bot$ or the formula $\forall x \forall y (x = y \oplus x \neq y)$, which are provable in μLK by using contraction, but not provable in μMALL. We now wish to identify classes of formulas for which provability in μLK is conservative over μMALL: we do this by making restrictions on the polarities of the connectives that appear in formulas. Both the small counterexamples we mentioned before involved an alternation of positive and negative connectives. Our first conservativity result involves examining purely positive formulas.

Theorem 7. *Let Γ be a multiset of purely positive formulas. If $\vdash \Gamma$ has a μLK proof, then there exists a $P \in \Gamma$ such that $\vdash P$ has a $\mu MALL$ proof.*

Proof. This proof proceeds by induction on the structure of cut-free μLK proofs. Since the $\mu\nu$ rule is not applicable, the only possible base cases are the introduction rules for $=$ and 1, and, in both cases, the theorem holds immediately.

In the inductive step, consider the case of an application of the $\otimes$ rule to derive the sequent $\vdash \Gamma, \Delta, P \otimes Q$ from the premises $\vdash \Gamma, P$ and $\vdash \Delta, Q$. By the inductive hypothesis, each of these premises contains a formula that is provable in μMALL. We distinguish three different subcases depending on which of the formulas in the premises are selected.

- P and Q are selected. Then by an application of $\otimes$ we can prove in μMALL $P \otimes Q$, which appears in the endsequent.
- $R \in \Gamma$ is selected. Then R is provable in μMALL by inductive hypothesis and appears in the endsequent.
- $R \in \Delta$ is selected. As in the previous point.

The cases for introducing $\oplus, \exists, \mu$ are analogous and simpler. We are left with the cases of weakening and contraction and in these cases, the conclusion is immediate. □

A consequence of this theorem is the following. Let Q be an arithmetic formula that can be polarized as a purely positive formula $\hat{Q}$. By definition 9, if $\vdash Q$ has a proof in Peano Arithmetic then $\vdash \hat{Q}^\circ$ has a proof in μMALL. It is important to note that induction (the ν inference rule) does not play a role in this proof, since, to use induction, we require the presence of the negative connective ν.

The use of induction in μMALL allows for some formulas to be weakened and contracted. The following proposition is well known and can be proved by induction of the structure of purely negative formulas [Bae12a].

Proposition 1. The weakening and contraction rules are admissible in μMALL for purely negative formulas.

Another way to state this proposition is that the linear logic equivalence $N \multimap ?N$ holds for purely negative formulas N. Thus, expressions such as $(\mathit{nat}\ \mathbf{5})^\perp$ and $(\mathit{plus}\ n\ m\ p)^\perp$ can be used any number of times within a μMALL proof. (If we presented a two-sided sequent system for μMALL then assumptions such as $\mathit{nat}\ \mathbf{5}$ and $\mathit{plus}\ n\ m\ p$ can be used any number of times.) As a result, μMALL proofs can, occasionally, feel like working in a classical logic setting. A similar result is known for linear lambda calculi, where, in the presence of a recursor, one can obtain duplication and erasure for the type of natural numbers (but not for functions) [Alv+06].

Definition 10. A *bipolar formula* is a negative formula in which no negative connective occurrence appears in the scope of a positive connective. Thus, a bipolar formula consists of some negative top-level connectives with purely positive subformulas underneath: there is at most one alternation of polarity from negative to positive.

By employing the polarizing translation from Peano Arithmetic, we can view these classes as containing some fragments of the arithmetical hierarchy.

Proposition 2. Let P be a formula of Peano Arithmetic. Then

- If P is Σ^0_1, there is a polarization $\hat{P}^\circ$ that is purely positive.
- If P is Π^0_2, there is a polarization $\hat{P}^\circ$ that is a bipolar formula.

One should not be led into confusion by thinking that, thanks to Theorem 7, we could prove open, purely positive formulas and then strengthen them by

universal quantification in order to get stronger theorems expressing, for example, the totality of a function. For example, a formula such as $\exists x\, .plus\, a\, b\, x$ is not provable since it requires more information on the a and b variables (and the proof needs to proceed by induction on them). The provable formula that expresses totality of the plus relation is then $\forall x.\, nat\, x \supset \forall y.\, nat\, y \supset \exists u\, .plus\, x\, y\, u$, which is a bipolar formula. Note that the two examples of μLK provable formulas without μMALL proofs at the start of this section are not bipolar formulas.

When the induction rule ν is available, we will restrict occurrences of inductive invariants to be purely positive in order to prevent complex formulas from appearing in proofs. We call μLK_1 the system consisting of the same rules as μLK but where the inductive invariants are restricted to be purely positive. The notation comes from the fact that this fragment is similar to the fragment $I\Sigma_1$ of Peano Arithmetic.

We now finish this section with proving that any bipolar formula provable in μLK_1 is provable in μMALL. This conservativity result can be applied to the formulas stating the *totality* and *determinancy* properties of relations defined by purely positive fixed points are all bipolar formulas. The proof of this result would be aided greatly if we had a focusing theorem for μLK. If we take the focused proof system for μMALL given in [Bae12a; BM07a] and add contraction and weakening in the usual fashion, we have a natural candidate for a focused proof system for μLK. However, the completeness of that proof system is currently open. As Girard points out in [Gir91], the completeness of such a focused (cut-free) proof system would allow the extraction of the constructive content of classical Π^0_2 theorems, and we should not expect such a result to follow from the usual ways that we prove cut-elimination and the completeness of focusing. As a result of not possessing such a focused proof system for μLK, we must reproduce aspects of focusing in order to prove our conservation result.

Definition 11. A *reduced sequent* is a sequent that contains only purely negative, purely positive, and bipolar formulas. If Γ_1 and Γ_2 are reduced sequents, we say that Γ_1 *contains* Γ_2 if Γ_2 is a sub-multiset of Γ_1. Finally, we say that a reduced sequent is a *pointed* sequent if it contains exactly one formula that is either purely positive or bipolar.

Definition 12. A *positive region* is a cut-free μLK_1 proof that contains only the inference rules $\mu\nu$, contractions, weakening, and introduction rules for the positive connectives.

Definition 13. The $C\nu\nu$ rule is the following derived rule of inference.

$$\frac{\vdash \Gamma, S\vec{t}, U\vec{t} \quad \vdash BU\bar{x}, (U\bar{x})^\perp \quad \vdash BS\bar{x}, (S\bar{x})^\perp}{\vdash \Gamma, \nu B\vec{t}}\, \text{C}\nu\nu$$

The $C\nu\nu$ rule is justified as the following combination of ν and contraction rules.

$$\dfrac{\dfrac{\vdash \Gamma, S\vec{t}, U\vec{t} \quad \vdash BU\bar{x}, (U\bar{x})^\perp}{\vdash \Gamma, \nu B\vec{t}, S\vec{t}}\,\nu \qquad \vdash BS\bar{x}, (S\bar{x})^\perp}{\dfrac{\vdash \Gamma, \nu B\vec{t}, \nu B\vec{t}}{\vdash \Gamma, \nu B\vec{t}}\,\text{C}}\,\nu$$

Since we are working within the setting of μLK_1, the invariants S and U are purely positive.

Definition 14. A *negative region* is a cut-free μLK_1 partial proof in which the open premises are all reduced sequent and where the only inference rules are introductions for negative connectives plus the $C\nu\nu$ rule.

Lemma 2. If a reduced sequent Γ has a positive region proof then Γ contains a pointed sequent that has a μMALL proof.

Proof. This proof is a simple generalization of the proof of theorem 7. □

Lemma 3. If every premise of a negative region contains a pointed sequent with a μMALL proof, then the conclusion of the negative region contains a pointed sequent with a μMALL proof.

Proof. This proof is by induction on the height of the negative region. The most interesting case to examine is the one where the last inference rule of the negative region is the $C\nu\nu$ rule. Referring to the inference rule displayed above, the inductive hypothesis ensures that the reduced sequent $\vdash \Gamma, S\vec{t}, U\vec{t}$ contains a pointed sequent Δ, C where Δ is a multiset of purely negative formula in Γ and where the formula C (that is either purely positive or is bipolar) is either a member of Γ or is equal to either $S\vec{t}$ or $U\vec{t}$. In the first case, Δ, C is also contained in the endsequent $\Gamma, \nu B\vec{t}$. In the second case, we have one of the following proofs:

$$\frac{\vdash \Delta, S\vec{t} \quad \vdash BS\bar{x}, (S\bar{x})^{\perp}}{\vdash \Gamma, \nu B\vec{t}}\ \nu \qquad \frac{\vdash \Delta, U\vec{t} \quad \vdash BU\bar{x}, (U\bar{x})^{\perp}}{\vdash \Gamma, \nu B\vec{t}}\ \nu$$

depending on whether or not C is $S\vec{t}$ or $U\vec{t}$. □

Lemma 4. If the reduced sequent Γ has a cut-free μLK_1 proof then Γ has a proof that can be divided into a negative region that proves Γ in which all its premises have positive region proofs.

Proof. This lemma is proved by appealing to the permutation of inference rules. As is shown in [Bae12b], the introduction rules for negative connectives permute down over all inference rules in μMALL. Not considered in that paper is how such negative introduction rules permute down over contractions. It is easy to check that such permutations do, in fact, happen except in the case of the ν rule. In general, contractions below a ν rule will not permute upwards, and, as a result, the negative region is designed to include the $C\nu\nu$ rule (where contraction is stuck with the ν rule). As a result, negative rules (including $C\nu\nu$) permute down while contraction and introductions of positive connectives permute upward. This gives rise to the two-region proof structure. □

By combining the results of this section we get the following comparison between classical and linear arithmetic:

Theorem 8. *Any bipolar formula provable in μLK_1 is provable in $\mu MALL$.*

Totality statements for primitive recursive functions are included among the formulas to which Theorem 8 can be applied. Recent work, such as [KPP21], shows how allowing a more general induction rule in a linear setting, where the context is duplicated and provided to the right branch, makes it possible to capture computations that go beyond primitive recursion. Such an extension to μMALL would increase the expressiveness of the system, as for example the proof of the totality of Ackermann's function necessitates precisely similar access to the global context during the inductive proof; whether this would make the system collapse to μLK remains to be assessed.

The *focused inductive theorem proving* strategy reported in [BMS10] can now be justified (in large part) by Theorem 8: in particular, the contraction rule was not used in the search procedure reported there.

3.3 Using proof search to compute functions

One way to prove that a binary relation ϕ encodes a function is to prove the *totality* and *determinancy* properties of ϕ: that is, prove

$$[\forall x \exists y.\phi(x,y)] \wedge [\forall x \forall y_1 \forall y_2.\phi(x,y_1) \supset \phi(x,y_2) \supset y_1 = y_2].$$

Clearly, these properties imply that for every natural number x, the predicate $\lambda y.\phi(x,y)$ denotes a singleton set. If our logic contains a choice operator, such as Church's *definite description* operator ι [Chu40], then this function can be represented via the expression $\lambda x.\iota y.\phi(x,y)$. A more computationally-oriented approach to encoding such functions follows the Curry-Howard approach of relating proof theory to computation [How80]: one extracts from a natural deduction proof of $\forall x \exists y.\phi(x,y)$ a λ-term, which can be seen as an algorithm for computing the implied function. The algorithmic content of such a λ-term arises from a non-deterministic rewriting process that iteratively selects β-redexes for reduction. In most typed λ-calculus systems, all such sequences of rewritings will end in the same normal form, although some sequences of rewrites might be very long, and others can be very short. This section will describe an alternative mechanism for computing functions from their relational specification that relies on using proof search mechanisms instead of the Curry-Howard correspondence.

Note that if P and Q are predicates of arity one and if P denotes a singleton, then $\exists x[Px \wedge Qx]$ and $\forall x[Px \supset Qx]$ are logically equivalent. We assume here that Px is a purely positive expression with x as its only free variable. Notice that the proof search semantics of these equivalent formulas are surprisingly different. In particular, if we attempt to prove $\exists x[Px \wedge Qx]$, then we must *guess* a term t and then *check* that t denotes the element of the singleton (by proving $P(t)$). In contrast, if we attempt to prove $\forall x[Px \supset Qx]$ then we allocate an eigenvariable y (which we will eventually instantiate with t) and then attempt to prove the sequent $\vdash Py \supset Qy$. Such an attempt at building a proof might actually *compute* the value t (especially if we can restrict proofs of that implication to not involve the general form of induction).

Example 7. The following derivation verifying that 4 is a sum of 2 and 2.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \mathbf{2} = (s\ \mathbf{1})}= \quad \dfrac{}{\vdash \mathbf{4} = (s\ \mathbf{3})}= \quad \vdash \mathit{plus}\ \mathbf{1}\ \mathbf{2}\ \mathbf{3}}{\vdash \mathbf{2} = (s\ \mathbf{1}) \otimes \mathbf{4} = (s\ \mathbf{3}) \otimes \mathit{plus}\ \mathbf{1}\ \mathbf{2}\ \mathbf{3}}\otimes \times 2}{\vdash \exists n' \exists p' (\mathbf{2} = (s\ n') \otimes \mathbf{4} = (s\ p') \otimes \mathit{plus}\ n'\ \mathbf{2}\ p')}\exists \times 2}{\vdash (\mathbf{2} = \mathbf{0} \otimes \mathbf{2} = \mathbf{4}) \oplus \exists n' \exists p' (\mathbf{2} = (s\ n') \otimes \mathbf{4} = (s\ p') \otimes P\ n'\ \mathbf{2}\ p')}\oplus}{\vdash \mathit{plus}\ \mathbf{2}\ \mathbf{2}\ \mathbf{4}}\mu}{\vdash \exists p.\mathit{plus}\ \mathbf{2}\ \mathbf{2}\ p}\exists$$

To complete this proof, we must construct a similar subproof verifying that $1+2=3$. In particular, the witness used to instantiate the final $\exists p$ is, in fact, that sum. Unfortunately, proof construction in this system does not help us construct this sum's value. Instead, the first step in building such a proof bottom-up starts with guessing a value and checking that it is the correct sum.

Example 8. Given the definition of addition on natural numbers above, the following totality and determinancy formulas

$$[\forall x_1 \forall x_2.\ \mathit{nat}\ x_1 \supset \mathit{nat}\ x_2 \supset \exists y.(\mathit{plus}(x_1, x_2, y) \wedge \mathit{nat}\ y)]$$
$$[\forall x_1 \forall x_2.\ \mathit{nat}\ x_1 \supset \mathit{nat}\ x_2 \supset \forall y_1 \forall y_2.\ \mathit{plus}(x_1, x_2, y_1) \supset \mathit{plus}(x_1, x_2, y_2) \supset y_1 = y_2]$$

can be proved in μMALL where these formulas are polarized using the multiplicative connectives. These proofs require both induction and the $\mu\nu$ rule. Using the cut rule with (the obvious) proofs of *nat* $\mathbf{2}$ and *nat* $\mathbf{3}$, we know that $\lambda y.(\mathit{plus}\ \mathbf{2}\ \mathbf{3}\ y)$ denotes a singleton. In order to compute the sole member of the singleton $\lambda y.(\mathit{plus}\ \mathbf{2}\ \mathbf{3}\ y)$, we could perform cut-elimination with the inductively proved totality theorem in this example. Instead of such a *proof-reduction* approach to computation, the *proof search* approach starts by replacing the goal $\exists y.(\mathit{plus}\ \mathbf{2}\ \mathbf{3}\ y \wedge \mathit{nat}\ y)$ with $\forall y.(\mathit{plus}\ \mathbf{2}\ \mathbf{3}\ y \supset \mathit{nat}\ y)$. Attempting to prove this second formula leads to an incremental construction of the answer substitution for y, namely, $\mathbf{5}$.

Assume that P is a purely positive predicate expression of type $i \to o$ and that we have a μMALL proof that P is a singleton. As we stated above, this means that we have a μMALL proof of $\forall x[Px \supset \mathit{nat}\ x]$. This proof can be understood as a means to compute the unique element of P except that there might be instances of the induction rule in the proof of $\forall x[Px \supset \mathit{nat}\ x]$. Suppose we can force, however, the proof of this latter formula to be restricted so that the only form of induction is unfolding. In that case, such a restricted proof can provide an explicit computation. As the following example shows, it is not the case that if there is a μMALL proof of $\forall x[Px \supset \mathit{nat}\ x]$ then it has a proof with the induction rule replaced by unfolding.

Example 9. Let P be $\mu(\lambda R \lambda x.x = \mathbf{0} \oplus (R\ (s\ x)))$. Clearly, P denotes the singleton set containing zero. There is also a μMALL proof that $\forall x[Px \supset \mathit{nat}\ x]$, but there is no (cut-free) proof of this theorem that uses unfolding instead of the more general induction rule: just using unfoldings leads to an unbounded proof search attempt which roughly follows the following

outline.

$$\dfrac{\vdots}{\dfrac{\vdots}{\dfrac{\vdash (P\ (s\ (s\ y)))^{\perp}, \mathit{nat}\ y}{\vdash (P\ (s\ y))^{\perp}, \mathit{nat}\ y}\ \mathit{unfold}, \&, \neq \qquad \vdash \mathit{nat}\ \mathbf{0}}{\vdash (P\ y)^{\perp}, \mathit{nat}\ y}\ \mathit{unfold}, \&, \neq}$$

Although proof search can contain potentially unbounded branches, we can still use the proof search concepts of unification and non-deterministic search to compute the value within a singleton. We define a non-deterministic algorithm as follows. The *state* of this algorithm is a triple of the form

$$\langle x_1, \ldots, x_n \,;\, B_1, \ldots, B_m \,;\, t \rangle,$$

where t is a term, $B_1, \ldots, B_m$ is a multiset of purely positive formulas, and all variables free in t and in the formulas $B_1, \ldots, B_m$ are in the set of variables $x_1, \ldots, x_n$. A *success state* is one of the form $\langle \cdot \,;\, \cdot \,;\, t \rangle$ (that is, when $n = m = 0$): such a state is said to have *value* t. Given the state $S = \langle \Sigma \,;\, B_1, \ldots, B_m \,;\, t \rangle$ with $m \geq 1$, we can non-deterministically select one of the B_i formulas: for the sake of simplicity, assume that we have selected B_1. We define the transition $S \Rightarrow S'$ of state S to state S' by a case analysis of the top-level structure of B_1.

- If B_1 is $u = v$ and the terms u and v are unifiable with most general unifier θ, then we transition to $\langle \Sigma\theta \,;\, B_2\theta, \ldots, B_m\theta \,;\, t\theta \rangle$.
- If B_1 is $B \otimes B'$ then we transition to $\langle \Sigma \,;\, B, B', B_2, \ldots, B_m \,;\, t \rangle$.
- If B_1 is $B \oplus B'$ then we transition to either $\langle \Sigma \,;\, B, B_2, \ldots, B_m \,;\, t \rangle$ or $\langle \Sigma \,;\, B', B_2, \ldots, B_m \,;\, t \rangle$.
- If B_1 is $\mu B \bar{t}$ then we transition to $\langle \Sigma \,;\, B(\mu B)\bar{t}, B_2, \ldots, B_m \,;\, t \rangle$.
- If B_1 is $\exists y.\ B\ y$ then we transition to $\langle \Sigma, y \,;\, B\ y, B_2, \ldots, B_m \,;\, t \rangle$ assuming that y is not in Σ.

This non-deterministic algorithm is essentially applying left-introduction rules in a bottom-up fashion and, if there are two premises, selecting (non-deterministically) just one premise to follow.

Lemma 5. Assume that P is a purely positive expression of type $i \to o$ and that $\exists y.Py$ has a μLK proof. There is a sequence of transitions from the initial state $\langle y \,;\, P\ y \,;\, y \rangle$ to a success state with value t such that $P\ t$ has a μMALL proof.

Proof.
An *augmented state* is a structure of the form $\langle \Sigma \mid \theta \,;\, B_1 \mid \Xi_1, \ldots, B_m \mid \Xi_m \,;\, t \rangle$, where

- θ is a substitution with domain equal to Σ and which has no free variables in its range, and
- for all $i \in \{1, \ldots, m\}$, Ξ_i is a μMALL proof of $\theta(B_i)$.

Clearly, if we strike out the augmented items (in red), we are left with a regular state. Given that we have a μLK proof of $\exists y.Py$, conservativity (Theorem 7) ensures us that we have a μMALL proof of $\exists y.Py$. Thus, we there exists a μMALL proof Ξ_0 of $P\ t$ for some term t. Note that there is no occurrence of induction in Ξ_0. We now set the initial augmented state to $\langle y \mid [y \mapsto t]\ ; Py \mid \Xi_0\ ; y\rangle$. As we detail now, the proof structures Ξ_i provide oracles that steer this nondeterministic algorithm to a success state with value t. Given the augmented state $\langle \Sigma \mid \theta\ ; B_1 \mid \Xi_1, \ldots, B_m \mid \Xi_m\ ; s\rangle$, we consider selecting the first pair $B_1 \mid \Xi_1$ and consider the structure of B_1.

- If B_1 is $B' \otimes B''$ then the last inference of Ξ_1 is $\otimes$ with premises Ξ' and Ξ'', and we make a transition to $\langle \Sigma \mid \theta\ ; B' \mid \Xi', B'' \mid \Xi'', \ldots, B_m \mid \Xi_m\ ; s\rangle$.
- If B_1 is $B' \oplus B''$ then the last inference rule of Ξ_1 is $\oplus$ and that rule selects either the first or the second disjunct. In either case, let Ξ' be the proof of its premise. Depending on which of these disjuncts is selected, we make a transition to either $\langle \Sigma \mid \theta\ ; B' \mid \Xi', B_2 \mid \Xi_2, \ldots, B_m \mid \Xi_m\ ; s\rangle$ or $\langle \Sigma \mid \theta\ ; B'' \mid \Xi', B_2 \mid \Xi_2, \ldots, B_m \mid \Xi_m\ ; s\rangle$, respectively.
- If B_1 is $\mu B\bar{t}$ then the last inference rule of Ξ_1 is μ. Let Ξ' be the proof of the premise of that inference rule. We make a transition to

$$]\langle \Sigma \mid \theta\ ; B(\mu B)\bar{t} \mid \Xi', B_2 \mid \Xi_2, \ldots, B_m \mid \Xi_m\ ; s\rangle$$

  .
- If B_1 is $\exists y.\ B\ y$ then the last inference rule of Ξ_1 is $\exists$. Let r be the substitution term used to introduce this $\exists$ quantifier and let Ξ' be the proof of the premise of that inference rule. Then we make a transition to

$$\langle \Sigma, w \mid \theta \circ \varphi\ ; B\ w \mid \Xi', B_2 \mid \Xi_2, \ldots, B_m \mid \Xi_m\ ; s\rangle$$

  , where w is a variable not in Σ and φ is the substitution $[w \mapsto r]$. Here, we assume that the composition of substitutions satisfies the equation $(\theta \circ \varphi)(x) = \varphi(\theta(x))$.
- If B_1 is $u = v$ and the terms u and v are unifiable with most general unifier φ, then we make a transition to $\langle \Sigma\varphi \mid \rho\ ; \varphi(B_2) \mid \Xi_2, \ldots, \varphi(B_m) \mid \Xi_m\ ; (\varphi t)\rangle$ where ρ is the substitution such that $\theta = \varphi \circ \rho$.

In each of these cases, we must show that the transition is made to an augmented state. This is easy to show in all but the last two rules above. In the case of the transition due to $\exists$, we know that Ξ' is a proof of $\theta(B\ r)$, but that formula is simply $\varphi(\theta(B\ w))$ since w is new and r contains no variables free in Σ. In the case of the transition due to equality, we know that Ξ_1 is a proof of the formula $\theta(u = v)$ which means that θu and θv are the same terms and, hence, that u and v are unifiable and that θ is a unifier. Let φ be the most general unifier of u and v. Thus, there is a substitution ρ such that $\theta = \varphi \circ \rho$ and, for $i \in \{2, \ldots, m\}$, Ξ_i is a proof of $(\varphi \circ \rho)(B_i)$. Finally, termination of this algorithm is ensured since the number of occurrences of inference rules in the included proofs decreases at every step of the transition. Since we have shown that there is an augmented path that terminates, we have that there exists a path of states to a success state with value t. □

This lemma ensures that our search algorithm can compute a member from a non-empty set, given an μLK proof that that set is non-empty.

We can now prove the following theorem about singleton sets. We abbreviate $(\exists x.P\ x) \wedge (\forall x_1 \forall x_2.P\ x_1 \supset P\ x_2 \supset x_1 = x_2)$ by $\exists!x.P\ x$ in the following theorem.

Theorem 9. *Assume that P is a purely positive expression of type $i \to o$ and that $\exists!y.Py$ has a μLK proof. There is a sequence of transitions from the initial state $\langle y\ ;\ P\ y\ ;\ y\rangle$ to a success state of value t if and only if $P\ t$ has a μLK proof.*

Proof. Given a (cut-free) μLK proof of $\exists!y.Py$, that proof contains a μLK proof of $\exists y.Py$. Since this formula is purely positive, there is a μMALL proof for $\exists y.Py$. The forward direction is immediate: given a sequence of transitions from the initial state $\langle y\ ;\ P\ y\ ;\ y\rangle$ to the success state $\langle \cdot\ ;\ \cdot\ ;\ t\rangle$, it is easy to build a μMALL proof of $P\ t$. Conversely, assume that there is a μLK proof of $P\ t$ for some term t. By conservativity, there is a μMALL proof of $P\ t$ and, hence, of $\exists y.P\ y$. By Lemma 5, there is a sequence of transitions from initial state $\langle y\ ;\ P\ y\ ;\ y\rangle$ to the success state $\langle \cdot\ ;\ \cdot\ ;\ s\rangle$, where $P\ s$ has a μMALL proof. Given that Pt and Ps and $\forall x_1 \forall x_2.P\ x_1 \supset P\ x_2 \supset x_1 = x_2$ all have μLKp proofs, using the cut rule, we can conclude that $t = s$. □

Thus, a (naive) proof-search algorithm involving both unification and non-deterministic search is sufficient for computing the functions encoded in relations.

3.4 Reconstructing model checking arguments

We conclude the chapter by presenting a different application of the focused proof theory of μMALL, as the foundation for the reconstruction of verification arguments. This serves as the foundation for some of the developments in the next chapter.

3.4.1 Logic programming and model checking

The μMALL system can be restricted by removing (co-)induction and including instead the following rule for fixpoint expansions:

$$\frac{\vdash \Gamma, B(\nu B)\bar{t}}{\vdash \Gamma, \nu B\bar{t}}$$

Call the system μMALL^{-}. As we show now, the proof theory of logic programming with Horn clauses is completely described by using proofs of purely positive formulas in this restricted system.

The connection between Horn clauses and least fixed points is well-known and goes back to at least the *Clark completion* of Horn clauses [Cla78]. To illustrate, consider the following two Horn clauses axiomatizing addition, written using Prolog-style syntax (meaning that `:-` denotes the reverse implication $\subset$).

$$\begin{array}{rll} \forall N. & \mathit{plus}\ \mathbf{0}\ N\ N. & \\ \forall N \forall M \forall P. & \mathit{plus}\ (s\ N)\ M\ (s\ P) & \texttt{:-}\ \mathit{plus}\ N\ M\ P \end{array}$$

By moving the term structures in the head to the body of these clauses, we have the equivalent clauses:

$$\forall N \forall M \forall P. \quad \mathit{plus}\ N\ M\ P \quad \texttt{:-}\ N = \mathbf{0} \wedge M = P.$$
$$\forall N \forall M \forall P. \quad \mathit{plus}\ N\ M\ P \quad \texttt{:-}\ \exists N' \exists P'\ (N = (s\ N') \wedge P = (s\ P') \wedge \mathit{plus}\ N'\ M\ P').$$

These two clauses can now be merged into one by introducing a disjunction.

$$\forall N \forall M \forall P.\ \mathit{plus}\ N\ M\ P\ \texttt{:-}\quad (N = \mathbf{0} \wedge M = P) \vee \exists N' \exists P'\ (N = (s\ N') \wedge P = (s\ P') \wedge \mathit{plus}\ N\ M\ P).$$

We have recovered the usual fixed point expression for the definition of addition in μMALL:

$$\mathit{plus} = \mu\lambda P \lambda n \lambda m \lambda p((n = \mathbf{0} \otimes m = p) \oplus \exists n' \exists p' (n = (s\, n') \otimes p = (s\, p') \otimes P\, n'\, m\, p'))$$

In general, all Horn clauses can be rewritten in this fashion so that all the predicates they define can be expressed as purely positive expressions.

An immediate consequence of theorem 7 (section 3.2) is the fact that if such a purely positive expression has a μLK proof, then it is also provable in μMALL. It is also clear that if there exists a μMALL proof of a purely positive formula, then that proof does not use induction, and is therefore a proof in μMALL^1. Finally, given that Horn clauses can interpret Turing machines [Tä77], it is undecidable whether a purely positive expression has a μMALL proof.

Example 10 (Taken from [HM18])**.** Let the sets $A = \{\mathbf{0}, \mathbf{1}\}$ and $B = \{\mathbf{0}, \mathbf{1}, \mathbf{2}\}$ be encoded as the λ-expressions $\lambda x.\ x = \mathbf{0} \vee x = \mathbf{1}$ and $\lambda x.\ x = \mathbf{0} \vee x = \mathbf{1} \vee x = \mathbf{2}$, respectively. A polarized version of the formula $\forall x. Ax \supset Bx$ is $\forall x.[(x \neq \mathbf{0}\ \&\ x \neq \mathbf{1})\ ⅋\ (x = \mathbf{0} \oplus x = \mathbf{1} \oplus x = \mathbf{2})]$ and this formula has the following μMALL proof.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \mathbf{0} = \mathbf{0}}\,=}{\vdash \mathbf{0} = \mathbf{0} \oplus \mathbf{0} = \mathbf{1} \oplus \mathbf{0} = \mathbf{2}}\,\oplus}{\vdash x \neq \mathbf{0}, x = \mathbf{0} \oplus x = \mathbf{1} \oplus x = \mathbf{2}}\,\neq \quad \dfrac{\dfrac{\dfrac{}{\vdash \mathbf{1} = \mathbf{1}}\,=}{\vdash \mathbf{1} = \mathbf{0} \oplus \mathbf{1} = \mathbf{1} \oplus \mathbf{1} = \mathbf{2}}\,\oplus}{\vdash x \neq \mathbf{1}, x = \mathbf{0} \oplus x = \mathbf{1} \oplus x = \mathbf{2}}\,\neq}{\vdash x \neq \mathbf{0}\ \&\ x \neq \mathbf{1}, x = \mathbf{0} \oplus x = \mathbf{1} \oplus x = \mathbf{2}}\,\&}{\vdash \forall x.[(x \neq \mathbf{0}\ \&\ x \neq \mathbf{1})\ ⅋\ (x = \mathbf{0} \oplus x = \mathbf{1} \oplus x = \mathbf{2})]}\,\forall, ⅋$$

Here, the doubled horizontal line indicates that more than one inference rule is applied.

Switching to model checking terminology, proofs involving logic programs express *reachability* problems. But what about *non-reachability*? As has been shown [HSH91], the notion of *negation as finite failure* can be captured (in our setting) by building a μMALL^1 proof of a purely negative expression. Along a similar vein, the model checking problem of determining simulation and bisimulation of two transition systems is easily written as a negative formula with at most one alternation of polarities [HM18; MMP03] (assuming that the transition systems are defined using purely positive expressions).

In general, capturing simulation with proofs restricted to μMALL^1 requires that transition systems are acyclic. Capturing both non-reachability and simulation (and non-simulation) for cyclic transitions is possible by including the

induction rule, and thus allows for adding an invariant. For example, consider a graph that may contain cycles and consider a proof that there is *no* path from, say, node a to node b. This is provable by using an invariant S that encodes a connected component containing a but not b. The coinductive proof that one must then build must show that the set S is closed under one-step transitions and contains a and does not contain b. Such reasoning does not, surprisingly, need to use the $\mu\nu$ rule [HM18].

3.4.2 Property-based testing as proof reconstruction

Property-based testing (PBT) [FB97] is a technique for testing whether some piece of code satisfies specific properties established by executable specifications, by automatically generating test data and checking it against the behavior of the specification. If the condition is not met, counterexamples are produced. The absence of counterexamples of course is of no direct use for establishing a proof, but having a counterexample can be very valuable for discovering errors in what one is trying to prove in the first place. Typically, the techniques for PBT revolve around efficient generation of randomized test data and extraction of small counterexamples.

A proof-theoretic reconstruction of PBT for relational specifications is presented in [BMM19], adopting techniques from foundational proof certificates and the focused proof theory of logics with fixed points. The authors showed how to account for several features of this testing paradigm: from various *generation* strategies for input data, to *shrinking* of counterexamples, and fault localization. We present it briefly here, as it forms the foundation for the development in subsection 4.2.4.

Consider the problem of PBT for a relational specification in the form of a collection of Horn clauses. We can express these clauses as the purely positive fixed points $A_1 \dots A_n$, and the property we are checking against as B; the statement that the property holds for data meeting the specification becomes: $\forall x_1 \dots x_n, A_1 \parr \dots \parr A_m \supset B$. It is usual that the data under consideration has some assigned typing or sorting information; in the relational presentation of computations that we dealt with in this chapter, this was obtained by assuming that specific predicates like nat hold of the variables. Then, for a collection $\tau_1, \dots, \tau_n$ of data type predicates we obtain the more general formulation of the query as:

$$\forall x_1 \dots x_n. [\tau_1(x_1) \parr \dots \tau_n(x_n) \supset (A_1 \parr \dots \wedge A_m \supset B)]$$

The PBT problem asks us to find counterexamples to this. In μMALL, we can refute this formula by looking for proofs of

$$\exists x_1 \dots x_n. \left[(\tau_1(x_1) \parr \dots \tau_n(x_n) \parr A_1 \parr \dots \parr A_m) \parr B^{\perp}\right]$$

Revisit now example 10: the problem of inclusion between the two sets is expressed as $\forall x. nat\ x \parr A\ x \supset B\ x$, which can be seen as a PBT query where the data is of type nat, the specification is the membership to A and the property we test against is membership to B. The counterexample query is $\exists x. nat\ x \parr A\ x \supset B^{\perp}\ x$, but the statement holds and there are no counterexamples.

The very simple formulas for counterexample querying have a clear polarized structure: they start with a purely positive outside, with the existential quantifiers and the (purely positive) predicates for typing and specification; then, the negation of the property $B^\perp$ is a purely negative fixpoint (because it is the dual of a purely positive). This separation matches two phases that are typical in PBT: data generation and testing. In a focused setting, the data generation phase needs to make choices that necessitate external steering; the testing phase is pure checking, performed in a single asynchronous phase.

Foundational Proof Certificates can easily be employed to guide the generation part. Remember though that the generation part is split into the typing predicates and the specification the data should satisfy, with the latter in the form of an executable specification. The don't know non-determinism in this part is part of the computational paradigm, and it is likely that the user wishes this computation not to be tied to a particular certificate. Therefore we distinguish

- *Generation* proper, which consists of proving the typing judgments based on certificate information from the user.
- *Execution*, which continues the synchronous phase with no certificate information provided.
- *Testing*, performed in the final, negative phase.

Consider for example the signature containing the constants $0, S, nil, cons$. We can define predicates

$$\begin{aligned}
listnat &:= \mu\lambda L.\, \lambda x.\, (x = nil \mathbin{\dot{\vee}} (\exists r.\, \exists y.\, L\ r \mathbin{\dot{\wedge}} nat\ y \mathbin{\dot{\wedge}} x = cons\ y\ r)) \\
memb &:= \mu\lambda M.\, \lambda l.\, \lambda x.\, (\exists l_1.\, \exists y.\, l = cons\ y\ l_1 \mathbin{\dot{\wedge}} (y = x \mathbin{\dot{\vee}} M\ l_1\ x)) \\
app &:= \mu\lambda A.\, \lambda xs.\, \lambda ys.\, \lambda zs.\, ((xs = nil \mathbin{\dot{\wedge}} ys = zs) \vee \\
&\qquad \exists x_1.\, \exists xs_1.\, \exists zs_1.\, (xs = cons\ x_1\ xs_1 \mathbin{\dot{\wedge}} zs = cons\ x_1\ zs_1 \mathbin{\dot{\wedge}} A\ xs_1\ ys\ zs_1))
\end{aligned}$$

Suppose we want to check that the app definition satisfies the fact that appending a singleton yields a list containing the new element. Then we would search for counterexamples by querying $\exists l_1.\, listnat\, l_1 \exists l_2.\, listnat\, l_2 \mathbin{\dot{\wedge}} \exists x.\, nat\, x \mathbin{\dot{\wedge}} app\ l_1\ (cons\ x\ nil)\ l_2 \mathbin{\dot{\wedge}} (memb\ x\ l_2)^\perp$. The predicates that are relevant for the generation phase are here $listnat\ l_1$ and $nat\ x$. A certificate for them can contain information such as how long a list to build, or how big a number to generate. In a randomized setting, the certificate could contain the expected length for lists and a bound on the number of generation attempts. The execution phase should then correspond with the interpretation as a logic program of $app\ l_1\ (cons\ x\ nil)\ l_2$, for an appropriate substitution of l_1 and x with some generated data. This in turn generates a substitution for l_2 with the result of the computation. Finally, the testing phase consists of establishing that $memb\ x\ l_2$ does not hold for this substitution instance: since all the data is at this point is instantiated, this does not need any external evidence in order to be checked.

Chapter 4

Experiments and prototypes

Several of the ideas in this thesis were implemented in some prototypes. Part of this was used in the development of an experimental Coq tactic that leverages Coq-Elpi in order to build an elaborator for proof certificates into Coq proofs.

4.1 Proof certificates and Type Theory

We introduce here some additional background elements that form the theoretical foundation of the integration between the FPC elaborator and the Coq proof assistant. Since the foundation of Coq is intuitionistic, and so far proof certificates have been introduced only for classical logic. We take a moment to introduce LJF^a, in fig. 4.1, the intuitionistic counterpart to the LKF^a calculus. LJF^a is based on the focused sequent calculus for intuitionistic logic LJF. Compared to LKF^a (fig. 1.4), it is a two-sided calculus, where the consequent can only contain one formula. Both sides have an active area (between the arrow and the turnstile) and a storage area (the leftmost and rightmost parts). The disjunction only comes in the positive variant. The synchronous and asynchronous phases now span across both sides of the sequent. Rules on the antecedent of a sequent have a dual role to those in the consequent, so negative formulas are stored during the asynchronous phase and treated in the synchronous phase.

4.1.1 Pure Type Systems

The Coq proof assistant has a type theoretic foundation based on the Calculus of Constructions. This calculus can be presented as a generalization of the simply typed lambda calculus (that we have introduced as a syntax for formulas in subsection 1.1.1) along three axes: in addition to having terms that are parametrized by other terms (with the λ constructor), we have terms that are parametrized by types (that is, polymorphic terms); types can be parametrized by terms as well (giving dependent types); and finally, types are higher order and can be parametrized by other types. We shall give more attention to this later.

The usual logical reading of this system is through the Curry-Howard correspondence [SU06]. The core of this approach is the observation that the typing rules for lambda terms are in a one-to-one correspondence with the rules

Asynchronous rules

$$\frac{\Xi_1; \Sigma \colon \Gamma \Uparrow A \vdash B \Uparrow \quad \supset_c (\Xi_0, \Xi_1)}{\Xi_0; \Sigma \colon \Gamma \Uparrow \cdot \vdash A \supset B \Uparrow} \qquad \frac{(\Xi_1 y); \Sigma, y : \iota \colon \Gamma \Uparrow \cdot \vdash [y/x]B \Uparrow \quad \forall_c(\Xi_0, \Xi_1)}{\Xi_0; \Sigma \colon \Gamma \Uparrow \cdot \vdash \forall x.\, B \Uparrow}$$

$$\frac{\Xi_1; \Sigma \colon \Gamma \Uparrow \cdot \vdash A \Uparrow \quad \Xi_1 : \Gamma \Uparrow \cdot \vdash B \Uparrow \quad \bar{\wedge}_c (\Xi_0; \Sigma, \Xi_1, \Xi_2)}{\Xi_0; \Sigma \colon \Gamma \Uparrow \cdot \vdash A \mathbin{\bar{\wedge}} B \Uparrow} \qquad \frac{\bar{\top}_c(\Xi_0)}{\Xi_0; \Sigma \colon \Gamma \Uparrow \cdot \vdash \bar{\top} \Uparrow}$$

$$\frac{\Xi_1; \Sigma \colon \Gamma \Uparrow A, B, \Theta \vdash \mathcal{R} \quad \barwedge_c (\Xi_0, \Xi_1)}{\Xi_0; \Sigma \colon \Gamma \Uparrow A \barwedge B, \Theta \vdash \mathcal{R}} \qquad \frac{\Xi_1; \Sigma \colon \Gamma \Uparrow \Theta \vdash \mathcal{R} \quad \dagger_c(\Xi_0, \Xi_1)}{\Xi_0; \Sigma \colon \Gamma \Uparrow \dagger, \Theta \vdash \mathcal{R}}$$

$$\frac{\Xi_1; \Sigma \colon \Gamma \Uparrow A, \Theta \vdash \mathcal{R} \quad \Xi_2; \Sigma \colon \Gamma \Uparrow B, \Theta \vdash \mathcal{R} \quad \vee_c(\Xi_0, \Xi_1, \Xi_2)}{\Xi_0; \Sigma \colon \Gamma \Uparrow A \vee B, \Theta \vdash \mathcal{R}} \qquad \frac{\bot_c(\Xi_0)}{\Xi_0; \Sigma \colon \Gamma \Uparrow \bot, \Theta \vdash \mathcal{R}}$$

$$\frac{(\Xi_1 y); \Sigma, y : \iota \colon \Gamma \Uparrow [y/x]B, \Theta \vdash \mathcal{R} \quad \exists_c(\Xi_0, \Xi_1)}{\Xi_0; \Sigma \colon \Gamma \Uparrow \exists x.\, B, \Theta \vdash \mathcal{R}}$$

Synchronous Rules

$$\frac{\Xi_1; \Sigma \colon \Gamma \vdash A \Downarrow \quad \Xi_2 : \Gamma \Downarrow B \vdash \mathcal{R} \quad \supset_e (\Xi_0, \Xi_1, \Xi_2)}{\Xi_0; \Sigma \colon \Gamma \Downarrow A \supset B \vdash \mathcal{R}}$$

$$\frac{\Xi_1; \Sigma \colon \Gamma \vdash A_i \Downarrow \quad \vee_e(\Xi_0, \Xi_1, i)}{\Xi_0; \Sigma \colon \Gamma \vdash A_1 \vee A_2 \Downarrow} \qquad \frac{\Xi_1; \Sigma \colon \Gamma \Downarrow A_i \vdash \mathcal{R} \quad \bar{\wedge}_e (\Xi_0, \Xi_1, i)}{\Xi_0; \Sigma \colon \Gamma \Downarrow A_1 \mathbin{\bar{\wedge}} A_2 \vdash \mathcal{R}}$$

$$\frac{\Xi_1; \Sigma \colon \Gamma \vdash A \Downarrow \quad \Xi_2 : \Gamma \vdash B \Downarrow \quad \barwedge_e (\Xi_0, \Xi_1, \Xi_2)}{\Xi_0; \Sigma \colon \Gamma \vdash A \barwedge B \Downarrow}$$

$$\frac{\Sigma \vdash t : \iota \quad \Xi_1; \Sigma \colon \Gamma \vdash [t/x]B \Downarrow \quad \exists_e(\Xi_0, \Xi_1, t)}{\Xi_0; \Sigma \colon \Gamma \vdash \exists x.\, B \Downarrow} \qquad \frac{\dagger_e(\Xi_0)}{\Xi_0; \Sigma \colon \Gamma \vdash \dagger \Downarrow}$$

$$\frac{\Sigma \vdash t : \iota \quad \Xi_1; \Sigma \colon \Gamma \Downarrow [t/x]B \vdash \mathcal{R} \quad \forall_e(\Xi_0, \Xi_1, t)}{\Xi_0; \Sigma \colon \Gamma \Downarrow \forall x.\, B \vdash \mathcal{R}}$$

Identity rules

$$\frac{\text{initL}_e(\Xi_0)}{\Xi_0; \Sigma \colon \Gamma \Downarrow N_a \vdash N_a} \qquad \frac{(l, P_a) \in \Gamma \quad \text{initR}_e(\Xi_0, l)}{\Xi_0; \Sigma \colon \Gamma \vdash P_a \Downarrow}$$

$$\frac{\Xi_1; \Sigma \colon \Gamma \Uparrow \cdot \vdash F \Uparrow \cdot \quad \Xi_2; \Sigma \colon \Gamma \Uparrow F \vdash \cdot \Uparrow R \quad \text{cut}_e(\Xi_0, \Xi_1, \Xi_2, F)}{\Xi_0; \Sigma \colon \Gamma \Uparrow \cdot \vdash \cdot \Uparrow R}$$

Structural rules

$$\frac{l : N \in \Gamma \quad \Xi_1 : \Gamma \Downarrow N \vdash R \quad \text{decideL}_e(\Xi_0, \Xi_1, l)}{\Xi_0; \Sigma \colon \Gamma \Uparrow \cdot \vdash \cdot \Uparrow R} \qquad \frac{\Xi_1; \Sigma \colon \Gamma \vdash P \Downarrow \quad \text{decideR}_e(\Xi_0, \Xi_1)}{\Xi_0; \Sigma \colon \Gamma \Uparrow \cdot \vdash \cdot \Uparrow P}$$

$$\frac{\Xi_1; \Sigma \colon \Gamma \Uparrow P \vdash \cdot \Uparrow \mathcal{R} \quad \text{releaseL}_e(\Xi_0, \Xi_1)}{\Xi_0; \Sigma \colon \Gamma \Downarrow P \vdash \mathcal{R}} \qquad \frac{\Xi_1; \Sigma \colon \Gamma \Uparrow \cdot \vdash N \Uparrow \cdot \quad \text{releaseR}_e(\Xi_0, \Xi_1)}{\Xi_0; \Sigma \colon \Gamma \vdash N \Downarrow}$$

$$\frac{\Xi_1; \Sigma \colon l : C, \Gamma \Uparrow \Theta \vdash \mathcal{R} \quad \text{storeL}_c(\Xi_0, \Xi_1, l)}{\Xi_0; \Sigma \colon \Gamma \Uparrow C, \Theta \vdash \mathcal{R}} \qquad \frac{\Xi_1; \Sigma \colon \Gamma \Uparrow \cdot \vdash \cdot \Uparrow D \quad \text{storeR}_c(\Xi_0, \Xi_1)}{\Xi_0; \Sigma \colon \Gamma \Uparrow \cdot \vdash D \Uparrow \cdot}$$

Figure 4.1: Rules of LJF^a, a calculus with proof certificates for first order intuitionistic logic. The focused zones are between the two arrows, and the storage is at the outside. Γ is a multiset of pairs of the form $l{:}R$ where l is an index and R is a positive formula or literal, and Θ is a list of formulas. $\mathcal{R}$ is any consequent (either with the single formula stored or not).

$$\frac{}{\emptyset\ wf} \qquad \frac{\Gamma \vdash A : s \quad (x \notin Dom(\Gamma))}{\Gamma, x : A\ wf} \qquad \frac{\Gamma\ wf \quad x : A \in \Gamma}{\Gamma \vdash x : A}$$

$$\frac{\Gamma\ wf \quad (s, s') \in \mathscr{A}}{\Gamma \vdash s : s'} \qquad \frac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2 \quad (s_1, s_2, s_3) \in \mathscr{R}}{\Gamma \vdash (x\colon A)B : s_3}$$

$$\frac{\Gamma \vdash (x\colon A)B : s \quad \Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x^A.M : (x\colon A)B} \qquad \frac{\Gamma \vdash t : (x\colon A)B \quad \Gamma \vdash u : A}{\Gamma \vdash t\ u : [t/x]B}$$

Figure 4.2: PTS in the usual Natural Deduction presentation

of Natural Deduction (for example, the rules for the arrow type in subsection 1.1.1 correspond to the rules for implication in Natural Deduction). The Curry-Howard correspondence then mandates that propositions in the logic are to be regarded as types of a type theory, and the proofs of a proposition are the lambda terms that inhabit the corresponding type.

The Calculus of Constructions includes therefore a basic type $Prop$ whose elements are the propositions, that can be inhabited by proofs. At the same time, to avoid circularity a basic axiom is introduced saying that $Prop : Type$ (when we read this out loud, we find the slogan that propositions are types). What this implies is that the syntax of propositions (that inhabit the type $Prop$) should be built out of the same elements that build proofs. As a consequence, there is no longer a grammatical distinction between terms and types. A uniform treatment of the different dependency relations between terms and types is provided by presenting the system in the style of a Pure Type System. This presentation depends on a set $\mathscr{S}$ of sorts, that we write $s, s_1, s_2, \ldots$. The grammar for terms and types in Pure Type Systems is:

$$t, u, A, B := x \mid \lambda x^A.B \mid (x\colon A)B \mid t\ u \mid s$$

In addition to this, we have a relation $\mathscr{A} \subseteq \mathscr{S} \times \mathscr{S}$ of axioms and a relation $\mathscr{R} \subseteq \mathscr{S} \times \mathscr{S}$ of rules. These two sets parametrize the inference rules shown in fig. 4.2, and control which sorts can depend on which sorts by only allowing certain forms for the product $(x\colon A)B$.

A presentation of first order (minimal) logic similar to the one we gave in chapter 1 can be obtained as a PTS if we consider as the sets of sorts $\{o, i, \star\}$ with axioms $\{(o, \star), (i, \star)\}$ and rules $\{(o, o, o), (i, i, i), (i, o, o)\}$: the first rule corresponds to the $\rightarrow$ connective, and the second to the $\forall$ connective. The Calculus of Constructions is given by $\mathscr{S} = \{\star, \Box\}, \mathscr{A} = \{(\star : \Box)\}, \mathscr{R} = \{(\star, \star, \star), (\star, \Box, \Box), (\Box, \star, \star), (\Box, \Box, \Box)\}$.

4.1.2 Sequent calculus and PTS

Pure Type Systems provide a presentation of typed lambda calculi that is both flexible and elegant, and are particularly effective for representing systems with dependent types. However, when we look at them as logical systems, they are overly restrictive since they come with built-in notions of what is a proof and what is a computation, by fixing them as Natural Deduction proofs and β-reduction of lambda terms. This seems at odds with what we desire for proof certificates, where we would like to have a middle ground where the user can specify their proof format.

Sorting rules

$$\frac{}{\emptyset\ wf} \qquad \frac{\Gamma \Uparrow \cdot \vdash A : \cdot \Uparrow s \quad (x \notin Dom(\Gamma))}{\Gamma, x : A\ wf} \qquad \frac{\Gamma\ wf \quad (s, s') \in \mathscr{A}}{\Gamma \Uparrow \cdot \vdash s : \cdot \Uparrow s'}$$

$$\frac{\Gamma \Uparrow \cdot \vdash A : \cdot \Uparrow s_1 \quad \Gamma, x : A \Uparrow \cdot \vdash B : \cdot \Uparrow s_2 \quad (s_1, s_2, s_3) \in \mathscr{R}}{\Gamma \Uparrow \cdot \vdash (x\colon A)B : \cdot \Uparrow s_3}$$

Initial

$$\frac{\Gamma \Uparrow \cdot \vdash N : \cdot \Uparrow s \qquad s \in \mathscr{S}}{\Gamma \Downarrow N \vdash \epsilon : \cdot \Downarrow N}\ init$$

Product

$$\frac{\Gamma \Uparrow \cdot \vdash (x\colon A)B : \cdot \Uparrow s \quad \Gamma \Uparrow A \vdash M : B \Uparrow \cdot}{\Gamma \Uparrow \cdot \vdash \lambda x^A.M : (x\colon A)B \Uparrow \cdot}\ P_r$$

$$\frac{\Gamma \Uparrow \cdot \vdash (x\colon A)B : \cdot \Uparrow s \quad \Gamma \Uparrow \cdot \vdash t : \cdot \Uparrow A \quad \Gamma \Downarrow [t/x]B \vdash l : \cdot \Downarrow C}{\Gamma \Downarrow (x\colon A)B \vdash t :: l : \cdot \Downarrow C}\ P_l$$

Structural rules

$$\frac{\Gamma, x : A \Downarrow A \vdash l : \cdot \Downarrow B}{\Gamma, x : A \Uparrow \cdot \vdash x\ l : \cdot \Uparrow B}\ decide \qquad \frac{\Gamma, x : N \Uparrow \cdot \vdash t : A \Uparrow \cdot}{\Gamma \Uparrow N \vdash t : A \Uparrow \cdot}\ store_l$$

Figure 4.3: Sequent calculus for PTS with negative bias

Luckily, alternative presentations for PTS exist that are based on a system more similar to the sequent calculus. The landmark work of Lengrand, Dyckhoff and McKinna [LDM11] leverages Herbelin's $\bar{\lambda}$ calculus (whose typing rules are in correspondence with the rules of a sequent calculus instead of natural deduction) and builds a complete framework for proof search in Pure Type Systems. In fig. 4.3 we show a variation on this system where we employ the notation from [BNGG15]; in that paper, a λ-calculus is introduced whose typing rules are shown to be corresponding to the rules of *LJF* from [LM09].

In particular, the arising system is in correspondence with the fragment of *LJF* everything is polarized with negative bias. Note that atoms are the only thing whose polarity we can describe, since we only have products and atoms, and products are negative by virtue of being a generalization of both implication and universal quantification. The syntax of terms in this fragment is:

$$\begin{aligned} t, A, B &::= \lambda x^A.B \mid (x\colon A)B \mid x\ k \mid s \\ k &::= \epsilon \mid t :: k \mid \end{aligned}$$

Since terms are to be in one-to-one correspondence with the rules of the calculus, we have two categories of terms for the synchronous and asynchronous rules. The first line corresponds to terms proper, that are introduced by asynchronous rules, while the second corresponds to continuations, introduced by synchronous rules.

Compare now the rules for Pure Type Systems in fig. 4.3 with those of LJF^a:

- Since everything is negative in this fragment, everything that reaches the antecedent during the asynchronous phase is stored.
- There are no structural rules on the right (neither of decide, store, release): this is because since everything is negative, a synchronous phase with a formula focused in the consequent can only release and store it right away, and never needs to decide on it again. For this reason the second premise to the P_l rule automatically has A released and stored, differently from *LJF*^{a}.
- Dually, there is no release on the left since there are no positive formulas. The only way a synchronous phase with a focused formula on the left can end is with init.
- Since we are in a Pure Type System, there is no distinct syntax for terms and formulas, so there is no separate context Σ. On the other side, sorting rules are present: note that sorts always appear as stored on the right; this is because they are negative atoms as well.

The more verbose syntax with respect to the traditional presentation of PTS in natural deduction makes for a slightly different formulation of types, since the same syntax is used for both types and terms. Therefore, for example, in a context with a constant $a : Prop$ we will derive that $a\ \epsilon \to a\ \epsilon$ is a well-formed $Prop$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\emptyset\ wf}}{\Uparrow \cdot \vdash Prop : Type \Uparrow \cdot}}{(a : Prop) \Downarrow Prop \vdash \epsilon : \cdot \Downarrow Prop}\ init}{a : Prop \Uparrow \cdot \vdash a\epsilon : \cdot \Uparrow Prop} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\emptyset\ wf}}{\Uparrow \cdot \vdash Prop : Type \Uparrow \cdot}}{(a : Prop) \Downarrow Prop \vdash \epsilon : \cdot \Downarrow Prop}\ init}{a : Prop \Uparrow \cdot \vdash a\epsilon : \cdot \Uparrow Prop}}{a : Prop \Uparrow \cdot \vdash a\epsilon \to a\epsilon : Prop \Uparrow \cdot}$$

Similarly, that the identity function can be typed with this type is verified as follows:

$$\dfrac{\dfrac{\vdots}{a : Prop \Uparrow \cdot \vdash a\epsilon \to a\epsilon : Prop \Uparrow \cdot} \qquad \dfrac{\dfrac{\dfrac{\vdots}{a : Prop \Uparrow \cdot \vdash a\epsilon : \cdot \Uparrow Prop}}{a : Prop, x : a\epsilon \Downarrow a\epsilon \vdash \epsilon : \cdot \Downarrow a\epsilon}\ init}{a : Prop, x : a\epsilon \Uparrow \cdot \vdash x : a\epsilon \Uparrow \cdot}\ decide}{a : Prop \Uparrow \cdot \vdash \lambda x^{a\epsilon}.x\epsilon : (a\epsilon \to a\epsilon) \Uparrow \cdot}\ P_r$$

In general, instances are always fully applied to a list of arguments, that could be the empty list. When no ambiguity is present, we write a in place of $a\ \epsilon$.

4.1.3 The Calculus of Inductive Constructions

The logic that Coq implements is an extension of the Calculus of Constructions that includes inductive type definitions, called the Calculus of Inductive Constructions (CiC). In this work we consider a fragment of CiC without induction principles, that we describe in the next paragraph. In addition to this, we place some restrictions also on the presentation of CoC that is implemented in Coq. The sorts in Coq are $Prop, SProp, Set$ and a numbered infinity of sorts

$Type(i)$ for any integer $i \geq 1$; roughly, they are the sorts of propositions, of proof-irrelevant propositions, of data types and then $Type(i)$ is a hierarchy of universes. We won't consider the hierarchy, nor proof-irrelevance, and are left with $\mathscr{S} := \{Prop, Set, Type\}$. The rules for the formation of the dependent product are those we presented in subsection 4.1.1 (where $\star$ is $Prop$ and $\Box$ is $Type$), with the addition that elements of Set can't be formed by abstracting over $Type$: $\mathscr{A} = \{(Prop : Type), (Set : Type)\}, \mathscr{R} = \{(s, Prop, Prop) \mid s \in \{Set, Prop, Type\}\} \cup \{(s, Set, Set) \mid s \in \{Prop, Set\}\} \cup \{(s, Type, Type) \mid s \in \{Prop, Type\}\}$. Finally, Coq maintains a notion of global context and one of local context. The local context Γ is used inside the development of a proof, while the global context E contains user provided assumptions or definitions for constants. The only moment where we access the global environment in this work is when dealing with inductive types, therefore we left them out of the presentation of the system in fig. 4.7.

Definitions for inductive types are included in E as constants that extend the base signature of the type theory, and are paired with their constructors; the system then generates additional constants that represent the destructors, or induction principles, for these types. Moreover, an inductive definition can contain several, mutually defined, inductive types; an inductive definition is represented as $\mathrm{Ind}[p]\,(\Gamma_I := \Gamma_C)$, where p is the number of parameters, Γ_I is the set of the newly defined types $(a_i : A_i)$, and Γ_C is the set of their constructors $(c_i : C_i)$. As an example, consider a definition of the inductive type of lists of an arbitrary type A: there is one parameter $(A : Set)$, the newly introduced type is one $(List : Set \to Set)$ and there are two constructors. The definition of the inductive type of lists augments the base signature with the following:

$$\mathrm{Ind}[1]\left(List : Set \to Set := \begin{array}{rcl} Nil & : & (A\colon Set)List\ A :: \epsilon \\ cons & : & (A\colon Set)A\ \epsilon \to List\ A :: \epsilon \to List\ A :: \epsilon \end{array}\right)$$

The global context E is built out of such inductive definitions, and we write $E[\Gamma]$ for the union of the local context Γ and E. The two typing rules for inductive types and constructors presented in Coq's reference manual are:

$$\frac{Ind[p](\Gamma_I := \Gamma_C) \in E \quad (a : A) \in \Gamma_I}{E[\Gamma] \vdash a : A} \qquad \frac{Ind[p](\Gamma_I := \Gamma_C) \in E \quad (c : C) \in \Gamma_C}{E[\Gamma] \vdash c : C}$$

These rules are expressed in natural deduction style. We formulate rules for these constants as an extension of the sequent calculus of fig. 4.3. Fix a global context E, then we have the following two rules that correspond to unfolding the inductive definition:

$$\frac{\mathrm{Ind}[p]\,(\Gamma_I := \Gamma_C) \in E \quad (A : T) \in \Gamma_I \quad c : D \in \Gamma_C \quad \Gamma \Downarrow D \vdash l : \cdot \Downarrow A}{\Gamma \Uparrow \cdot \vdash c\ l : \cdot \Uparrow A}$$

$$\frac{\mathrm{Ind}[p]\,(\Gamma_I := \Gamma_C) \in E \quad (A : T) \in \Gamma_I \quad \Gamma \Downarrow T \vdash l : \cdot \Downarrow s}{\Gamma \Uparrow \cdot \vdash A\ l : \cdot \Uparrow s}$$

Figure 4.4: Unfolding rules for inductive definitions

Finally, it is worth noting that the logical connectives $\wedge, \vee, \exists$ are given in Coq as inductive types over the type of propositions. The unfolding of these

definitions through the rules in fig. 4.4 gives rise to the usual right rules for $\bar{\wedge}$, $\dot{\vee}$ and $\exists$. Consider for example the inductive definition for the $\exists$ connective:

$$\mathrm{Ind}[2]\Big(\exists : (A\colon Type)(P\colon A \to Prop)Prop := \\ \exists_{intro} : (x\colon A)(Px \to \exists A\ \epsilon :: P\ \epsilon :: \epsilon)\Big)$$

Just as in subsection 1.1.1, the binder for the existential is handled by a more generic binding device. When the type T is unambiguous, the concrete syntax for $\exists\ A\ \epsilon\ ::\ P\ \epsilon\ ::\ \epsilon$ is $\exists y.\, P$. This inductive definition gives rise to the unfolding rule for its sole constructor (where the subproofs for checking that the products are well-sorted are omitted, and we use the concrete syntax $\exists y.\, P$):

$$\dfrac{\dfrac{\vdots}{\Gamma \Uparrow \cdot \vdash t : \cdot \Uparrow A} \quad \dfrac{\dfrac{\vdots}{\Gamma \Uparrow \cdot \vdash u : \cdot \Uparrow Pt} \quad \dfrac{\dfrac{\Pi}{\Gamma \Uparrow \cdot \vdash \exists\ A\ \epsilon :: P\ \epsilon :: \epsilon : \cdot \Uparrow Prop}}{\Gamma \Downarrow \exists y.\, Py \vdash \epsilon : \cdot \Downarrow \exists y.\, P}\ init}{\Gamma \Downarrow [x/t](Px \to \exists y.\, Py) \vdash u :: \epsilon : \cdot \Downarrow \exists y.\, P}}{\dfrac{\Gamma \Downarrow (x\colon A)(Px \to \exists y.\, P) \vdash t :: u :: \epsilon : \cdot \Downarrow \exists y.\, P}{\Gamma \Uparrow \cdot \vdash \exists_{intro}\, t :: u :: \epsilon : \cdot \Uparrow \exists y.\, P}\ \text{unfold}}$$

This is the usual right introduction rule for $\exists$, similar to the one of *LJF*: we need to provide a term t of type A and a proof of Pt in order to conclude the existential. The signature check for the term t is now encoded in the same sequent calculus as the logic. The second unfolding rule concerns the check for well-sortedness of the formulas using the connective, and can be used directly for the premise Π:

$$\dfrac{\dfrac{\Gamma \Uparrow \cdot \vdash A\ \epsilon : \cdot \Uparrow Type \quad \dfrac{\Gamma \Uparrow \cdot \vdash P\ \epsilon : \cdot \Uparrow A\ \epsilon \to Prop \quad \dfrac{\dfrac{\emptyset\ wf}{\Uparrow \cdot \vdash Prop : \cdot \Uparrow Type}}{\Gamma \Downarrow Prop \vdash \epsilon : \cdot \Downarrow Prop}}{\Gamma \Downarrow (P\colon A\ \epsilon \to Prop)Prop \vdash P\ \epsilon :: \epsilon : \cdot \Downarrow Prop}}{\Gamma \Downarrow (A\colon Type)(P\colon A \to Prop)Prop \vdash A\ \epsilon :: P\ \epsilon :: \epsilon : \cdot \Downarrow Prop}}{\Gamma \Uparrow \cdot \vdash \exists\ A\ \epsilon :: P\ \epsilon :: \epsilon : \cdot \Uparrow Prop}\ \text{unfold}$$

In order to conclude the two open premises of this proof, we need the context Γ to contain the judgments $A : Type$ and $P : A\ \epsilon \to Prop$.

4.2 Implementing an elaborator for proof certificates

In this section we detail the implementation of two Coq plugins based on the proposed sequent calculus. So far we presented all of our development in a logic programming paradigm, whereas most proof assistants (including Coq) adopt predominantly functional programming languages and tools. For this reason, we start with a general discussion on the role of logic programming in proof assistants. We then present the tooling for writing logic programming tactics in Coq with Coq-Elpi, and then we present the implementations based on Foundational Proof Certificates.

4.2.1 Logic programming and proof assistants

Several automation components that are present in proof assistants employ mechanism that resemble logic programming. Perhaps the most known cases are simple automation tactics, like Coq's `auto` and `eauto`, which are usually presented as solvers that will perform Prolog-like computation. However, the presence of logic programming computations goes much further than this, as they underlay several other components like typeclass or coercion resolution: this is true to the extent that there has been an attempt to reuse Coq's typeclass resolution component as an engine for an improved version of the `auto` tactic itself [ZH17].

The resolution of typeclasses and coercions happens in the component of a proof assistant that is responsible for synthesizing a fully explicit, complete proof term that the kernel can check, starting from the ambiguous and possibly partial proof scripts that the user has typed in. This component, named the elaborator, is by itself another typical instance of a logic program, that has to deal with backtracking search and unification. The Elpi dialect of λProlog [Dun+15] was born precisely with the aim of providing a flexible language to specify the different parts of Coq's elaborator.

So far, we have referred to our implementations of LKF^a as the *kernel* of our proof-checking system. In the context of a proof assistant however the aim of an FPC checker should not be that of replacing its kernel: a proof object always reaches this stage in a specific, maximally explicit, format. Allowing more would be meaningful only in the context of a hypothetical fully format-agnostic proof assistant, which is something that is not even clearly specified. Rather, we see more clearly the role of FPC checkers in the context of proof elaboration: given a proof certificate in any FPC specification, our checker should help rebuild a proof in the format assumed by the proof assistant and make it available to the user for interaction in their workflow.

4.2.2 Tactics in Coq-Elpi

Coq-Elpi [Tas18] is a plugin for Coq that provides an API for manipulating Coq's environment with λProlog. It also provides the possibility to build tactics that can be called when in proof mode, have access to the currently open goals, and can solve them completely or progress on them by possibly opening new goals. The embedding of terms in λProlog takes advantage of the native λProlog constructs such as lists and binders; moreover, logic variables in λProlog directly correspond to Existential Variables in Coq. The following is part of the Coq-Elpi API signature of constants we use:

Listing 4.1: Signature for the main components of Coq-Elpi's term representation

```
kind term   type.                % reification of Coq terms
kind gref   type.                % references to global terms
type app    list term → term.    % n-ary application
type prod   name → term → (term → term) → term. % products
type fun    name → term → (term → term) → term. % λ-abstraction
type global gref → term.         % terms from global context
```

```
type indt   inductive → gref.   % inductive types in glob. ctx
type indc   constructor → gref. % constructors of ind. types
```

The representation of terms resembles more the Sequent Calculus presentation that we have discussed in the previous section rather than the Natural Deduction one: `app` is an applicative list, and not an application between two terms like in Natural Deduction. Nevertheless, in the case the list of arguments of a term `t` is empty, the term is represented as `t` rather than `app [t|nil]`. Note that `prod` and `fun` encode dependent products and functions by taking a name for pretty printing, a term for the abstracted type, and a λProlog abstraction from terms to terms: i.e., $(x : B)D$ is encoded by `prod ``x'' B (x\ D x)`.

Two types of extensions can be developed with Coq-Elpi: tactics and commands. Commands are used outside the proof mode, and have access to the global state of the prover. Tactics are used during proof mode and have additional access to the currently unsolved goals and their context.

Listing 4.2: Signature for tactics

```
type solve goal → list sealed-goal → prop.
type goal goal-ctx → term → term → term → list argument → goal.
```

They are implemented by providing a definition of the `solve` predicates, that has type `goal → sealed-goal → prop`; a sealed goal is just a goal where all variables are locally bound, and `solve` transforms the currently open goal into a list of new goals. The only constructor for the type `goal` is used with: `goal Ctx RawSolution Ty Solution Arguments` where `Ctx` is the context, `Ty` is the type that needs to be inhabited, `Solution` is the variable that should be assigned with the proof term. `RawSolution` has some additional syntactic constraints, and `Arguments` carries some additional information, but we won't be using them. A trivial tactic that simply prints out the current goal, and leaves the state unchanged is implemented like this:

```
Elpi Tactic show.
Elpi Accumulate lp:{{
  solve Goal _ :-
    Goal = (goal Ctx _R Type _Sol _Args),
    coq.say Ctx Goal.
}}
Elpi Typecheck.
```

This code is used like this:

```
Lemma foo: ∀ A, A → A.
elpi show.
```

```
Lemma foo: ∀ A, A → A.
intros.
elpi show.
```

With the code on the left column, the tactic will print this line:

```
[] (prod `A` (sort (typ «Top.2»)) c0 \ prod `_` c0 c1 \ c0)
```

Whereas with the one in the right column, it will print this line:

```
[decl c1 `X` c0, decl c0 `A` (sort (typ «Top.2»))] c0
```

In the first case the context is empty, the goal is the entire type. After we introduce the hypotheses with `intros`, the context contains the type declaration and names for the two λProlog eigenvariables `c0` (of type `Type` in a certain

universe, and named A) and `c1` (of type `c0`, and named X). The goal is A, that is, the eigenvariable `c0`.

For a more interesting example, let's implement a tactic that performs bounded goal directed proof search, mimicking the behavior of Coq's `eauto` tactic. This prolog-like search is obtained by considering a fragment of the sequent calculus in fig. 4.3: we restrict the goal to be atomic, we don't perform the check for well-sortedness (and instead rely on Coq's unification when applying init), we include the rule for unfolding constructors of inductive types from fig. 4.4 and add an extra case for proof of equality by reflexivity. The result is the code in listing 4.3, where the asynchronous sequents are simplified as goal-reduction (`go`) and the synchronous ones are the backchaining sequents (`bc`). No storage is present since there are only atoms on the consequent of sequents.

Listing 4.3: A goal-directed, bounded proof search tactic in Coq-Elpi

```
%% Goal reduction
check Cert (go {{lp:G1 = lp:G2}} {{eq_refl}}):-
  coq.unify-eq G1 G2 ok,
  eq_expert Cert.
% Unfold inductive goals
check Cert (go Atom Term) :-
  coq.safe-dest-app Atom (global (indt Prog)) _Args,
  coq.env.indt Prog _ _ _ _Type Kn KnTypes,
  Kons = global (indc K),
  std.mem Kn K,
  % Use the selected constructor as key to find its
  % clause in the zipped list of constructors and clauses.
  std.lookup {std.zip Kn KnTypes} K Clause,
  Cert > 0, Cert' is Cert - 1,
  check Cert' (bc Clause Atom ListArgs),
  coq.mk-app Kons ListArgs Term.
% Weak head reduction
check Cert (go (app [(fun Name Type Body)| Args]) Term) :-
  coq.mk-app (fun Name Type Body) Args App,
  check Cert (go App Term).

%% Backchain
check _ (bc A A' []) :-
  coq.unify-eq A A' ok.

check Cert (bc (prod _ Ty1 Ty2) Goal OutTerm) :-
  check Cert (bc (Ty2 Tm) Goal ListArgs),
  coq.typecheck Tm Ty1 ok,
  OutTerm = [Tm|ListArgs].
```

The second parameter to the `check` predicate is the sequent as we have described it, while its first parameter is the user provided bound, that decreases every time a decision on a constructor is made. The unfolding rule makes use of the Coq-Elpi primitives `coq.safe-dest-app` in order to obtain the head term of a (possibly nested) application, and `coq.env.indt` in order to access

the global context of inductive definitions and query for information about them. Then, a constructor is non-deterministically selected together with its type, and the backchaining phase is initiated.

While no special rule is present for the existential quantifier, remember that $\exists$ is given in Coq as an inductive type with two parameters (the type it quantifies over and the quantified proposition). The unfolding rule treats it by backchaining on its only constructor, which has type $(x\colon A)P\ x \to \exists y.\ P\ y$. The backchain rule introduces fresh variables `Tm` for every dependent product, obtaining the behavior of `eauto`, which would have similarly introduced a Coq Existential variable.

This code is packaged in an Elpi tactic with the following glue code:

```
Elpi Tactic dprolog.
Elpi Accumulate lp:{{
solve (goal _Ctx _RawEv Goal _Ev [int N] as G) OutGoals :-
  check N (go Goal Term),
  refine Term G OutGoals.
}}.
```

The `solve` predicate simply calls `check` to synthesize a term of type `Goal`, and the Coq-Elpi built-in predicate `refine` is then used to try to close the goal. Consider a predicate that defines insertions on lists of naturals in relational style as follows:

```
Inductive insert (x:nat) : list nat → list nat → Prop :=
 i_n : insert x [] [x]
 | i_s : ∀ y: nat, ∀ ys: list nat, x ≤ y →
       insert x (y :: ys) (x :: y :: ys)
 | i_c : ∀ y: nat, ∀ ys: list nat, ∀ rs: list nat,
       y ≤ x → insert x ys rs → insert x (y :: ys) (y :: rs).
```

This tactic can solve some simple queries such as:

```
Lemma i1: ∃ R, insert 2 ([0; 1]) R.
 elpi dprolog 5.
Qed.
```

Additional features of `eauto` such as hints and databases of hints are also easily accounted for, by leveraging the `Database` feature of Coq-Elpi which allows building pieces of λProlog code that are conditionally loaded: we would then have an additional goal-reduction rule that starts a backchaining phase by selecting a clause from the available hints.

4.2.3 Elaborators for Foundational Proof Certificates

The size bound parameter to the `dprolog` tactic can already be seen as a small instance of an FPC elaborator for the restricted backchaining calculus. It is not by chance that the height parameter was named `Cert`. When we wish to extend this approach to encompass more complex certificates and more of the type theory, there are several directions that we can take. For one, the logic we have discussed so far is first order and distinguishes the clerks and experts for implication from those for universal quantification. Both are here

subsumed by the dependent product. Additionally, the calculus of fig. 4.3 only has negative rules, and lacks some structural rules of LJF^a.

The most direct generalization is obtained by augmenting the calculus composed by the rules in fig. 4.3 with clerks and experts handling proof certificates for each rule in the initial, product and structural groups. The signatures for such clerks and experts is presented in listing 4.4.

Listing 4.4: Clerks and experts for the calculus in fig. 4.3

```
type storeL_jc          cert → (A → cert) → index → prop.
type decideL_je         cert → cert → index → prop.
type initialL_je        cert → prop.
type prod_jc            cert → (A → cert) → prop.
type prod_je            cert → cert → cert → prop.
```

No clerk nor expert is present for the sorting rules: the certificate information here is provided by the set $\mathscr{A}, \mathscr{R}$ of axioms and rules. In addition to this we can include an expert for the unfolding rules in fig. 4.4:

```
type unfoldL_je     list constructor → cert →
                                  cert → constructor → prop.
```

This small set of clerks and experts gives us a playground for more expressive certificates, although still limited to have a purely negative polarity. Simple examples for this are (in addition to the usual depth or size bounds) different representations of λ-terms. In the example code at `https://github.com/proofcert/fpc-elpi` a tactic for elaborating terms with de Bruijn indexes instead of bound variables is presented.

In order to treat more proofs coming from FPC specifications in the style of LJF^a, we need some way to treat connectives and positive polarities. There is no way around having positive polarities, since the existential quantifier and the intuitionistic disjunction are positive. However, the calculus we have chosen both as a foundation for certificates and as a representation of Coq terms does not allow positives: this is enforced in the P_l rule, where we expect to build a list of arguments; if B is some form of positive type, it must be immediately decomposed on the left, and thus the typed term can't be the list l but would instead be the term corresponding with the typing rule for B (for example, an injection if B is a disjunction).

We can recover positive polarities restricted to first-order by decoupling the term representation and the foundation for proof certificates. We obtain a version of LJF^a enriched with Coq terms in natural deduction style. The rules for universal quantification and implication are separated, but they are annotated with the same Coq terms, and are no longer tied to the negative polarity assumptions:

$$\frac{\Xi_1 : \Gamma \Downarrow \cdot \vdash t : A \Downarrow \cdot \quad \Xi_2 : \Gamma \Downarrow y : B \vdash u : \cdot \Downarrow C \quad \supset_e (\Xi_0, \Xi_1, \Xi_2)}{\Xi_0 : \Gamma \Downarrow z : A \supset B \vdash [z\,t/y]u : \cdot \Downarrow C}$$

$$\frac{\Xi_1 : \Gamma \Downarrow \cdot \vdash t : A \Downarrow \cdot \quad \Xi_2 : \Gamma \Downarrow y : [t/x]A \vdash u : \cdot \Downarrow C \quad \forall_e(\Xi_0, \Xi_1, t)}{\Xi_0 : \Gamma \Downarrow z : \forall x.\, A \vdash [z\,t/y]u : \cdot \Downarrow C}$$

Now u can be a term of any shape, and thus positive polarities are allowed. Since the signature for the terms instantiated by the quantifiers is now the same

as the one for terms, which is in turn the internal representation of proofs, we see in the two expert predicates two different takes in action for a similar task: in the first, a certificate is provided to guide the proof of A and the simultaneous synthesis of t; in the second, a representation of t is directly present in the certificate. Both approaches can be meaningful in a dependently typed setting, although the first is more general and can encompass the second. The structural rules of *LKF*^{a} that are not present in fig. 4.3 will thread through the unmodified Coq term. For example:

$$\frac{\Xi_1 ; : \Gamma \Uparrow \cdot \vdash t : N \Uparrow \cdot \quad \text{release}\mathrm{R}_e(\Xi_0, \Xi_1)}{\Xi_0 ; : \Gamma \vdash t : N \Downarrow}$$

By decoupling the internal term representation from the proof checker we can elaborate proofs in first-order logic containing positive polarities into Coq proofs. In this way, we get a consistent treatment of positive connectives as well. To obtain the left rules for the connectives, we need to consider the induction principles associated with them. The induction principles are automatically derived by Coq for every inductive definition, and are inhabited by proof terms that perform recursion with case analysis. We don't look into the structure of these terms, but just at their types. Consider the induction principle `or_ind` for the inductively defined disjunction:

```
Coq < About or_ind.
or_ind : forall [A B P : Prop], (A→ P)→ (B→ P)→ A \/ B→ P
```

In Coq's concrete syntax, we get something which corresponds to the Natural Deduction rule for disjunction elimination: P is provable if we can prove it independently of the assumption A and from the assumption B, and we can prove $A \vee B$. We can use this in the *LKF*^{a} rule for the left introduction of disjunction in the following way:

$$\frac{\Xi : \Gamma \Uparrow A, \Theta \vdash u_1 : \cdot \Uparrow C \quad \Xi : \Gamma \Uparrow B, \Theta \vdash u_2 : \cdot \Uparrow C \quad \vee_c(\Xi_0, \Xi_1, \Xi_2)}{\Xi_0 : \Gamma \Uparrow x : A \vee B, \Theta \vdash or_{ind}\ A\ B\ C\ u_1\ u_2\ x : \cdot \Uparrow C}$$

This allows us to have a full elaborator for *LJF*^{a} proof certificates into Coq proofs.

4.2.4 An Elpi tactic for PBT

The second prototype implementation is a lightweight plugin that employs proof certificates to guide the generation of data for Property-Based Testing of relational specifications, with the strategy introduced in subsection 3.4.2. Given the common foundation, it is in principle possible to port all the PBT features that are accounted for in [BMM19]. However, we will now present only FPC corresponding to different flavors of *exhaustive* generation, as adopted, e.g., in SmallCheck [RNL08] and αCheck [CM17].

Coq already features QuickChick [Par+15], which is a sophisticated and well-supported PBT tool, based on a different perspective: being a clone of Haskell's QuickCheck, it emphasizes testing executable specifications with functional, decidable generators. While current research [LPP18] aims to increase

```
type interp term → o.
type backchain term → term → o.

interp {{True}}.
interp (sort _).

interp {{lp:G1 /\ lp:G2}} :-
        interp G1,
        interp G2.

interp {{lp:G1 \/ lp:G2}} :-
        interp G1;
        interp G2.

interp {{lp:G1 = lp:G2}} :-
  coq.unify-eq G1 G2 ok.

interp (app [global (indt Prog) | _Args] as Atom) :-
  coq.env.indt Prog _ _ _ _Type _Kn Clauses,
        std.mem Clauses D,
        get_head D Atom,
        backchain D Atom.

backchain (prod _ Ty (x\P)) G :- !, backchain P G, interp Ty.
backchain (prod _ _Ty (x\P x)) G :- !, backchain (P X_) G.
backchain G G' :- coq.unify-eq G G' ok.
```

Figure 4.5: Meta interpreter for PBT

automation, it is fair to say that testing with QuickChick is still very labor-intensive, in particular when it comes to relational specifications. We do not intend to compete with QuickChick at this stage, but we shall see that we can test immediately `Inductive` definitions that corresponds to pure Horn programs, without having to provide a decidability proof for those definitions. Furthermore, we are not committed to a fixed random generation strategy, which in general requires additional work in the configuration of generators and shrinkers.

Recall the setting we described for PBT in subsection 3.4.2: we have a data type τ, an executable specification P and a property Q that we wish the specification to meet. Looking for a counterexample then means looking for an existential witness for the formula

$$\exists x[(\tau(x) \wedge P(x)) \wedge \neg Q(x)]$$

In our Coq setting P and Q will be propositions, while τ is a datatype. We generate data for the types τ, while predicates will only be checked.

The intended usage of the plugin we develop is to be invoked inside a proof context, where the goal the user is trying to prove is the property Q that the specification should meet. At this point, the typing judgments for the variables

and the specification information are part of the context. Invoking the plugin will try to find counterexamples to Q based on the input parameters taken from the context.

The user specifies which variables of the environment should be used for generating data and which for executing the specification. In addition to this, the user should specify all the certificate information that will guide the data generation phase. For a context where the wished specification formulas are $H_1 : A_1 \wedge \cdots \wedge H_m : A_m$ and where the variables for which generation is needed according to `cert` are $x_1 : \tau_1, \ldots x_n : \tau_n$, the call to the tactic will be:

elpi dep_pbt <cert> $(H_1 \wedge \cdots \wedge H_m)$ $(x_1) \ldots (x_n)$**.**

The implementation of the tactic is structured as follows:

- First, a pre-processing step retrieves the variables in the context and pairs them with λProlog metavariables. Then, they are substituted with their metavariable whenever they appear in the specification $A_1, \ldots A_m$ or in the property (that is, the goal formula) Q. From a logical viewpoint, this corresponds to turning the entire context from universally quantified to existentially quantified.

- Then, the types of the variables $x_1, \ldots, x_n$ that were marked for data generation are retrieved. Goals for the FPC elaborator are created by pairing their corresponding metavariables `X1`,...,`Xn` (that were introduced at the previous point) with their types: `X1:`τ_1,...,`Xn:`τ_n. The FPC elaborator is called on these goals paired with the user provided certificate.

- If the call to the elaborator has been successful, `X1`,...,`Xn` are now instantiated, including in the specification $A_1, \ldots, A_m$ that had been generalized in the first step. The interpreter of fig. 4.5 is run on the specification, obtaining further instantiated values for the remaining metavariables.

- Finally, we have a completely ground instance of the goal formula Q. Since it is completely ground, once we see it as a Horn clause attempting to negate it corresponds to its negation-as-failure. Therefore, we call `not (interp Q)`: if this succeeds, a counterexample has been found and the substitution instances for `X1`,...,`Xn` are returned to the user. Otherwise, backtracking is automatically triggered and new data is generated if the backtracking points allowed by the certificate are not completely exhausted.

The preprocessing steps take a big advantage from the handling of λ-tree syntax in λProlog. The substitutions in terms are performed with different sets of `copy` clauses, and logic metavariables are used in place of the existentially quantified variables. Consider now for a first example this definition of ordered lists:

```
Inductive ordered : list nat → Prop :=
| o_n : ordered []
| o_s : ∀ x : nat, ordered [x]
| o_c : ∀ (x y : nat), ∀ xs, ordered xs →
                    x ≤ y → ordered (x :: y :: xs).
```

A property we may wish to check before embarking in a formal proof is whether insertion preserves order:

```
Conjecture ins_ord: ∀ (x : nat) xs rs, ordered xs →
                insert x xs rs → ordered rs.
intros x xs rs Ho Hi.
elpi dep_pbt height 5 (Ho ∧ Hi ) (x) (xs).
Abort.
```

In this query the tactic tests the hypotheses `Ho` and `Hi` against data `x,xs` generated exhaustively up to height at most 5 from the library `Inductive` definitions of `nat` and `list`. We do not generate values for `rs`, since by mode information we know that it will be computed. Since we did slip in an error, our tactic reports a counter-example, namely `x = 0` and `xs = [0; 1; 0]`. As the latter is definitely not an ordered list, this points to a quite evident bug in the definition of `ordered`. We leave it to the reader, and we abort the proof.

Testing the above conjecture with QuickChick would have required more setup: if we wished to proceed relationally as above, we would have had to provide a proof of decidability of the relevant notions. Were we to use functions, then we would have to implement a generator and shrinker for ordered lists, since automatic derivation of the former does not (yet) work for this kind of specification.

For a more significant case study, let us turn to the semantics of programming languages, where PBT has been used extensively and successfully [Kle+12]. Here we will consider a far simpler example, a small typed arithmetic language featuring numerals with predecessor and test-for-zero, and Booleans with if-then-else, which comes from the *Software Foundations* book series [Pie22]. The terms for the language are given by:

```
Inductive tm : Type :=
  | ttrue : tm | tfalse : tm | tif : tm→ tm→ tm→ tm | tzero
      : tm | tsucc : tm→ tm
  | tpred : tm→ tm | tiszero : tm→ tm.
Inductive typ : Type :=
  | TBool : typ | TNat : typ.
```

The completely standard static and small step dynamic semantics rules are reported in fig. 4.6. We pose ourselves the task of proving the subject expansion property for this calculus; it seems likely that it doesn't hold, so we call the PBT tactic:

```
Conjecture subexp:  forall e e' t, e ==>e' → has_type e' t →
    has_type e t.
 intros e e' t HS HT.
 elpi dep_pbt height 2 (HS /\ HT) (e).
```

We generate a term `e`, perform a step and obtain the type of the reduced term. Then, it is checked whether `e` has this same type. Indeed, we obtain a counterexample assignment: `e = tif ttrue tzero ttrue`, which is not well-typed but reduces to a term of type `TNat`.

Another way to assess the fault detection capability of a PBT setup is via mutation analysis [CCM20], whereby localized bugs are voluntarily inserted,

with the view that they should be caught by a well-prepared testing suite. Following on an exercise in the aforementioned chapter of Software Foundations, we modify the typing relation by adding the following (nonsensical) clause:

```
Module M1.
Inductive has_type : tm → typ → Prop :=
$\dots$
  | T_SuccBool : forall t, has_type t TBool →
                 has_type (tsucc t) TBool.
end M1.
```

Some of the required properties for the system under test now fail: not only type uniqueness, but also progress:

```
Definition progress (e :tm) (Has_type : tm → typ → Prop)
  (Step : tm → tm → Prop):= forall t, Has_type e t → notstuck
      e Step.

Conjecture progress_m1:  forall e, progress e M1.has_type step.
  unfold progress.
  intros e t Ht.
  elpi dep_pbt height 2 (Ht) (e) .
```

The inserted bug is immediately located, and we obtain the counterexample assignment `e = tsucc ttrue`.

These examples are quite simple, but have already been used in the literature as a benchmark for evaluating QuickChick's generators [Lam+17]. An important part of Property-Based Testing is the random generation of data. Here, we just discussed minimal formats of certificates providing a mere size bound, since the focus was the integration of the tool with Coq; however, the underlying implementation technique is the same as that in [BMM19], and the randomized generation from that work can immediately be put to work in the current context.

```
Inductive has_type : tm → typ → Prop :=
  | T_Tru : has_type ttrue TBool
  | T_Fls : has_type tfalse TBool
  | T_Test : ∀ t1 t2 t3 T, has_type t1 TBool →
has_type t2 T →
          has_type t3 T → has_type (tif t1 t2 t3) T
  | T_Zro : has_type tzero TNat
  | T_Scc : ∀ t1, has_type t1 TNat → has_type (tsucc t1) TNat
  | T_Prd : ∀ t1, has_type t1 TNat → has_type (tpred t1 ) TNat
  | T_Iszro : ∀ t1, has_type t1 TNat →
has_type (tiszero t1) TBool.

Inductive nvalue : tm → Prop :=
 | nv_zero : nvalue tzero
 | nv_succ : ∀ t, nvalue t → nvalue (tsucc t).
Inductive bvalue : tm → Prop :=
 | bv_t : bvalue ttrue
 | bv_f : bvalue tfalse.
Reserved Notation "t1 '⟹ ' t2" (at level 40).
Inductive step : tm → tm → Prop :=
  | ST_IfTrue : ∀ t1 t2, (tif ttrue t1 t2) ⟹ t1
  | ST_IfFalse : ∀ t1 t2, (tif tfalse t1 t2) ⟹ t2
  | ST_If : ∀ t1 t1' t2 t3,
     t1 ⟹ t1' → (tif t1 t2 t3) ⟹ (tif t1' t2 t3)
  | ST_Succ : ∀ t1 t1',
     t1 ⟹ t1' → (tsucc t1) ⟹ (tsucc t1')
  | ST_PredZero : (tpred tzero) ⟹ tzero
  | ST_PredSucc : ∀ t1,
     nvalue t1 → (tpred (tsucc t1)) ⟹ t1
  | ST_Pred : ∀ t1 t1',
     t1 ⟹ t1' → (tpred t1) ⟹ (tpred t1')
  | ST_IszeroZero : (tiszero tzero) ⟹ ttrue
  | ST_IszeroSucc : ∀ t1,
     nvalue t1 → (tiszero (tsucc t1)) ⟹ tfalse
  | ST_Iszero : ∀ t1 t1',
     t1 ⟹ t1' → (tiszero t1) ⟹ (tiszero t1')
where "t1 '⟹ ' t2" := (step t1 t2).

Inductive notstuck (e : tm) (Step : tm → tm → Prop) : Prop :=
  | pn : nvalue e → notstuck e Step
  | pb : bvalue e → notstuck e Step
  | ps e' : Step e e' → notstuck e Step.
```

Figure 4.6: Semantics for a simple language with arithmetic expressions. With an error!

Conclusion

We have presented some advancements to Foundational Proof Certificates and the proof theory behind them along several directions. We briefly them up in the light of the global project, and sketch some future research lines.

Proof certificates and automation In chapter 2 we have shown how to interpret proof-theoretically some logical transformation that are commonly used as pre-processing steps in automated provers. The presented techniques should be developed further in experiments with real-world automated provers. In particular, the Alethe proof format [Sch+21], that has recently been proposed as a formalism to be employed by veriT and possibly other SMT solver, should be provided with an FPC presentation.

Linearized arithmetic In chapter 3 we have presented some results on the proof theory of logics with fixed points relative to focusing and polarization. We have used them as the foundation for some of the implementations in chapter 4. Much remains to be understood about the logical content and properties of these theories: a focusing theorem for classical logic with fixed points needs to be established, and the relation between intuitionistic, linear and classical systems needs to be studied. Based on this, the semantics of proof certificates could be more clearly extended to systems with induction or fixed points.

Interactive proof developments We have presented two simple extensions based on proof-theoretic foundations that increase the automation of proof assistants, either by automatically reconstructing arguments based on external proof evidence or by providing counterexamples. The integration of counterexamples into mathematical practice constitutes a central point in the work of Lakatos [Lak77]. In his critique of the formalist approach to mathematics, Lakatos argues against the view that mathematical discovery happens in a strictly deductive way, where precise definitions are laid out at first, and then theorems and their proofs are proposed and carried out as purely deductive activities. To this view Lakatos opposes one where mathematical discovery happens by a repeated process of proposing informal proofs and discovering counterexamples and refutations to the claimed proofs. In a twist of events, these views of Lakatos have been of inspiration for defenses of the practice of formalized mathematics such as [AGN09]. The argument here is that what the user carries out during the development of a formal proof does indeed happen in an informal setting: the actual formalization step is brought in when one appeals to the proof assistant's kernel to check the validity of the input.

The component that turns the user script (which might contain holes and ambiguous notation) into a full term to be checked is the elaborator, that we have discussed in chapter 4.

Two directions stem from this observation. One direction concerns increasing the expressivity of Foundational Proof Certificates as a device for easier elaboration of proof scripts into full proofs, by investigating more the features of interactive theorem prover. As an example, it is natural to encode typeclass resolution mechanisms in logic programming (see for example [BKL19]), and the style of this implementation resembles that of FPC; what would be interesting to explore is whether these devices require an extension to the FPC paradigm or whether they can be accounted for already in the current presentation.

The other direction is the analysis of counterexamples in this context. Indeed, it seems that the authors of [AGN09] contented themselves with considering the duality of partial versus complete proof scripts as a presentation of a paradigm in the style of Lakatos. However we have shown how a semantics of proofs can be useful for considering (formal) counterexamples. Lakatos distinguishes *global counterexample* to the primitive conjecture, and *local counterexamples* to a particular lemma. The PBT tactic we developed in chapter 4 can be useful only for an analysis of local counterexamples: it can be used only on purely positive goals, and not on complex conjectures; the resulting trace of a synchronous phase is a single data structure (a term invalidating the property, or an index in a case analysis) and it is of no help in localizing where a smaller offending lemma might be. Therefore, the intended usage is still contrary to the method of Lakatos: the user needs to elaborate a precise proof strategy for their theorem by pinpointing a series of small lemmas, and only using the empirical, PBT-supported method on such smaller lemmas. However, we see a path beyond this once we consider counterexamples to more complex claims, that is, once we go beyond purely positive formulas. In this setting, counterproofs generated by an automated helper contain more information that is not available at first sight: as a minimal example, consider an universal quantifier around a purely positive formula, for which a refutation certificate contains only a method refuting each user provided witness. The user should embark in an interaction with the certificate, for example by testing several witnesses and seeing them all refuted, in order to learn more about their conjecture. A proposal for this interaction has been done in [Mil21], and it should be possible to integrate this in a future version of our plugins for Coq.

Abstract proof representations We have referred in chapter 1 to the work on transcendental syntax, a research project proposed by J.-Y. Girard and brought forward by Eng [Eng]. While it is not clear whether the analysis of Foundational Proof Certificates is in total agreement with Transcendental Syntax, it could be interesting to study the properties of certificates as objects of their own right, without interpreting them as fully detailed proofs in a sequent calculus. In particular, the paradigm of interaction with counterexamples that we sketched in the previous paragraph should happen in this uninterpreted setting, since it involves partial and possibly incorrect derivations. On the other hand, an easier task should be that of describing the Stellar Resolution of Eng ans Seiller [ES22], which embodies the ideas of Transcendental Syntax, as a

format of proof certificates.

Positive polarities in type theories In chapter 4 we have presented an adaptation of a focused proof system for depentent types as the foundation for the elaboration of proof certificates in proof assistants based on such type theories. However, in order to account for positive polarities we restricted ourselves to first order proofs and we decoupled the term representation from the proof system. In order to obtain a clear treatment of proof certificates in proof assistants, a focused proof system with both positive and negative polarities is needed.

We have described in chapter 1 and 2 how positive polarities correspond to locally named subproofs. When proofs are represented by means of λ-terms, this corresponds to using `let` binders, which is a feature present in many proof assistants. The system $\lambda\kappa$ from [BNGG15], whose negative fragment we have employed in our work, provides a decomposition of `let` binders by means of the binder κ. When building a continuation (in the negative fragment, an applicative list terminated by ϵ, corresponding to the initial rule), if the negative phase ends by a release instead of init the κ binder is inserted, which names the currently built subproof and handles it to the store rule, to be indexed and stored. The proof term built in such a phase has the form $k :: l :: \kappa x^A.(t)$, which corresponds in Coq's syntax to `let x : A := k l in t`. The same authors of $\lambda\kappa$ noted in [GG15] that their system would have been beneficial for proof assistants, but didn't reach a complete presentation of the system.

We leave a proposal for an extension of Pure Type Systems that accounts for positive types in fig. 4.7: the main idea we add is that of adding, on top of the set $\mathscr{S}$ of sorts, a covering $\{\mathscr{N}, \mathscr{P}\}$. The intended meaning is that inhabitants of a sort belonging to $\mathscr{N}$ are negative, and the inhabitants of a sort belonging to $\mathscr{P}$ are positive. Therefore, we obtain for example the types $Prop^-$ of negative propositions, or $Type^+$ of positive types, or Set^+ of positive data types. The metatheory of this system needs to be studied, and further applications can be explored. A prototype implementation of this system is available at `https://github.com/manmatteo/playground/tree/master/fpc-cic`

Sorting rules

$$\frac{}{\emptyset\ wf} \qquad \frac{\Gamma \Uparrow \cdot \vdash A : s \Uparrow \cdot \quad (x \notin Dom(\Gamma))}{\Gamma, x : A\ wf} \qquad \frac{\Gamma\ wf \quad (s, s') \in \mathscr{A}}{\Gamma \Downarrow \cdot \vdash s : s' \Downarrow \cdot} \qquad \frac{\Gamma\ wf \quad s \in \mathscr{S}}{\Gamma \Downarrow s \vdash \epsilon : \cdot \Downarrow s}$$

$$\frac{\Gamma \Downarrow \cdot \vdash A : s_1 \Downarrow \cdot \quad \Gamma, x : A \Downarrow \cdot \vdash B : s_2 \Downarrow \cdot \quad \Gamma \Downarrow s_3 \vdash l : s_3 \Downarrow \cdot \quad (s_1, s_2, s_3) \in \mathscr{R}}{\Gamma \Uparrow \cdot \vdash (x\colon A)B\ l : s_3 \Uparrow \cdot}$$

Initial

$$\frac{\Gamma \Uparrow \cdot \vdash N : s^- \Uparrow \cdot}{\Gamma \Downarrow N \vdash \epsilon : \cdot \Downarrow N}\ axiom_n \qquad \frac{\Gamma \Uparrow \cdot \vdash B : s^+ \Uparrow \cdot}{\Gamma, x : B \Downarrow \cdot \vdash x : B \Downarrow \cdot}\ axiom_p$$

Product

$$\frac{\Gamma \Uparrow \cdot \vdash (x\colon A)B : s \Uparrow \cdot \quad \Gamma \Uparrow A \vdash M : B \Uparrow \cdot}{\Gamma \Uparrow \cdot \vdash \lambda x^A.M : (x\colon A)B \Uparrow \cdot}\ P_r$$

$$\frac{\Gamma \Uparrow \cdot \vdash (x\colon A)B : s \Uparrow \cdot \quad \Gamma \Downarrow \cdot \vdash p : A \Downarrow \cdot \quad \Gamma \Downarrow [x/p]B \vdash l : \cdot \Downarrow C}{\Gamma \Downarrow (x\colon A)B \vdash p :: l : \cdot \Downarrow C}\ P_l$$

Structural rules

$$\frac{\Gamma \Uparrow \cdot \vdash A : s^- \Uparrow \cdot \quad \Gamma, x : A \Downarrow A \vdash l : \cdot \Downarrow B}{\Gamma, x : A \Uparrow \cdot \vdash x\ l : \cdot \Uparrow B}\ decide_l \qquad \frac{\Gamma, x : N \Uparrow \Theta \vdash t : A \Uparrow \cdot}{\Gamma \Uparrow N, \Theta \vdash t : A \Uparrow \cdot}\ store_l$$

$$\frac{\Gamma \Uparrow \cdot \vdash P : s^+ \Uparrow \cdot \quad \Gamma \Downarrow \cdot \vdash p : P \Downarrow \cdot}{\Gamma \Uparrow \cdot \vdash \lceil p \rceil : \cdot \Uparrow P}\ decide_r \qquad \frac{\Gamma \Uparrow \cdot \vdash t : \cdot \Uparrow A}{\Gamma \Uparrow \cdot \vdash t : A \Uparrow \cdot}\ store_r$$

$$\frac{\Gamma \Uparrow \cdot \vdash P : s^+ \Uparrow \cdot \qquad \Gamma \Uparrow P \vdash M : \cdot \Uparrow A}{\Gamma \Downarrow P \vdash \kappa x^P.(M) : \cdot \Downarrow A}\ release_l$$

$$\frac{\Gamma \Uparrow \cdot \vdash N : s^- \Uparrow \cdot \qquad \Gamma \Uparrow \cdot \vdash t : N \Uparrow \cdot}{\Gamma \Downarrow \cdot \vdash \lfloor t \rfloor : N \Downarrow \cdot}\ release_r$$

Figure 4.7: A calculus for polarized pure type systems, where $s^- \in \mathscr{N}$ and $s^+ \in \mathscr{P}$

Bibliography

[AB19] Juan P. Aguilera and Matthias Baaz. "Unsound Inferences Make Proofs Shorter". In: *The Journal of Symbolic Logic* 84.1 (Mar. 2019), pp. 102–122. ISSN: 0022-4812, 1943-5886. DOI: 10.1017/jsl.2018.51 (cit. on p. 37).

[AGN09] Andrea Asperti, Herman Geuvers, and Raja Natarajan. "Social Processes, Program Verification and All That". In: *Mathematical Structures in Computer Science* 19.5 (Oct. 2009), pp. 877–896. ISSN: 0960-1295, 1469-8072. DOI: 10.1017/S0960129509990041 (cit. on pp. 95, 96).

[Alv+06] Sandra Alves et al. "The Power of Linear Functions". In: *Computer Science Logic*. Ed. by Zoltán Ésik. Red. by David Hutchison et al. Vol. 4207. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 119–134. ISBN: 978-3-540-45458-8 978-3-540-45459-5. DOI: 10.1007/11874683_8 (cit. on p. 65).

[Alv+10] Sandra Alves et al. "Gödel's System T Revisited". In: *Theoretical Computer Science* 411.11-13 (Mar. 2010), pp. 1484–1500. ISSN: 03043975. DOI: 10/c8p35w (cit. on p. 64).

[And92] Jean-Marc Andreoli. "Logic Programming with Focusing Proofs in Linear Logic". In: *J. of Logic and Computation* 2.3 (1992), pp. 297–347. DOI: 10.1093/logcom/2.3.297 (cit. on p. 18).

[Bae08] David Baelde. "A Linear Approach to the Proof Theory of Least and Greatest Fixed Points". École Polytechnique, 2008. URL: http://www.lix.polytechnique.fr/Labo/Dale.Miller/baelde08phd.pdf (visited on 06/20/2017) (cit. on p. 64).

[Bae12a] David Baelde. "Least and greatest fixed points in linear logic". In: *ACM Trans. on Computational Logic* 13.1 (Apr. 2012), 2:1–2:44. DOI: 10.1145/2071368.2071370 (cit. on pp. 65, 66).

[Bae12b] David Baelde. "Least and Greatest Fixed Points in Linear Logic". In: *ACM Transactions on Computational Logic* 13.1 (2012), p. 48 (cit. on p. 67).

[BHW12] Matthias Baaz, Stefan Hetzl, and Daniel Weller. "On the Complexity of Proof Deskolemization". In: *The Journal of Symbolic Logic* 77.2 (2012), pp. 669–686. ISSN: 0022-4812. JSTOR: 41713912 (cit. on pp. 38, 40).

[BKL19] Henning Basold, Ekaterina Komendantskaya, and Yue Li. "Coinduction in Uniform: Foundations for Corecursive Proof Search with Horn Clauses". In: *Programming Languages and Systems*. Ed. by Luís Caires. Vol. 11423. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 783–813. ISBN: 978-3-030-17183-4 978-3-030-17184-1. DOI: 10.1007/978-3-030-17184-1_28 (cit. on p. 96).

[BL22] Matthias Baaz and Anela Lolic. "Andrews Skolemization May Shorten Resolution Proofs Non-elementarily". In: *Logical Foundations of Computer Science*. Ed. by Sergei Artemov and Anil Nerode. Cham: Springer International Publishing, 2022, pp. 9–24. ISBN: 978-3-030-93100-1. DOI: 10.1007/978-3-030-93100-1_2 (cit. on p. 37).

[Bla17] Roberto Blanco. "Applications for Foundational Proof Certificates in theorem proving". PhD thesis. Université Paris-Saclay, Dec. 2017. URL: https://tel.archives-ouvertes.fr/tel-01743857 (cit. on pp. 6, 9, 23).

[BM07a] David Baelde and Dale Miller. "Least and Greatest Fixed Points in Linear Logic". In: *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 2007, pp. 92–106. URL: http://link.springer.com/chapter/10.1007/978-3-540-75560-9_9 (visited on 06/06/2017) (cit. on pp. 62, 66).

[BM07b] David Baelde and Dale Miller. "Least and greatest fixed points in linear logic". In: *International Conference on Logic for Programming and Automated Reasoning (LPAR)*. Ed. by N. Dershowitz and A. Voronkov. Vol. 4790. Lecture Notes in Computer Science. 2007, pp. 92–106. DOI: 10.1007/978-3-540-75560-9_9 (cit. on p. 57).

[BMM19] Roberto Blanco, Dale Miller, and Alberto Momigliano. "Property-Based Testing via Proof Reconstruction". In: *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages 2019*. PPDP '19: Principles and Practice of Programming Languages 2019. Porto Portugal: ACM, Oct. 7, 2019, pp. 1–13. ISBN: 978-1-4503-7249-7. DOI: 10.1145/3354166.3354170 (cit. on pp. 74, 89, 93).

[BMM20a] Roberto Blanco, Matteo Manighetti, and Dale Miller. "FPC-Coq: Using ELPI to Elaborate External Proof Evidence into Coq Proofs". The Coq Workshop 2020 (Aubervilliers (Online)). July 5, 2020. URL: https://coq-workshop.gitlab.io/2020/#talk-1-1-1 (cit. on pp. 6, 10).

[BMM20b] Roberto Blanco, Matteo Manighetti, and Dale Miller. "Linking a λProlog Proof Checker to the Coq Kernel". Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2020) (Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2020), Paris (online)). June 30, 2020. URL: https://lfmtp.org/workshops/2020/program.shtml (cit. on pp. 6, 10).

[BMS10] David Baelde, Dale Miller, and Zachary Snow. "Focused Inductive Theorem Proving". In: *Automated Reasoning*. Ed. by Jürgen Giesl and Reiner Hähnle. Red. by David Hutchison et al. Vol. 6173. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 278–292. ISBN: 978-3-642-14202-4 978-3-642-14203-1. DOI: 10.1007/978-3-642-14203-1_24 (cit. on pp. 57, 68).

[BNGG15] Taus Brock-Nannestad, Nicolas Guenot, and Daniel Gustafsson. "Computation in Focused Intuitionistic Logic". In: *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming*. PPDP '15. New York, NY, USA: ACM, 2015, pp. 43–54. ISBN: 978-1-4503-3516-4. DOI: 10.1145/2790449.2790528 (cit. on pp. 80, 97).

[Boh09] Sascha Bohme. "Proof Reconstruction for Z3 in Isabelle/HOL". In: *7th International Workshop on Satisfiability Modulo Theories (SMT'09)* (2009), p. 11 (cit. on p. 55).

[Bou] *Bound Variables, in Mathematical Markup Language (MathML) Version 4.0*. World Wide Web Consortium. URL: https://w3c.github.io/mathml/#contm_bvar_sec (visited on 10/18/2022) (cit. on p. 55).

[Boy94] Robert S. Boyer. "A Mechanically Proof-Checked Encyclopedia of Mathematics: Should We Build One? Can We?" In: *Automated Deduction — CADE-12*. Ed. by Alan Bundy. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1994, pp. 237–251. ISBN: 978-3-540-48467-7. DOI: 10.1007/3-540-58156-1_17 (cit. on pp. 4, 8).

[CCM20] Matteo Cavada, Andrea Colò, and Alberto Momigliano. "MutantChick: Type-Preserving Mutation Analysis for Coq". In: *CILC*. Vol. 2710. CEUR Workshop Proceedings. CEUR-WS.org, 2020, pp. 105–112. URL: http://ceur-ws.org/Vol-2710/short2.pdf (cit. on p. 92).

[CH00] Pierre-Louis Curien and Hugo Herbelin. "The Duality of Computation". In: *ACM Sigplan Notices*. Vol. 35. ACM, 2000, pp. 233–243 (cit. on p. 19).

[Chi15] Zakaria Chihani. "Certification of First-order Proofs in Classical and Intuitionistic Logics". École Polytechnique, 2015 (cit. on pp. 5, 9, 26).

[Chu40] Alonzo Church. "A Formulation of the Simple Theory of Types". In: *Journal of Symbolic Logic* 5.2 (June 1940), pp. 56–68. ISSN: 0022-4812, 1943-5886. DOI: 10.2307/2266170 (cit. on pp. 11, 68).

[CIM16] Zakaria Chihani, Danko Ilik, and Dale Miller. *Classical Polarizations Yield Double-Negation Translations*. Aug. 18, 2016. URL: https://hal.inria.fr/hal-01354298/document (visited on 05/02/2017) (cit. on pp. 26, 27).

[Cla78] K. L. Clark. "Negation as failure". In: *Logic and Data Bases*. Ed. by J. Gallaire and J. Minker. Plenum Press, New York, 1978, pp. 293–322 (cit. on p. 72).

[CM17] James Cheney and Alberto Momigliano. "αCheck: A mechanized metatheory model checker". In: *Theory and Practice of Logic Programming* 17.3 (2017), pp. 311–352 (cit. on p. 89).

[CMM19] Kaustuv Chaudhuri, Matteo Manighetti, and Dale Miller. "A Proof-Theoretic Approach to Certifying Skolemization". In: *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs - CPP 2019*. The 8th ACM SIGPLAN International Conference. Cascais, Portugal: ACM Press, 2019, pp. 78–90. ISBN: 978-1-4503-6222-1. DOI: 10.1145/3293880.3294094 (cit. on pp. 6, 10).

[CMR17] Zakaria Chihani, Dale Miller, and Fabien Renaud. "A Semantic Framework for Proof Evidence". In: *Journal of Automated Reasoning* 59.3 (Oct. 1, 2017), pp. 287–330. ISSN: 0168-7433, 1573-0670. DOI: 10.1007/s10817-016-9380-6 (cit. on pp. 11, 23, 24, 48).

[DL05] Ugo Dal Lago. "The Geometry of Linear Higher-Order Recursion". In: *20th Annual IEEE Symposium on Logic in Computer Science (LICS' 05)*. 20th Annual IEEE Symposium on Logic in Computer Science (LICS' 05). June 2005, pp. 366–375. DOI: 10.1109/LICS.2005.52 (cit. on p. 64).

[DN15] Roy Dyckhoff and Sara Negri. "Geometrisation of First-Order Logic". In: *Bulletin of Symbolic Logic* 21.2 (June 2015), pp. 123–163. ISSN: 1079-8986, 1943-5894. DOI: 10.1017/bsl.2015.7 (cit. on p. 55).

[DP60] Martin Davis and Hilary Putnam. "A Computing Procedure for Quantification Theory". In: *Journal of the ACM* 7.3 (July 1, 1960), pp. 201–215. ISSN: 0004-5411. DOI: 10.1145/321033.321034 (cit. on pp. 35, 54).

[Dum93] Michael Dummett. *The Logical Basis of Metaphysics*. Third Printing edition. Harvard University Press, Jan. 1, 1993. 366 pp. ISBN: 978-0-674-53786-6 (cit. on p. 29).

[Dun+15] Cvetan Dunchev et al. "ELPI: Fast, Embeddable, λProlog Interpreter". In: Proceedings of LPAR. Nov. 24, 2015. URL: https://hal.inria.fr/hal-01176856/document (visited on 03/23/2019) (cit. on pp. 15, 84).

[Ebn+16] Gabriel Ebner et al. "System Description: GAPT 2.0". In: *Automated Reasoning*. Ed. by Nicola Olivetti and Ashish Tiwari. Vol. 9706. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016, pp. 293–301. ISBN: 978-3-319-40228-4 978-3-319-40229-1. DOI: 10.1007/978-3-319-40229-1_20 (cit. on p. 40).

[EH21] Yacine El Haddad. "Integrating Automated Theorem Provers in Proof Assistants". Université Paris-Saclay, 2021 (cit. on p. 40).

[Eng] Boris Eng. "A Gentle Introduction to Girard's Transcendental Syntax for the Linear Logician". In: (), p. 47 (cit. on pp. 28, 29, 96).

[ES22] Boris Eng and Thomas Seiller. *Multiplicative Linear Logic from a Resolution-Based Tile System*. July 18, 2022. URL: http://arxiv.org/abs/2207.08465 (visited on 11/09/2022). arXiv: 2207.08465 [cs, math] (cit. on p. 96).

[FB97] George Fink and Matt Bishop. "Property-based testing: a new approach to testing for assurance". In: *ACM SIGSOFT Software Engineering Notes* (July 1997), pp. 74–80. URL: http://doi.acm.org/10.1145/263244.263267 (cit. on p. 74).

[Fin85] Kit Fine. "Natural Deduction and Arbitrary Objects". In: *Journal of Philosophical Logic* 14.1 (Feb. 1985), pp. 57–107. ISSN: 0022-3611, 1573-0433. DOI: 10.1007/BF00542649 (cit. on p. 32).

[FK16] Michael Färber and Cezary Kaliszyk. "No Choice: Reconstruction of First-order ATP Proofs without Skolem Functions." In: *PAAR Workshop at IJCAR*. 2016, pp. 24–31. URL: http://cs.ru.nl/paar16/PAAR2016.pdf#page=28 (visited on 10/11/2017) (cit. on p. 40).

[FO12] Gilda Ferreira and Paulo Oliva. "On the Relation Between Various Negative Translations". In: *Logic, Construction, Computation*. Ed. by Ulrich Berger et al. DE GRUYTER, Dec. 31, 2012, pp. 227–258. ISBN: 978-3-11-032453-2. DOI: 10.1515/9783110324921.227 (cit. on p. 27).

[Fre80] Gottlob Frege. *The Foundations of Arithmetic*. 2 Revised. Northwestern University Press, 1980. ISBN: 978-0-8101-0605-5 0-8101-0605-1 (cit. on pp. 3, 7).

[Gen35] Gerhard Gentzen. "Investigations into Logical Deduction". In: *The Collected Papers of Gerhard Gentzen*. Ed. by M. E. Szabo. Amsterdam: North-Holland, 1935, pp. 68–131. DOI: 10.1007/BF01201353 (cit. on pp. 12, 29).

[GG15] Nicolas Guenot and Daniel Gustafsson. "Sequent Calculus and Equational Programming". In: *Electronic Proceedings in Theoretical Computer Science* 185 (July 27, 2015), pp. 102–109. ISSN: 2075-2180. DOI: 10.4204/EPTCS.185.7 (cit. on p. 97).

[Gir01] Jean-Yves Girard. "Locus Solum: From the Rules of Logic to the Logic of Rules". In: *Mathematical Structures in Computer Science* 11.3 (June 2001), pp. 301–506. ISSN: 1469-8072, 0960-1295. DOI: 10/fq8tpk (cit. on p. 28).

[Gir17] Jean-Yves Girard. "Transcendental Syntax I: Deterministic Case". In: *Mathematical Structures in Computer Science* 27.5 (June 2017), pp. 827–849. ISSN: 0960-1295, 1469-8072. DOI: 10/gdcfp8 (cit. on p. 28).

[Gir87a] Jean-Yves Girard. "Linear Logic". In: *Theoretical Computer Science* 50.1 (1987), pp. 1–102. DOI: 10.1016/0304-3975(87)90045-4 (cit. on p. 17).

[Gir87b] Jean-Yves Girard. "Linear Logic". In: *Theoretical Computer Science* 50.1 (Jan. 1, 1987), pp. 1–101. ISSN: 0304-3975. DOI: 10/cmv5mj (cit. on p. 61).

[Gir91] Jean-Yves Girard. "A New Constructive Logic: Classic Logic". In: *Mathematical Structures in Computer Science* 1.3 (Nov. 1991), pp. 255–296. ISSN: 1469-8072, 0960-1295. DOI: 10.1017/S0960129500001328 (cit. on pp. 28, 66).

[Gir92] Jean-Yves Girard. "A Fixpoint Theorem in Linear Logic". Feb. 1992 (cit. on p. 61).

[Har09] John Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, Mar. 12, 2009. 703 pp. ISBN: 978-0-521-89957-4. Google Books: lYSYLPWJQKMC (cit. on pp. 43, 44).

[Hen10] Anders Starcke Henriksen. *Using LJF as a Framework for Proof Systems*. report. 2010. URL: https://hal.inria.fr/inria-00442159 (visited on 10/16/2022) (cit. on p. 27).

[Her95] Hugo Herbelin. "A λ-Calculus Structure Isomorphic to Gentzen-style Sequent Calculus Structure". In: *Computer Science Logic*. Ed. by Leszek Pacholski and Jerzy Tiuryn. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1995, pp. 61–75. ISBN: 978-3-540-49404-1. DOI: 10.1007/BFb0022247 (cit. on p. 18).

[HM18] Quentin Heath and Dale Miller. "A Proof Theory for Model Checking". In: (2018), p. 31 (cit. on pp. 73, 74).

[How80] William A. Howard. "The Formulae-as-Type Notion of Construction, 1969". In: *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*. Ed. by J. P. Seldin and R. Hindley. New York: Academic Press, 1980, pp. 479–490 (cit. on p. 68).

[HSH91] Lars Hallnäs and Peter Schroeder-Heister. "A Proof-Theoretic Approach to Logic Programming: II. Programs as Definitions". In: *Journal of Logic and Computation* 1.5 (1991), pp. 635–660. ISSN: 0955-792X, 1465-363X. DOI: 10.1093/logcom/1.5.635 (cit. on p. 73).

[HUW16] John Harrison, Josef Urban, and Freek Wiedijk. "Preface: Twenty Years of the QED Manifesto". In: *Journal of Formalized Reasoning* 9.1 (1 Jan. 29, 2016), pp. 1–2. ISSN: 1972-5787. DOI: 10.6092/issn.1972-5787/5974 (cit. on pp. 5, 9).

[HW18] Stefan Hetzl and Tin Lok Wong. "Some Observations on the Logical Foundations of Inductive Theorem Proving". In: *Logical Methods in Computer Science ; Volume 13* (2018), Issue 4 ; 1860–5974. DOI: 10/gfwfhj (cit. on p. 57).

[Kle+12] Casey Klein et al. "Run Your Research: On the Effectiveness of Lightweight Mechanization". In: *ACM SIGPLAN Notices* 47.1 (Jan. 18, 2012), pp. 285–296. ISSN: 0362-1340, 1558-1160. DOI: 10.1145/2103621.2103691 (cit. on p. 92).

[KPP21] Denis Kuperberg, Laureline Pinault, and Damien Pous. "Cyclic Proofs, System t, and the Power of Contraction". In: *Proceedings of the ACM on Programming Languages* 5 (POPL Jan. 4, 2021), pp. 1–28. ISSN: 2475-1421. DOI: 10.1145/3434282 (cit. on pp. 64, 68).

[Lak77] Imre Lakatos. *Proofs and Refutations: The Logic of Mathematical Discovery*. Cambridge University Press, 1977. ISBN: 978-0-521-29038-8. URL: http://gen.lib.rus.ec/book/index.php?md5=2bab7900d47c08dd874bb93c0641d2d6 (visited on 11/08/2022) (cit. on p. 95).

[Lam+17] Leonidas Lampropoulos et al. "Beginner's Luck: A Language for Property-Based Generators". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL '17: The 44th Annual ACM SIGPLAN Symposium on Principles of Programming Languages. Paris France: ACM, Jan. 2017, pp. 114–129. ISBN: 978-1-4503-4660-3. DOI: 10.1145/3009837.3009868 (cit. on p. 93).

[LDM11] Stéphane Lengrand, Roy Dyckhoff, and James McKinna. "A Focused Sequent Calculus Framework for Proof Search in Pure Type Systems". In: *Logical Methods in Computer Science* 7.1 (Mar. 23, 2011). Ed. by Henk Barendregt, p. 6. ISSN: 18605974. DOI: 10.2168/LMCS-7(1:6)2011 (cit. on p. 80).

[LM07] Chuck Liang and Dale Miller. "Focusing and Polarization in Intuitionistic Logic". In: *International Workshop on Computer Science Logic*. Springer, 2007, pp. 451–465. URL: http://link.springer.com/10.1007%2F978-3-540-74915-8_34 (visited on 05/03/2017) (cit. on p. 26).

[LM09] Chuck Liang and Dale Miller. "Focusing and Polarization in Linear, Intuitionistic, and Classical Logics". In: *Theoretical Computer Science* 410.46 (Nov. 2009), pp. 4747–4768. ISSN: 03043975. DOI: 10.1016/j.tcs.2009.07.041 (cit. on pp. 19, 80).

[LPP18] Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. "Generating good generators for inductive relations". In: *Proc. ACM Program. Lang.* 2.POPL (2018), 45:1–45:30 (cit. on p. 89).

[MB08] Leonardo de Moura and Nikolaj Bjørner. "Proofs and Refutations, and Z3". In: *The 7th International Workshop on the Implementation of Logics*. 2008, p. 10. URL: http://ceur-ws.org/Vol-418/ (cit. on p. 55).

[Mil21] Dale Miller. "How to Explain a Counterexample". Philmath Seminar (IHPST, Paris). Dec. 14, 2021 (cit. on p. 96).

[Mil87] Dale A. Miller. "A Compact Representation of Proofs". In: *Studia Logica* 46.4 (Dec. 1987), pp. 347–370. ISSN: 0039-3215, 1572-8730. DOI: 10.1007/BF00370646 (cit. on pp. 39, 40).

[Mil+91] Dale Miller et al. "Uniform Proofs as a Foundation for Logic Programming". In: *Annals of Pure and Applied Logic* 51.1-2 (Mar. 1991), pp. 125–157. ISSN: 01680072. DOI: 10.1016/0168-0072(91)90068-W (cit. on p. 18).

[ML96] Per Martin-Löf. "On the Meanings of the Logical Constants and the Justifications of the Logical Laws". In: *Nordic journal of philosophical logic* 1.1 (1996), pp. 11–60 (cit. on p. 29).

[MM22a] Matteo Manighetti and Dale Miller. *Computational Logic Based on Linear Logic and Fixed Points*. Feb. 18, 2022. URL: https://hal.inria.fr/hal-03579451 (visited on 10/16/2022) (cit. on pp. 6, 10).

[MM22b] Matteo Manighetti and Dale Miller. "Peano Arithmetic and muMALL: An Extended Abstract". In: 8826 (Sept. 11, 2022). ISSN: 2516-2314. URL: https://ww.easychair.org/publications/preprint/HJTJ (visited on 10/16/2022) (cit. on pp. 6, 10).

[MMM20] Matteo Manighetti, Dale Miller, and Alberto Momigliano. "Two Applications of Logic Programming to Coq". In: *Submitted to TYPES 2020 Post-Proceedings* (Nov. 2020), p. 19 (cit. on pp. 6, 10).

[MMP03] Raymond McDowell, Dale Miller, and Catuscia Palamidessi. "Encoding transition systems in sequent calculus". In: *Theoretical Computer Science* 294.3 (2003), pp. 411–437 (cit. on p. 73).

[MN12] Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, 2012 (cit. on pp. 15, 33).

[MP99] Dale Miller and Catuscia Palmidessi. "Foundational Aspects of Syntax". In: *ACM Computing Surveys* 31 (3es Sept. 1999), p. 11. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/333580.333590 (cit. on p. 11).

[NM10] Vivek Nigam and Dale Miller. "A Framework for Proof Systems". In: *Journal of Automated Reasoning* 45.2 (Aug. 1, 2010), pp. 157–188. ISSN: 0168-7433, 1573-0670. DOI: 10.1007/s10817-010-9182-1 (cit. on p. 27).

[Oka99] Mitsuhiro Okada. "Phase Semantic Cut-Elimination and Normalization Proofs of First- and Higher-Order Linear Logic". In: *Theoretical Computer Science* 227.1-2 (Sept. 1999), pp. 333–396. ISSN: 03043975. DOI: 10.1016/S0304-3975(99)00058-4 (cit. on p. 61).

[Par+15] Zoe Paraskevopoulou et al. "Foundational Property-Based Testing". In: *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*. Ed. by Christian Urban and Xingyuan Zhang. Vol. 9236. Lecture Notes in Computer Science. Springer, 2015, pp. 325–343. DOI: 10.1007/978-3-319-22102-1_22 (cit. on p. 89).

[Pie22] Benjamin C. Pierce. *Types: Type Systems*. Software Foundations Volume 2: Programming Languages Foundations. 2022. URL: https://softwarefoundations.cis.upenn.edu/plf-current/Types.html (visited on 05/08/2022) (cit. on p. 92).

[Pra06] Dag Prawitz. "Meaning Approached Via Proofs". In: *Synthese* 148.3 (Feb. 1, 2006), pp. 507–524. ISSN: 0039-7857, 1573-0964. DOI: 10.1007/s11229-004-6295-2 (cit. on p. 29).

[PSU21] Wolfram Pfeifer, Jonas Schiffl, and Mattias Ulbrich. "Reconstructing Z3 Proofs in KeY: There and Back Again". In: *Proceedings of the 23rd ACM International Workshop on Formal Techniques for Java-like Programs*. ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis. Virtual Denmark: ACM, July 13, 2021, pp. 24–31. ISBN: 978-1-4503-8543-5. DOI: 10.1145/3464971.3468421 (cit. on p. 55).

[Put80] Hilary Putnam. "Models and Reality". In: *The Journal of Symbolic Logic* 45.3 (1980), pp. 464–482. ISSN: 0022-4812. JSTOR: 2273415. DOI: 10.2307/2273415 (cit. on p. 54).

[Ral20] Benjamin Ralph. "Herbrand Proofs and Expansion Proofs as Decomposed Proofs". In: *Journal of Logic and Computation* 30.8 (Dec. 10, 2020), pp. 1711–1742. ISSN: 0955-792X. DOI: 10.1093/logcom/exaa052 (cit. on p. 37).

[RNL08] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. "Smallcheck and Lazy Smallcheck: automatic exhaustive testing for small values". In: *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008*. Ed. by Andy Gill. ACM, 2008, pp. 37–48. DOI: 10.1145/1411286.1411292 (cit. on p. 89).

[RV01] John Alan Robinson and Andrei Voronkov. *Handbook of Automated Reasoning*. Vol. 1. Elsevier, 2001. ISBN: 0-444-50813-9 (cit. on pp. 24, 34).

[Sch+21] Hans-Jörg Schurr et al. "Alethe: Towards a Generic SMT Proof Format (Extended Abstract)". In: *Proceedings PxTP 2021*. Electronic Proceedings in Theoretical Computer Science 336 (July 7, 2021), pp. 49–54. ISSN: 2075-2180. DOI: 10.4204/EPTCS.336.6 (cit. on p. 95).

[Sem] *Semantics of the PROV Data Model*. URL: https://www.w3.org/TR/2013/NOTE-prov-sem-20130430/ (visited on 10/18/2022) (cit. on p. 55).

[SH93] Peter Schroeder-Heister. "Rules of Definitional Reflection". In: [*1993*] *Proceedings Eighth Annual IEEE Symposium on Logic in Computer Science*. [1993] Proceedings Eighth Annual IEEE Symposium on Logic in Computer Science. June 1993, pp. 222–232. DOI: 10/bpz3rc (cit. on p. 61).

[SU06] Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard Isomorphism*. Vol. 149. Elsevier, 2006. URL: https://doi.org/10.1016/s0049-237x(06)x8001-1 (cit. on p. 77).

[SZS] Geoff Sutcliffe, Jurgen Zimmer, and Stephan Schulz. "TSTP Data-Exchange Formats for Automated Theorem Proving Tools". In: (), p. 15 (cit. on p. 54).

[Tas18] Enrico Tassi. *Elpi: an extension language for Coq*. CoqPL 2018: The Fourth International Workshop on Coq for Programming Languages. Jan. 2018. URL: https://hal.inria.fr/hal-01637063/ (cit. on p. 84).

[Tä77] Sten-Åke Tärnlund. "Horn Clause Computability". In: *BIT* 17.2 (June 1977), pp. 215–226. ISSN: 0006-3835, 1572-9125. DOI: 10.1007/BF01932293 (cit. on p. 73).

[Zei] Noam Zeilberger. "The Logical Basis of Evaluation Order and Pattern-Matching". Carnegie Mellon University. URL: https://apps.dtic.mil/sti/pdfs/ADA507018.pdf (visited on 02/18/2022) (cit. on p. 19).

[Zei08] Noam Zeilberger. "On the Unity of Duality". In: *Annals of Pure and Applied Logic* 153.1 (Apr. 1, 2008), pp. 66–96. ISSN: 0168-0072. DOI: 10.1016/j.apal.2008.01.001 (cit. on p. 29).

[ZH17] Théo Zimmermann and Hugo Herbelin. "Coq's Prolog and Application to Defining Semi-Automatic Tactics". In: *Type Theory Based Tools*. Paris, France, Jan. 2017. URL: https://hal.archives-ouvertes.fr/hal-01671994 (visited on 07/20/2022) (cit. on p. 84).

ECOLE DOCTORALE

Titre : Developpements de théorie de la démonstration pour le partage de démonstrations

Mots clés : Logique, Démonstration automatisé de théorèmes, Théorie de la démonstration, Raisonnement automatisé, Certificats de preuve, Vérification de démonstrations

Résumé : La vérification automatisée de démonstrations mathématiques est une application de la logique computationnelle qui est de grande importance à la fois pour les mathématiques et pour l'informatique. Ces applications vont de la vérification de propriétés de systèmes logiciels, afin d'accroître la confiance que le logiciel fonctionnera selon les expectations, à la création de corpus de mathématiques formalisées, où l'activité des mathématiciens est vérifiée par la machine et rendue disponible à d'autres mathématiciens. Les démonstrations formelles peuvent, de leur part, être produites soit par des humains à l'aide d'un logiciel, soit par des logiciels de démonstration automatisée. Ces certificats pour la validité des énoncés viennent donc en beaucoup de formats différents et sont exprimés dans une variété de paradigmes : à une extrémité, il y a les langages les plus expressifs, destinés à être lisibles par des humains ; à l'autre extrémité, il y a les formats plus succincts qui proposent une évidence minimale pour la validité d'un énoncé obtenue par des logiciels très optimisés. Cette thèse concerne certaines avancées dans le projet des Certificats de Preuve Fondamentaux (FPC), un projet de recherche qui vise à ancrer la signification des langages concrets pour la représentation des preuves à la théorie structurelle des démonstrations. La théorie structurelle des démonstrations est l'étude mathématique des preuves mathématiques, initiée par Gentzen. On montrera comment les FPC sont suffisamment expressifs pour prendre en compte l'interprétation directe des deux passages d'optimisation utilisés par les démonstrateurs automatisés : la Skolemization et les transformations de Tseitin. Habituellement, un utilisateur qui souhaite inclure dans son travail des preuves obtenues à l'aide d'un démonstrateur automatisé utilisant ces techniques aurait dû inclure plus d'axiomes ou faire confiance en quelque façon à ces procédures ; on montrera une méthode pour interpréter ces démonstrations directement comme démonstrations de l'énoncé original, avant la transformation. Puis, on montrera des propriétés de la théorie de la démonstration des logiques avec point fixe qui avancent les fondations nécessaires pour la définition de la signification des démonstrations qui utilisent l'induction mathématique dans le contexte des FPC. Finalement, nous présenterons le développement de deux extensions de l'assistant de preuve Coq qui intègrent le traitement des certificats de preuve dans le processus interactive de démonstration avec cet assistant.

Title: Developing proof theory for proof exchange

Keywords: Logic, Theorem proving, Proof theory, Automated Reasoning, Proof Certificates, Proof Verification

Abstract: The mechanized verification of mathematical proofs is an application of computational logic that is of importance in both mathematics and computer science. These applications range from the verification of formal properties of software systems, to increase the trust that software will work as expected, to the creation of corpora of formalized mathematics, where the activity of mathematicians is checked by machine and made available to other mathematicians for integration into their work. Formal proofs, in turn, can be produced either by humans with the aid of software, or by automated theorem provers and similar pieces of software that produce some form of evidence that a statement should hold. These certificates for the validity of statements come therefore in various formats with wildly different paradigms: on one extreme, there are languages that are very expressive and aimed at being readable by humans, while at the other extreme there are very succinct formats that represent minimal evidence provided by highly optimized pieces of software. This thesis concerns some advancements in the project of Foundational Proof Certificates (FPC), a research program that anchors the meaning of concrete languages for the representation of proofs to structural proof theory. Structural proof theory is the mathematical study of mathematical proofs, pioneered by the works of Gentzen. We show how FPC are expressive enough to account for the direct interpretation of two common preprocessing steps performed by theorem provers: Skolemization and Tseitin transformations. Usually, a user who wishes to include in their work proofs produced by an automated prover employing these techniques would have needed to include additional axioms or trust in some other way these procedures, while we show a way to interpret these proofs directly as proofs of the original statement, before the transformations. Then, we show some properties of the proof theory of logics with fixed points that advance the foundations needed to define the meaning of proofs involving induction in terms of FPC. Finally, we present the development of two plugins for the Coq proof assistant that integrate the treatment of Foundational Proof Certificates in the interactive process of proving theorems with this proof assistant.

Institut Polytechnique de Paris
91120 Palaiseau, France