Pontifícia Universidade Católica
do Rio de Janeiro

Bruno Cuconato Claro

A labelled Natural Deduction Logical Framework

Tese de Doutorado

Thesis presented to the Programa de Pós–graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Doutor em Ciências — Informática.

Advisor: Prof. Edward Hermann Haeusler

Rio de Janeiro
September 2023

Pontifícia Universidade Católica
do Rio de Janeiro

Bruno Cuconato Claro

A labelled Natural Deduction Logical Framework

Thesis presented to the Programa de Pós–graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Doutor em Informática. Approved by the Examination Committee:

Prof. Edward Hermann Haeusler
Advisor
Pontifícia Universidade Católica do Rio de Janeiro – PUC-Rio

Prof. Jefferson de Barros Santos
FGV

Prof. Mario Roberto Folhadela Benevides
UFF

Prof. Bruno Lopes Vieira
UFF

Prof. Jean-Baptiste Joinet
Lyon III

Prof. Luiz Carlos Pinheiro Dias Pereira
UERJ

Rio de Janeiro, September 15th, 2023

Bruno Cuconato Claro

Bruno developed a passion for Computer Science topics during his undergraduate studies at FGV's School of Economics and Finance (EPGE), where he obtained a BSc in Economics with a minor in international relations. He went on to pursue a Master's at FGV's School of Applied Mathematics (EMAp), writing a computational grammar for the Portuguese language after having worked on other natural language processing tasks. It was through computational linguistics that Bruno's interest in logic was sparked, leading him to pursue his PhD in Logic at PUC-Rio. During his PhD, Bruno worked for at a time at IBM Research, and taught introduction to programming classes at FGV.

Bibliographic data

Cuconato Claro, Bruno

A labelled Natural Deduction Logical Framework / Bruno Cuconato Claro; advisor: Edward Hermann Haeusler. – 2023.

119 f: il. ; 30 cm

Tese (doutorado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2023.

Inclui bibliografia

1. Lógica – Teses. 2. Dedução Natural – Teses. 3. Arcabouço Lógico. 4. Framework Lógico. 5. Dedução Natural. 6. Sistemas Dedutivos Rotulados. 7. Sistemas Lógicos Rotulados. 8. Assistente de Provas. 9. Haskell. I. Haeusler, Edward Hermann. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Acknowledgments

First I would like to thank the people without whom this thesis would not have been: my advisor for the guidance and patience, and my parents for the gift of life and for their support.

I thank all of my family for the conversations and for their understanding: my siblings, my cousins, my aunts & uncles, and my beloved godmother. My friends too provided unwavering support; I thank them all, specially: Bia, who in trying to understand what I was doing unknowingly helped me write Section 1.1; Inari, who ended her PhD journey just as I was starting mine, and was a constant great listener and life-saver; and Kátia, whose PhD also overlapped a bit with mine and who has been a great friend for more than ten years now. My girlfriend arrived late in this PhD journey, but I do not know whether I could have finished without her — I might have burned out before. Thank you very much, my darling.

I also thank my co-advisees in the TecMF laboratory for the camaraderie overt the years. Bernardo — who started his PhD in the same year as me — was always one step ahead, and so answered many of my questions over the years; sorry about that and thank you! I also thank the overall TecMF family, Bruno Lopes and Jefferson Santos in special for their guidance and advice.

I am also thankful to PUC-Rio for this opportunity and for the environment that I became a part of. (I am explicitly **not** thankful for the pandemic that threw a wrench to this environment for a good stretch of time.) I thank all the professors of the Department for their teaching, in special Roberto Ierusalimschy, Noemi Rodriguez, and Sergio Lifschitz. I also thank professor Philip Wadler for the short courses he taught at PUC-Rio. Finally, I thank all of the people who support PUC-Rio and its students & researchers, the Department secretariat (thank you, Cosme!), the librarians, the maintenance & cleaning & cooking staff, thank you all.

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior — Brasil (CAPES) — Finance Code 001. I also thank CNPq for the financial support (Grant no. 157833/2019–4).

Abstract

Cuconato Claro, Bruno; Haeusler, Edward Hermann (Advisor). **A labelled Natural Deduction Logical Framework**. Rio de Janeiro, 2023. 119p. Tese de Doutorado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

We propose a Logical Framework for labelled Natural Deduction systems. Its meta-language is based on a generalization of the rule schemas proposed by Prawitz, and the use of labels allows the definition of intentional logics, such as Modal Logic and Description Logic, as well as some quantifiers, such as Keisler's "for non-denumerable-many individuals property P", or "for almost all individuals P holds", or "generally P holds", not to mention standard first-order logic quantifiers, all in a uniform way.

We also show an implementation of this framework as a freely-available web-based proof assistant. We then compare the definition of logical systems in our implementation and in other proof assistants — Agda, Isabelle, Lean, Metamath. As a sub-product of this comparison experiment, we contribute a formal proof (in Lean) of De Zolt's postulate for three dimensions, using the Z_p system proposed by Giovaninni *et al.*

Keywords

Logical Framework; Natural Deduction; Labelled Deductive Systems; Labelled Logical Systems; Proof assistant; Haskell.

Resumo

Cuconato Claro, Bruno; Haeusler, Edward Hermann. **Um framework lógico para Dedução Natural rotulada**. Rio de Janeiro, 2023. 119p. Tese de Doutorado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Neste trabalho propomos um framework lógico para sistemas de Dedução Natural rotulados. Sua meta-linguagem é baseada numa generalização dos esquemas de regras propostos por Prawitz, e o uso de rótulos permite a definição de lógicas intencionais como lógicas modais e de descrição, bem como a definição uniforme de quantificadores como o "para um número não-enumerável de indivíduos vale a propriedade P" (lógica de Keisler), ou "para quase todos os indivíduos vale P" (lógica de ultra-filtros), sem mencionar os quantificadores padrões de lógica de primeira-ordem.

Mostramos também a implementação deste framework em um assistente de prova virtual disponível livremente na web, e comparamos a definição de sistemas lógicos nele com o mesmo feito em outros assistentes — Agda, Isabelle, Lean, Metamath. Como subproduto deste experimento comparativo, também contribuímos uma prova formal em Lean do postulado de Zolt em três dimensões usando o sistema Z_p proposto por Giovaninni *et al.*

Palavras-chave

Arcabouço Lógico; Framework Lógico; Dedução Natural; Sistemas Dedutivos Rotulados; Sistemas Lógicos Rotulados; Assistente de Provas; Haskell.

Table of contents

List of Abreviations

LF — Logical Framework

GLF — the logical framework presented in this thesis

LDS — Labelled Deductive Systems

LCF — Logic of Computable Functions

ND — Natural Deduction

FOL — First-order logic

CTL — Computation Tree Logic

RAA — *reductio ad absurdum*

MP — *modus ponens*

CIC — Calculus of Inductive Constructions

C'est même des hypothèses simples qu'il faut le plus se défier, parce que ce sont celles qui ont le plus de chances de passer inaperçues.

Henri Poincaré, *Thermodynamique (1892).*

1
Introduction

1.1
Motivation

This section is intended as an introduction to the overall topic of the thesis, and is targeted at a more general audience. Specialist readers are welcome to jump ahead to Section 1.2.

Why do humans try to prove things? Sometimes we want or need to be sure of things. Often the motivation for proving things is curiosity, as is the case for most of the proving going on in mathematics, but there are also engineering applications to proofs: very expensive and difficult-to-service equipment like space probes may demand proofs of reliable operation, as may do tools that may put human lives in danger if they fail.[1] Think of the semaphore component of a rail-road system; it is not enough to *think* it has been well-designed, we want to be *sure* that it does not signal for two vehicles to go ahead and crash, and one way to do this is to design it so that we can prove this never happens.

Unlike what laypeople might think, Logic as a discipline is not the study of Truth. In their quest for truth and certainty, humans early on noticed that depending on the assumptions you made, you could reasonably reach any conclusions. Therefore Logic as a discipline has focused not on truth but on studying correct reasoning — independent of any hypotheses made.

Logic is the theoretical basis of any proof, even informal ones. The proofs we will be talking about in this text are more of the formal kind: they are made according to a logic system, a strict set of rules concerning what makes a proof correct — and so what kind of deductions one can make.

We speak of **Logic** (often written with a capital L) as the discipline or overall field, but logicians study and propose different **logics** (always written with a lower-case l). Each one reflects a different perspective of what

[1] Proofs are, of course, the ultimate level of certainty; whenever it is unreachable, other methods (such as extensive testing) are used to ascertain a high degree of confidence in a particular behaviour

constitutes valid reasoning: e.g., most mathematicians are content to accept as a proof a reasoning that depends on an assumption that a proposition is either true or it is false, even if it is not known which it is. Some other mathematicians disagree, saying that for such a reasoning to be valid we also have to show whether the proposition is true or false, that it does not suffice to show that it must be one or the other without knowing which one is the case. Each of these views gives rise to different logics, with different valid statements, i.e., with some things being true in one logic but not the other. Different logics offer different "views of the world": a logic used for mathematical reasoning usually does not care about time; mathematical truths are true regardless of what time it is, after all. But a logic used to reason about a semaphore system might make temporal statements possible, so we can speak about a condition being true at some point in time (e.g., the semaphore signalling STOP at time t_0) but not necessarily being true at other times.[2]

Logical systems are mechanizations of reasoning according to a certain logic. They propose rules that when followed strictly allow us to prove truthful deductions according to a certain system. The correspondence between the proof of an assertion and its truth within the system is of course proved too, and is called a correctness proof.

The idea of mechanizing reasoning is fruitful because human minds are prone to interpolation and thus often elide important (and sometimes not so important) details in informal proofs. For some applications this is no problem — most mathematical proofs are at least to a certain degree informal — but for some others full proofs are best required. The problem with formal proofs is that because they require every step to be painstakingly detailed, they are often much longer and harder to obtain than informal proofs. The 'longer' part is necessary,[3] but the 'harder' is not: it may be due to it sometimes being monotonous to make all the proof steps explicit, but a part of its hardness might come from the fact that reasoning this strictly is too different from how humans usually think.

Unlike humans, however, computers 'think' in exactly this strict way. Applying rules mindlessly is what they were designed to. Therefore it is natural to suggest that we implement logical systems in computers,[4] so that humans can be assisted in proving (hence **interactive theorem proving**), or computers can do the proving themselves (**automatic theorem proving**).

[2]One can make a small and imprecise analogy with natural languages: ancient Tupi did not have a word for 'snow' (although it did have one for 'frost'), while Finnish has more than ten words related snow and other phenomena related to water's solid phase.

[3]Although this need not impact the user much: the 'obvious' steps of a proof can sometimes be obtained automatically, and hidden from the user's view if deemed helpful.

[4]Indeed it has been tried since the 1950s at least.

Programming a logical system can be hard. Not only the system itself can be complex, but there is also a high standard for the correctness of this code: since it is used to prove things so one can be sure of them, one should not have their proof be wrong because the implementation of the system has an error.[5] As we will discuss later on, the interface between the user and the implementation of the logical system can be another source of complexity. There have been several different designs, and none seem to be without its flaws.

A partial solution to these problems is the idea of **logical frameworks** (LFs). LFs are meta-languages developed to describe logical systems; in layman's terms, we can say they are building blocks that can be combined to implement different logical systems.[6] When we use a LF to implement a logical system, we can reuse several of its component parts (like the parser or the printer or the evaluator), reducing our workload and thus our costs. If a logical framework has become popular and has been used by several people in different projects, we can more safely assume that most programming bugs have been discovered and reported (and hopefully fixed too). Although this does not eliminate the risk of computer bugs causing problems in the logical proofs, it certainly reduces it. The use of logical frameworks also offers benefits with respect to the matter of user interfaces. Although different LFs will have different interfaces (and all of them are imperfect), at least all the logical systems implemented in the same LF will have a consistent interface, which is useful since learning a different interface for each system would be challenge. All in all, logical frameworks are about obtaining implementations of logical systems at smaller costs (of both time and money) and with better safety guarantees *vis-à-vis* implementing them independently.

1.2 About this thesis

In this text we propose a new logical framework for labelled Natural Deduction systems, named GLF. Although many LFs have been already been proposed, we believe there is a gap in the design space for a framework & software implementation with a simple meta-language that can be used without programming language knowledge, and that strives to make the definition of logical systems — and deductions in them — as close as possible to their traditional paper versions.

[5] Many of the programs we use in our day-to-day do not have this concern; it is perfectly fine if a social media app fails from time to time.

[6] LFs also have theoretical advantages of which we will not talk about for the moment.

In Chapter 2 we give the reader some of the necessary background to understand this text: we introduce Natural Deduction, explain what labelled logical systems are, and discuss logical frameworks in more depth.

In Chapter 3 we advance to the core of this thesis. We describe our proposed framework, and show how it can embed logical systems with complex rules and labelling schemes. Here we take inspiration from the seminal work of Prawitz [1] and some of its offshoots in defining introduction and elimination rule schemas that serve as GLF's meta-language. We thus contribute schemas for labelled systems, including an appropriate algebra for label modification.

It would be impossible to write about labelled logical systems without mentioning Gabbay's work [2, 3] in disseminating them. The work we carry out here can be seen as a specialization of his proposed LDS framework; much like Viganò specializes it to non-classical logics [4], here we specialize it to Natural Deduction systems. In both cases the specialization allows the specification of an algebra for the labels in the systems, which in our case also helps the implementation.

Another inspiration for our framework was Rentería's collection of Natural Deduction systems [5] that uses labels to handle general quantifiers. We differ from Rentería by using the aforementioned rule schemas, which includes a unified labelling scheme that Rentería's work lacks. Unlike him, we also provide a concrete implementation for the systems discussed in this thesis, available at `https://glf.tecmf.inf.puc-rio.br/`. This implementation builds upon our previous work [6, 7].

One of the foremost investigations carried out in this thesis is how general GLF's meta-language is; i.e., how wide a range of logical systems can be expressed within it. As an answer to this question, in Chapter 4 we describe several logical systems for different logics, and show that they conform to our LF's rule schemas. The logical systems we describe include all of the ones from Rentería's work, but also include a Natural Deduction system for iALC described by Alkmim [8]; a system for propositional modal logic K; and a system for Z_p, a weak type theory (the latter two are described in Chapter 6). The diversity of systems implemented in GLF demonstrates in practice that it can handle a wide variety of logical systems, including those with unusual quantifiers and those with non-trivial labelling schemes.

Chapter 5 describes the software implementation of our logical framework for labelled Natural Deduction systems. We also make a brief overview of related work in web-based user interfaces for logical systems. Finally, we show in Chapter 6 two experiments comparing the implementation of two logical systems in both our LF and in other proof assistants, namely: Agda, Isabelle,

Lean, and Metamath. The latter experiment involved the implementation of the Z_p type theory system proposed by Giovannini *et al.* [9] in Lean and in GLF; as a sub-product of this work we contribute a new formal proof of a three-dimension version of De Zolt's postulate in the Z_p system (see Section 6.2).

Chapter 7 wraps up this text with some concluding remarks, including a recapitulation of our contributions, some limitations of GLF, and future work.

About notation Throughout this thesis, we use upper-case Latin letters to represent formulas in rule schemas, but use lower-case Greek letters in all other contexts, including for formulas in deductive rules and in axioms. Therefore we would write the *modus ponens* inference rule as $\varphi \to \psi, \varphi \ / \ \psi$. If a formula is known or required to have a certain form, we write it in this form, using lower-case Greek letters for the parts that are not fixed. E.g., the major premise of the *modus ponens* rule must be an implication, so we may write it as $\varphi \to \psi$. Note that when we write φ for a formula, it could equally well be a propositional letter, a quantified formula like $\forall x.\psi$, or any formula admitted by the logical language in question. Both propositional letters and variables are written with lower-case Latin letters, usually p, q, r, s and x, y, z, respectively.

When writing labelled formulas, we put the formula first and the label second, with the : symbol separating them. Labels are represented by lower-case Latin letters, usually l, m, u, v, w. Thus $\phi : l$ is the formula ϕ with label l. A formula can only have a single label, and the colon operator binds as loosely as possible, so that it always sits at the root of its parse tree.[7]

[7]The sole exception is in the case of iALC formula syntax, in which ∶ binds with even lower precedence (see Section 4.7).

2
Background

If Logic (as a field) is the study of correct reasoning, we can see *a* logic as an instance of a theory of correct reasoning. A logical system would then be a tool to carry out correct reasoning mechanically, following a fixed set of rules, in accordance with a specific logic. This 'accordance' is defined by two properties: completeness and correctness. Completeness guarantees that every valid formula of the logic is provable in the logical system in question (i.e., is a theorem in it), while correctness establishes that every theorem provable in the logical system is a valid formula of the logic. A system that is complete but not correct with respect to a logic can prove all of its valid formulas, but also some propositions that are invalid. Conversely, a system that is correct but not complete with respect to a logic will only prove valid formulas of the logic, but is not capable of proving *all* of its valid formulas, only some of them.

Logical systems are specially interesting at the intersection of logic and computer science. To create a proof in a logical system one needs to carry out specific instructions mechanically, an activity most people have difficulties at performing, but at which computers excel. This leads to an interest in software implementations of logical systems, which eventually lead to logical frameworks (in one sense of the word). We introduce logical frameworks in Section 2.3.

Different logical systems may nonetheless follow a common style. In this thesis we propose a logical framework for Natural Deduction, which is one of these styles. More specifically, our logical framework is tailored to labelled Natural Deduction systems, which we explain in Section 2.2. We now introduce Natural Deduction, with some comments about another style of logical systems — axiomatic systems; other styles exist, but we will not discuss them here.

2.1
Natural Deduction

The main characteristic of Natural Deduction (ND) is making hypothetical reasoning explicit [10, 11].

There are other secondary characteristics of Natural Deduction that help distinguish it (some of which we will discuss shortly), but they are either not exclusive to Natural Deduction systems or are not shared by all

of them. As Pelletier [12, §2] argues, the absence of axioms may be presented as characteristic of Natural Deduction, but some Natural Deduction systems do have axioms or axiom-like constructs like tautologies or axiomatic rules. Similarly, it is often said that the 'naturalness' of the rules of a Natural Deduction system is its main characteristic, but one can argue that rules in other kinds of logical systems are just as natural.

Natural Deduction was proposed by Jaśkowski (working on a lead by Łukasiewicz) in the late 1920's [13], with his definitive paper being published in 1934, the same year Gentzen independently published his work on Natural Deduction (see Prawitz [14, App. C, §1] and Pelletier [12]). Before then, axiomatic (or Hilbert-style) systems were the only ones available. Axiomatic systems have their advantages, but they are very different from the way informal reasoning and informal proofs look like. For example, see the proofs of the tautology $(q \to r) \to ((p \to q) \to (p \to r))$, in a Hilbert-style system (Figure 2.2, see Church [15, §20] for details) and in a Natural Deduction system (Figure 2.1). Figure 2.2 also features the two axiom schemas of the system we use for the example.

Another characteristic of Natural Deduction systems is that we can separate (most) of their logical rules into introduction and elimination rules. As Prawitz notes in the preface to his work [14], they are supposed to correspond to each other according to a certain inversion principle. We say "supposed" because while this inversion is clear in the rules for disjunction, conjunction and implication, a rule like *reductio ad absurdum* has no clear inverse. The same observation goes to rules that are axiomatic in nature (like some we will see in Chapter 4) and to those which are non-logical in nature (e.g., structural rules), although these exceptions can perhaps be excused due to their nature.

To understand the overall form of a Natural Deduction proof in the style of Gentzen–Prawitz we take Figure 2.1 as an example. A natural deduction starts with assumptions (the formulas at the trees' leaves, with no horizontal lines above them). On these assumptions we apply deduction rules, obtaining other formulas (we may call them consequences as Prawitz does, or conclusions). The rule application is shown by writing a horizontal line below the rule's premises, and its consequence (or conclusion) is shown below the line. Continuing in this fashion, by using new assumptions or conclusions of previous rule applications to apply new rules, we may eventually be able to prove the goal formula we want, reaching the tree's root at the same time. With this we have proved the goal formula from the set of assumptions we used. Some rules allow us to discharge assumptions, however, as we can see in Figure 2.1. A discharged assumption means that the proof does not depend on

$$\dfrac{\dfrac{\dfrac{\dfrac{[p]^1 \quad [p \to q]^2}{q}\to\text{E} \qquad [q \to r]^3}{r}\to\text{E}}{p \to r}\to\text{I}_1}{(p \to q) \to (p \to r)}\to\text{I}_2}{(q \to r) \to ((p \to q) \to (p \to r))}\to\text{I}_3$$

Figure 2.1: Natural Deduction proof of $(q \to r) \to ((p \to q) \to (p \to r))$

Axiom schemas:

(A1) $A \to (B \to A)$
(A2) $(A \to (B \to C)) \to ((A \to B) \to (A \to C))$

Proof.

(a)	$((p\to(q\to r))\to((p\to q)\to(p\to r)))\to((q\to r)\to((p\to(q\to r))\to((p\to q)\to(p\to r))))$	(A1)
(b)	$(p \to (q \to r)) \to ((p \to q) \to (p \to r))$	(A2)
(c)	$(q \to r) \to ((p \to (q \to r)) \to ((p \to q) \to (p \to r)))$	(MP a,b)
(d)	$((q\to r)\to((p\to(q\to r))\to((p\to q)\to(p\to r))))\to(((q\to r)\to(p\to(q\to r)))\to((q\to r)\to((p\to q)\to(p\to r))))$	(A2)
(e)	$((q \to r) \to (p \to (q \to r))) \to ((q \to r) \to ((p \to q) \to (p \to r)))$	(MP d,c)
(f)	$(q \to r) \to (p \to (q \to r))$	(A1)
(g)	$(q \to r) \to ((p \to q) \to (p \to r))$	(MP e,f)

□

Figure 2.2: Axiomatic proof of $(q \to r) \to ((p \to q) \to (p \to r))$

that assumption; because in Figure 2.1 we are proving a tautology, there are no undischarged assumptions there — if there were, we would not have a proof, but a deduction from the undischarged assumptions to the formula we reach at the root of the deduction tree. For ease of reading, we surround the discharged assumptions with square brackets to denote their status, and we additionally label them with a number that also appears at the site of the rule application that discharged them.

In the first chapter of his seminal monograph, Prawitz [14] defines Gentzen-style Natural Deduction systems formally. Here we give a similar account of such systems, based on his work. Because deductions are trees of formulas, we first need a definition for formula-trees.

Definition 1 (Formula-trees)**.** Π *is a formula-tree if and only if* Π *is a formula or* Π *is of the form* $\Pi_1, \Pi_2 \ldots \Pi_n \;/\; \varphi$*, where* $\Pi_1, \Pi_2 \ldots \Pi_n$ *is a sequence of*

formula-trees and φ is formula. For the latter case, we could also write

$$\frac{\Pi_1 \quad \Pi_2 \quad \ldots}{\varphi}$$

for a more diagrammatic view.

Take the deduction in Figure 2.1. The whole proof is a formula-tree, but given the recursive definition of formula-trees, so are its sub-deductions. As an example, the formula-tree ending at r (before the first application of the implication introduction rule) is of form $\Pi_1, \Pi_2 \,/\, r$, with Π_1 being the sub-tree rooted at the premise q, and Π_2 being the singleton sub-tree $q \to r$.

Now that we have a definition for the structure of a deduction, we need definitions that guarantee their well-formedness. After all, one can easily write any formula-tree one wants, but not all of them will be valid deductions in a given system. It is only when a formula-tree conforms to a system's axioms and rules that it is considered a deduction in that system.

In his monograph Prawitz describes two related concepts he calls inference and deductive rules (or improper inference rules). An inference rule is of the form $A_1, A_2, \ldots, A_n \,/\, B$. The sequence of formulas $A_1, A_2, \ldots, A_n$ are the rule's premisses, while the formula B is the rule's consequence. The implication elimination rule is an instance of an inference rule, with $n = 2$: A_1 takes the form $\chi \to \varphi$ and A_2 takes the form χ, with B being equated with φ. We thus write the $\to$E rule as: $\chi \to \varphi, \chi \,/\, \varphi$.

Inference rules do not involve any kind of assumption taking/discarding, as their general form shows. That is what differs them from improper inference rules, or deduction rules. Deduction rules are of the general form $\langle\langle\Gamma_1, A_1\rangle, \langle\Gamma_2, A_2\rangle, \ldots \langle\Gamma_n, A_n\rangle, \langle\Delta, B\rangle\rangle$. The implication introduction rule is an instance of a deduction rule, with $n = 1$: the premise $\langle\Gamma_1, A_1\rangle$ is equated with $\langle\Theta, \varphi\rangle$ (for some set of assumptions Θ), and the consequence $\langle\Delta, B\rangle$ takes the form of $\langle\Theta \setminus \{\chi\}, \chi \to \varphi\rangle$.

Because proper inference rules do not care about assumptions, we may think of them as being local in nature, while deductive rules are non-local, having whole deductions as (at least some of their) premises. We could also think of inference rules as deductive rules which do not change (or care about) the set of assumptions made to reach any given conclusion, unifying their definitions.

Now that we know what inference and deductive rules are, we are ready to define Natural Deduction logical systems. Prawitz takes a system of Natural Deduction to be a triple of a set of inference rules, a set of deductive rules,

and a set of axioms.[1] A deduction in such a system is defined as follows:

Definition 2 (Deduction)**.** Π *is a deduction in a system* S *of a formula* φ *depending on a set* Γ *of assumptions iff* Π *is a formula-tree and:*

1. *if* φ *is not an axiom in* S*, then* φ *is a deduction in* S *of* φ *depending on* $\{\varphi\}$*, or*

2. *if* φ *is an axiom in* S*, then* φ *is a deduction in* S *depending on the empty set* $\{\}$*, or*

3. *if* Π_i *is a deduction in* S *of* φ_i *depending on* Γ_i *for* $i \leq n$*, then* $\Pi_1, \Pi_2, \ldots, \Pi_n \;/\; \psi$ *is a deduction of* ψ *in* S *depending on* Δ*, as long as:*

   (a) $\varphi_1, \varphi_2, \ldots, \varphi_n \;/\; \psi$ *is a proper inference rule in* S*, and* $\Delta = \cup_{i=1}^{n} \Gamma_i$*, or*

   (b) $\langle\langle \Gamma_1, \varphi_1\rangle, \langle \Gamma_2, \varphi_2\rangle, \ldots \langle \Gamma_n, \varphi_n\rangle, \langle \Delta, \psi\rangle\rangle$ *is an instance of a deduction rule in* S*.*

We can of course say that Π is a deduction in S of φ from Γ even if not all assumptions in Γ are utilized (including none of them). If we have a deduction of φ that depends on no assumptions at all we say we have a proof of φ. A deduction in S from Γ concluding φ is written $\Gamma \vdash_S \varphi$; we will omit the subscript denoting the system when the context clarifies of which system we are talking about, or when we are talking about an arbitrary system.

2.2 Labelled Natural Deduction systems

We have just seen the Prawitz definition of Gentzen-style Natural Deduction systems. In this thesis we will be discussing a variant of such a style of logical system, namely that of labelled Natural Deduction systems.

Not all labelled logical systems are Natural Deduction systems. Labelled logical systems in general are systems where the syntax of formulas is extended to include a label annotation, and the system's rules are likewise changed to be able to 'act' on them. The labels should form an algebra that the system's rules then manipulate according to its rules. As De Queiroz and Gabbay [16] put it, "a logical system is taken to be not simply a calculus of logical deductions on formulae, but a suitably harmonious combination of a functional calculus on the labels and a logical calculus on the formulae." Adding labels to a deductive

[1] Axioms may be seen as inference rules with no premises.

systems allows us to include meta-theoretical aspects of the system's logic's consequence relation into the system itself, which may materialize into fewer rule provisos (or at least into simpler provisos).[2]

One can think of type theory systems as labelled systems, since their formulas are terms annotated by their types, and the same can be thought of modal systems; it helps if we write $w \models \phi$ (ϕ is true at world w) as $\phi : w$.[3] Indeed, in his thesis [17] Simpson does just that, proposing labelled Natural Deduction systems for intuitionistic modal logics. See also Van Benthem [18] for another example. Despite their prevalence, type theory and modal systems are not the only early labelled systems; according to Gabbay the idea is already present in Anderson's and Belnap's 1975 book Entailment [19], and an even earlier 1962 collaboration of the two already uses something akin to labels [20]. Labelled systems have also found applications in expert systems using Fuzzy logic, in provenance systems,[4] and in temporal logic applications, among others [3, pp. 13–16].

To give a more detailed example, we may put forward a labelled system for relevance logic modelled on the one used by Gabbay [3, Ex. 2.2.1]. In this system, any assumption occurrences must be labelled by a different atomic symbol (which symbol it is does not matter, as long as it unique). The only logical connective is the implication, and the only rules are implication introduction and elimination. These two rules are slightly different from their usual formulations, given the presence of the labels; see Figure 2.3, where l, m are sets of atomic labels, a is an atomic label, and $\cup, \setminus$ represent set union and set difference, respectively. The premises of the two rules are unchanged save for the labelling on their formulas, but the conclusion of the implication elimination rule has a label that is the set union of the labels of its premises, while the conclusion of the implication introduction rule has the same label as its premise, minus the atom that is the label of the assumption discharged by the rule, with the proviso that the rule can only be applied if this atom is present in the label of the premise.

Early labelled logical systems introduced labels in an *ad hoc* manner, because they were deemed necessary or useful. Later, noting their increasing prevalence, Gabbay [2, 3] proposed a general framework which would subsume most of them.

Later on, Viganò would specialize this idea, proposing his own labelled

[2]Points similar to these are made by Gabbay [2, §1] and by Viganò [4, §1.2.1].

[3]A small note about notation: many authors in Logic write the label before the formula, but in this work I choose to write formulas first.

[4]Provenance systems keep track of information (such as the changing ownership of artwork over time), and also where this information came from — its sources.

$$\frac{\varphi \to \psi : l \quad \varphi : m}{\psi : l \cup m} \to\text{E} \qquad\qquad a \in l \, \frac{\begin{array}{c}[\varphi : \{a\}] \\ \vdots \\ \psi : l\end{array}}{\varphi \to \psi : l \setminus \{a\}} \to\text{I}$$

Figure 2.3: Implication rules in a labelled relevance logic system

framework for non-classical logics [4]. In restricting the scope of the original framework by Gabbay, Viganò is able to further specialize the labelling algebra to be that of Horn relational theories, and to provide an implementation of several systems in Isabelle [21].

In this work we too provide a specialization of the idea of a general framework for labelled deductive systems. Instead of focusing on a subset of logics like Viganò does for non-classical logics, we focus on a subset of deductive systems (namely Natural Deduction systems). Much like Viganò, this allows us to fix a single algebra for our system's labels (as we will see in Chapter 3). Choosing a subset of deductive systems also helps us provide an implementation of several systems with a consistent user interface.

2.3 Logical Frameworks

Logical frameworks (LFs) are meta-languages used to specify deductive systems, as Pfenning [22] puts it. Logical frameworks typically allow us to carry out derivations in the implemented deductive system and to implement algorithms that solve a problem related to the implemented deductive system. Some frameworks even allow the user to investigate the meta-theory of the implemented systems (e.g., proving that a system possesses a particular property).

Many logical frameworks have been born from attempts of answering the question of '*what is logic?*' [23, 24, 25, 26]. Incidentally, these tentative answers have provided theoretical frameworks that aim to describe any logical system. In proposing our own logical framework, we do not purport to answer the question of what logic is, but we do intend to produce a framework able to describe a large class of logical systems, and use it in practice for that end.

Logical frameworks can be purely theoretical constructs, but may also refer to software artifacts. Huet and Plotkin [27] distinguish these two senses in which the term *logical framework* is used:

> First, very many logics are of interest in Computer Science, and great repetition of effort is involved in implementing each. It would therefore be helpful to create a single framework, a kind of meta-logic, which is itself implementable and in which the logics of interest can be represented.
>
> In the second sense, one chooses a particular "universal" logic which is strong enough to do all that is required, and sticks to it. [...] Even within a fixed logic there is the need for a descriptive apparatus for particular mathematical theories, notations, derived rules and so on.

In this thesis we focus on the first sense of the word, although some LFs can be interpreted in both senses. Following this first sense, logical frameworks are often implemented as a core component of software tools such as proof assistants, theorem provers, or proof checkers.

We will be discussing some of these possible applications of logical frameworks later on, so we will explain what they are here. Proof assistants are software tools that help humans construct formal proofs; they incorporate proof checkers to verify that proofs (or ongoing proofs) are correct, but usually offer more than this, giving information about open goals of an ongoing proof, or even giving a certain degree of automation by proving sub-goals automatically. Proof assistants may also be called interactive theorem provers, but the term theorem prover in general also includes automatic theorem provers, and those are what is usually meant when the word is used without any qualifications. Automatic theorem provers, as their name says, are tools to automatically prove logical propositions; they are non-interactive, requiring only the bare minimum input such as which proposition to attempt to prove and in which logic, and optionally what strategies to use and several parameters that guide and restrict the proof search. By contrast, proof checkers simply check existing proofs supplied by the user. They can stand independently, but are often a component of proof assistants and of automatic theorem provers.

It is important to note that not all proof assistants/automatic theorem provers/proof checkers make use of logical frameworks as their theoretical underpinnings. Using logical frameworks as a base is a way of making these tools more general, so that they can handle several logical systems more easily and in a uniform way, but one can still build a proof assistant or proof checker or automatic theorem prover for a single logical system without the use of a logical framework.[5] Note that throughout this thesis we may refer to

[5]One can also have proof assistants or proof checkers or automatic theorem provers for

proof assistants that are based on logical frameworks as logical frameworks themselves.

By making logical applications such as proof assistants more general and uniform, logical frameworks reduce the cost of implementing deductive systems. This is because there is now a common base upon which all systems can build. Often this common base will amount to a common formula representation and a shared mechanism for hypothesis introduction & discharge, but it may go farther and provide core rules that other rules must be defined with (as is the case of Metamath [28], for example).

There is another advantage to having a shared code-base besides reducing implementation costs: since logical frameworks are critical software in which bugs are unacceptable (unlike, say, a social media app, or a video game), it is common practice to have a small kernel where the main logic of the application code lies, and that is easily verifiable by third parties. The more deductive systems implemented using the same kernel, the more robust the LF's kernel becomes, which benefits all the superjacent systems.

Logical frameworks have a long history in computer science, beginning with project Automath in 1966 (see de Bruijn [29] for a survey on Automath).[6] Automath pioneered an idea that has continued to be a core part of the design of several logical frameworks. According to Geuvers [30], it was the first system to get inspiration from the Curry–Howard correspondence (or the formula-as-types interpretation), making proofs be first-class terms of the system, and using the type checker as the proof checker — since Curry–Howard guarantees they correspond.

The application of the Curry–Howard correspondence to logical frameworks led to a flurry of activity in type–theory-based proof assistants that continues to this today. A very influential early system is confusingly named LF (created by Harper *et al.* [31]), and it has a few direct descendants like Twelf by Pfenning and Schürmann [32]. Another early influential system is Coq [33, 34], underpinned by Coquand's Calculus of Inductive Constructions [35, 36]. Although first released in 1989, it is still actively used nowadays, and has also helped inspire a few other dependent type theory-based proof assistants: Idris by Brady [37], Agda by Norell [38] (based on Martin-Löf's intuitionistic type theory [39, 40]), and Lean by de Moura [41, 42] are notable examples. Both

several logical systems without using a logical framework, but one can argue that these are actually separate implementations joined together giving the impression of a monolithic whole.

[6] There are even earlier attempts at automating proofs using computers, for instance the Logic Theorist automated theorem prover by Allen Newell, Herbert A. Simon, and Cliff Shaw.

Agda and Lean will feature later in this text, in Section 6.1.1 and Section 6.2.2, respectively.

Another landmark in the history of LFs in Geuvers' view [30] is the 'LCF approach' that inspired Isabelle [21], HOL, and HOL-light. The LCF approach is paradigmatic of the idea of having a small trusted kernel in a logical framework: LCF-like system have an abstract data type for theorems, with axioms being the only constants of this type, and inference rules being the only functions that output this 'theorem type'. The user may write elaborate tactics combining different functions, but in the end what they have is a combination of the elementary pieces (axioms and inference rules), meaning the system is correct by construction. See Geuvers for a more detailed history of logical frameworks and proof assistants [30, §2].

Despite this long history and the existence of several logical frameworks with different meta-languages, their use by practitioners (e.g., mathematicians) is still rare. Buzzard et al.'s formalization of perfectoid spaces [43] seems to be the first formalization of contemporary advanced mathematics (in that it involves a sophisticated object only taught at graduate-level) in a logical framework. Most other formalization achievements concern more elementary objects, being notorious for improving on the original proof (e.g., the proof of the four-color theorem by Gonthier [44]), or for simply tackling contemporary mathematics (e.g., Gouezel & Shchur's [45] formalization of the Morse lemma).

In the case of mathematicians, there are intrinsic barriers to the adoption of logical frameworks and their applications (namely, proof assistants). Gowers [46] gives an account of why mathematicians are ready to believe unproven but likely-true statements. Given this readiness, it stands to reason that mathematicians are not very interested in formal proofs of theorems they are reasonably sure of, since they need no extra confirmation/certainty. This leaves proof assistants with a smaller niche: they are seen as useful only in the cases where the truth value of a statement is still uncertain, and a regular proof seems difficult to believe in or even to understand. The proof of the four color theorem [44] would be a paradigmatic example of this, and is indeed one of the most well-known computer-assisted proofs. Of course, proof assistants can also be made attractive to mathematicians if they make their work easier, be it in communicating mathematics better/easier, or in helping them discover new proofs, or in making finding proofs faster. Note however that in this case the main attraction will not be the formality level of the proof afforded by the assistant.

There are extrinsic barriers to the adoption of logical frameworks too. Formalizing a pen-and-paper system or a proof is rarely a straightforward

matter. It is not that a lot of implicit details need to be filled in (this is expected, even if it might be surmountable), but that the subject of the formalization often needs to be adjusted to suit better the LF in question. This point is briefly brought up by Buzzard et al. in their formalization of perfectoid spaces [43, §10], and Allais' formalization of intuitionistic multiplicative-additive linear logic [47] is also an example of this phenomenon.

Another matter is that most LFs have a user interface problem, having been designed by computer scientists, but having a larger intended audience. Some of the most popular LFs like Coq are programming language environments, which may impose an entry barrier to some — and even experienced programmers sometimes have trouble with the kind of programming languages such LFs employ. The need to learn the LF's meta-language, especially when it is a sophisticated one, also adds to this barrier.

To the best of our knowledge, no investigation of why logical frameworks have not been widely adopted so far has been conducted, and the present work is not such an investigation. It is difficult to imagine LFs in widespread use before some of these shortcomings are surmounted, however.[7]

In the spirit of alleviating some of these shortcomings of LFs, we propose a logical framework and an accompanying proof assistant for labelled Natural Deduction systems. Our LF — GLF — uses a very simple meta-language based on rule schemas, and its proof assistant interface is as close as possible to a pen-and-paper proof. Similarly, we also strive for the same correspondence in the implementation of logical systems in our LF: they should look much the same as they do on the paper of a journal or of a textbook. For more details on the framework we propose, see the next chapter.

[7]Surmounting these problems is no guarantee of success, of course.

3
The Framework

Defining deductive systems in our logical framework amounts to defining their rules. Theoretically, rule definition is based on rule schemas inspired by Prawitz' [1] and built upon by Haeusler' [48, §2][49, §II.1] and Hao Chi [50, §3.1]. Schroeder-Heister [51] also proposed rule schemas based on Prawitz' seminal work, but they target his extension of Natural Deduction where both rules and formulas may be assumed and discharged instead of regular ND.

Prawitz was the first to devise rule schemas for the introduction/elimination of connectives in Natural Deduction systems, but his elimination schema was not strong enough to have the implication elimination rule as an instance. The later work by Haeusler and by Hao Chi fixes this shortcoming, as does our version. The major difference between the Prawitz–Haeusler–Chi rule schemas and ours is the presence of labels in the formulas, but for explanatory reasons we first show the unlabelled versions.

Figure 3.1 shows the unlabelled schema for introduction rules, while Figure 3.2 shows the unlabelled schema for elimination rules. Both unlabelled schemas are based on Haeusler [48, §2] and on Hao Chi [50, §3.1]. For the unlabelled schemas, they stipulate there must be at most one elimination rule, but there may be more than one introduction rule. The schemas below relax this restriction, which allows us to define more rules, such as those for the non-disjunction connective.[1]

As an example to aid the understanding of the rule schemas, take the implication connective. Its only introduction rule is as usual, but its elimination is a bit different from the common presentation (see Figure 3.3). In the case of the introduction rule, we have that $i = p_1 = j_1^1 = 1$, so we have ${}_1A_1^1 = \varphi$, ${}_1B_1 = \psi$, and $\mathbf{c}({}_1B_1, {}_1A_1^1) = \mathbf{c}(\psi, \varphi) = \varphi \to \psi$. For the elimination rule we have $e = n_1 = d_1 = 1$, with $\mathbf{c}({}_1B_1, {}_1A_1^1) = \mathbf{c}(\psi, \varphi) = \varphi \to \psi$, $H_1^1 = \varphi$, $\Gamma_1^1 = [\psi]$, and $C = \chi$.

This explanation about the unlabelled rule schemas follows Haeusler and Hao Chi more closely, but more details and motivation can be found

[1] Its rules are:
$\langle\langle\Gamma_1, \bot\rangle, \langle\Gamma_2, \bot\rangle, \langle(\Gamma_1 \setminus \{\varphi\}) \cup (\Gamma_2 \setminus \{\psi\}), \varphi \downarrow \psi\rangle\rangle$ (introduction)
$\langle\langle\Gamma_1, \varphi \downarrow \psi\rangle, \langle\Gamma_2, \varphi\rangle, \langle\Gamma_3, \chi\rangle, \langle\Gamma_1 \cup \Gamma_2 \cup (\Gamma_3 \setminus \{\bot\}), \chi\rangle\rangle$,
$\langle\langle\Gamma_1, \varphi \downarrow \psi\rangle, \langle\Gamma_2, \psi\rangle, \langle\Gamma_3, \chi\rangle, \langle\Gamma_1 \cup \Gamma_2 \cup (\Gamma_3 \setminus \{\bot\}), \chi\rangle\rangle$ (elimination)

$$\frac{\begin{array}{ccccc}\left[{}_iA_1^1\right],\ldots,\left[{}_iA_{j_1^i}^1\right] & & \left[{}_iA_1^l\right],\ldots,\left[{}_iA_{j_l^i}^l\right] & & \left[{}_iA_1^{p_i}\right],\ldots,\left[{}_iA_{j_{p_i}^i}^{p_i}\right] \\ \vdots & & \vdots & & \vdots \\ {}_iB_1 & \ldots & {}_iB_l & \ldots & {}_iB_{p_i}\end{array}}{\mathbf{c}(((({}_iB_k)_{k=1}^{p_i})_{i=1}^{s},(((({}_iA_m^l)_{m=1}^{j_l^i})_{l=1}^{p_i})_{i=1}^{s})}\,\mathbf{c}I_{\mathrm{i}}$$

where:

1. $\mathbf{c}$ is the logical connective being introduced
2. $i = 1,\ldots,s$ (the connective $\mathbf{c}$ has s introduction rules)
3. $l = 1,\ldots,p_i$ (the i-th introduction rule has p_i premises of the form ${}_iB_l$)
4. the l-th premise ${}_iB_l$ of the i-th introduction rule may have up to j_l^i discharged hypotheses of the form ${}_iA_m^l$

Figure 3.1: Introduction rule schema for a connective $\mathbf{c}$

$$\frac{\mathbf{c}(((({}_iB_k)_{k=1}^{p_i})_{i=1}^{s},(((({}_iA_m^l)_{m=1}^{j_l^i})_{l=1}^{p_i})_{i=1}^{s}) \quad H_1^e \quad \ldots \quad H_{n_e}^e \quad \begin{array}{c}\Gamma_1^e\\ \vdots\\ C\end{array} \quad \ldots \quad \begin{array}{c}\Gamma_g^e\\ \vdots\\ C\end{array} \quad \ldots \quad \begin{array}{c}\Gamma_{d_e}^e\\ \vdots\\ C\end{array}}{C}\,\mathbf{c}E_{\mathrm{e}}$$

where:

1. $\mathbf{c}$ is the logical connective being introduced
2. $e = 1,\ldots,r$ (the connective $\mathbf{c}$ has r elimination rules)
3. the e-th elimination rule has n_e minor premises and d_e discharging premises
4. Γ_g^e are sets of hypotheses that may be discharged by applications of the rule
5. $\uplus_{e=1}^{r}\uplus_{f=1}^{n_e}\{\!\{H_f^e\}\!\} = \uplus_{i=1}^{s}\uplus_{l=1}^{p_i}\uplus_{m=1}^{j_l^i}\{\!\{{}_iA_m^l\}\!\}$ and
   $\uplus_{e=1}^{r}\uplus_{g=1}^{d_e}\{\!\{\gamma \mid \gamma \in \Gamma_g^e\}\!\} = \uplus_{i=1}^{s}\uplus_{k=1}^{p_i}\{\!\{{}_iB_k\}\!\}$
   where $\{\!\{a\}\!\}$ is a multiset with one element a with multiplicity one, and $\uplus$ is multiset union.

Figure 3.2: Elimination rule schema for a connective $\mathbf{c}$

$$\dfrac{\begin{array}{c}[\varphi]\\ \vdots\\ \psi\end{array}}{\varphi \to \psi} \qquad\qquad \dfrac{\varphi \to \psi \quad \varphi \quad \begin{array}{c}[\psi]\\ \vdots\\ \chi\end{array}}{\chi}$$

Figure 3.3: Rules for implication

on Prawitz' work [1]. Like those proposed by Prawitz, the rule schemas by Haeusler and by Hao Chi attempt to generalize rules for natural deduction systems for structural propositional logics. In addition, the schemas in [48, §2][50, §3.1] do not contemplate provisos, thus restricting themselves to a subset of propositional logics. To express a wider range of logics, we enrich the rule schemas in Figures 3.1 and 3.2 with labels and an optional rule proviso (not shown in the figures for legibility reasons).

The labelling system is motivated by the wish to encompass the labelling systems of all the deductive systems proposed by Rentería in his thesis [5] — although we consider more systems than the ones he proposed. Every formula is thus annotated with a label (whose syntax is system dependent), and rules may deconstruct and construct labels just as they do to formulas. The rule proviso is a predicate on the formulas appearing in the rule and their labels; only if the predicate holds can the rule be applied.

The schemas we propose are shown in Figures 3.4 and 3.5. They are very similar to the ones in Figures 3.1 and 3.2, but their formulas are all labelled, and the labels are arguments to the rule predicate (not pictured for lack of space) and to the $\mathbf{h}_i, \mathbf{k}^e_t, {}_e\mathbf{f}^c_g$ functions that construct the labels of the conclusion for the introduction and elimination rules, respectively.

All that remains is to characterize the predicate and the label-building k functions. A solution would be to only require them to be computable, which would be non-restrictive but too general, potentially increasing the complexity of defining new systems. In defining his Labelled Deduction Systems (LDS) paradigm, Gabbay [2, p. 70] faced a similar conundrum:

> The reader may further have doubts about the use of labels from the computational point of view. What do we mean by a unifying framework [for labels]? Surely a Turing machine can simulate any logic, is that a unifying framework? The use of labels is powerful, as we know from computer science, are we using labels to play the role of a Turing machine? The answer to the question is twofold. First

$$\frac{\begin{array}{cccc}\left[{}_iA_1^1 : {}_il_1^1\right], \ldots, \left[{}_iA_{j_1^i}^1 : {}_il_{j_1^i}^1\right] & & \left[{}_iA_1^{p_i} : {}_il_1^{p_i}\right], \ldots, \left[{}_iA_{j_{p_i}^i}^{p_i} : {}_il_{j_{p_i}^i}^{p_i}\right] \\ \vdots & & \vdots \\ {}_iB_1 : {}_im_1 & \ldots & {}_iB_{p_i} : {}_im_{p_i}\end{array}}{\mathbf{c}((({}_iB_k)_{k=1}^{p_i})_{i=1}^{s}, (((({}_iA_m^l)_{m=1}^{j_l^i})_{l=1}^{p_i})_{i=1}^{s}) : \mathbf{h}_i((({}_im_k)_{k=1}^{p_i}, ((({}_il_m^l)_{m=1}^{j_l^i})_{l=1}^{p_i})}\,\mathbf{c}\mathrm{I}_\mathrm{i}$$

where:

1. **c** is the logical connective being introduced
2. $i = 1, \ldots, s$ (the connective **c** has s introduction rules)
3. $t = 1, \ldots, p_i$ (the i-th introduction rule has p_i premises of the form ${}_iB_t : {}_im_t$)
4. the t-th premise ${}_iB_t : {}_im_t$ of the i-th introduction rule may have up to j_t^i discharged hypotheses of the form ${}_iA_a^t : {}_il_a^t$
5. $\mathbf{h}_i$ is the constructor function for the conclusion's label

Figure 3.4: Introduction rule schema for a connective **c** with labelled formulae

> that we are not operating at the meta-level, but at the object-level.
> Second, there are severe restrictions on the way we use LDS.

Gabbay goes on to enumerate the restrictions established by LDS, but he does not propose a fixed set of restrictions on labels, simply requiring them to be an algebra that can be chosen by the system's proponent. For GLF, we go further: we require the labelling system to be an algebra on lists.

In practice, this means that labels must be lists (of what, the system's proponent may choose, but usually they are lists of variables), and that the label functions must be list functions such as `cons`, `uncons`, `intersection`, `union`, `difference`, `sublist` (and their compositions). These functions can all be defined in terms of the two list constructors (the one for the empty list, usually named `Nil`, and the other that constructs non-empty lists, usually named `Cons`) and recursion. We also allow primitive logical functions such as FV that return the list of free variables in a formula and logical predicates such as Atomic (which determines if a formula is atomic or not), plus their compositions. This way we avoid demanding general computable functions for predicate definition, as is the case in Haeusler's framework [49], and can still define all the systems described in Chapter 4, and more.

As an example to aid the understanding of our rule schemas, take the the universal quantifier introduction rule in a labelled natural deduction system for first-order logic. This system (presented by Rentería [5, §2]) uses the labels

$$\frac{\mathbf{c}(((_iB_k)_{k=1}^{p_i})_{i=1}^{s}, (((_iA_m^l)_{m=1}^{j_l^i})_{l=1}^{p_i})_{i=1}^{s}) : u_e \quad \begin{matrix} H_1^e : v_1^e \\ \vdots \\ H_{n_e}^e : v_{n_e}^e \end{matrix} \quad \begin{matrix} \chi : w \\ \vdots \\ \Gamma_1^e \end{matrix} \quad \dots \quad \begin{matrix} \chi : w \\ \vdots \\ \Gamma_{d_e}^e \end{matrix}}{\chi : w}\ \mathbf{c}E_{\mathrm{e}}$$

where:

1. $\mathbf{c}$ is the logical connective being introduced
2. $e = 1, \dots, r$ (the connective $\mathbf{c}$ has r elimination rules)
3. the e-th elimination rule has n_e minor premises and d_e discharging premises
4. Each label v_{t+1}^e of the minor premises is a label construction function $\mathbf{k}_{t+1}^e(u_e, (v_o^e)_{o=1}^t)$
5. Γ_g^e are sets of hypotheses that may be discharged by applications of the rule; each such hypothesis is of the form ${}_e\mathbf{f}_g^c(u_e, (v_o^e)_{o=1}^{n_e})$ with $c = 1 \dots |\Gamma_g^e|$
6. $\uplus_{e=1}^{r} \uplus_{f=1}^{n_e} \{\!\{H_f^e\}\!\} = \uplus_{i=1}^{s} \uplus_{l=1}^{p_i} \uplus_{m=1}^{j_l^i} \{\!\{{}_iA_m^l\}\!\}$ and $\uplus_{e=1}^{r} \uplus_{g=1}^{d_e} \{\!\{\gamma \mid \gamma \in \Gamma_g^e\}\!\} = \uplus_{i=1}^{s} \uplus_{k=1}^{p_i} \{\!\{{}_iB_k\}\!\}$ where $\{\!\{a\}\!\}$ is a multiset with one element a with multiplicity one, and $\uplus$ is multiset union.

Figure 3.5: Elimination rule schema for a connective $\mathbf{c}$ with labelled formulae

to keep track of the free variables introduced by the hypotheses of the proof, so every hypothesis must be labelled with a list of its free variables. Given this labelling convention, we are able to define the rule in Figure 3.6, with the predicate checking whether the variable bound by the introduced quantifier is not present in the premise's label.

The usual rule for the universal quantifier introduction would involve a non-local proviso, over the undischarged hypotheses of the tree. The labelling system allows us to create a local proviso (that is, one ranging over the sub-formulas appearing in the rule, either in the premises or the conclusion, or their labels).

The same first-order system provides an interesting example of label construction (e.g., the functions $\mathbf{h}_i, \mathbf{k}_t^e, {}_e\mathbf{f}_g^c$ in Figures 3.4 and 3.5). Because of the invariant requiring labels to be a list of the free variables appearing in undischarged hypotheses at any point of the proof, the rule for implication introduction (see Figure 3.6) builds a label for its conclusion by making the multi-set difference (represented by the $\setminus$ symbol) between the premise's

$$x \notin l \,\frac{\varphi : l}{\forall x \varphi : l} \qquad\qquad \frac{\begin{array}{c}[\varphi : m] \\ \vdots \\ \psi : l\end{array}}{\varphi \to \psi : l \setminus m}$$

Figure 3.6: Rule for universal quantifier introduction and for implication introduction in a labelled first-order logic system

$$\frac{\varphi \wedge \psi : u \qquad \begin{array}{c}[\varphi : u]\,[\psi : u] \\ \vdots \\ \chi : m\end{array}}{\chi : m}\,\wedge\mathrm{E} \qquad\qquad \frac{\exists x\, \varphi : l \qquad \begin{array}{c}[\varphi : l, x] \\ \vdots \\ \chi : m\end{array}}{\chi : m}\,\exists\mathrm{E}$$

Figure 3.7: Rules for conjunction (FOL) and existential (Keisler) elimination

label and the discharged hypothesis' label. Note that the definition of label construction functions are given outside the rule definitions, in Chapter 5 we show how rules and label functions are defined in GLF in practice (see Figure 5.7 in particular).

As examples of elimination rules, we take the conjunction elimination rule for the first-order logic system in Section 4.1 and the existential elimination rule for the Keisler logic system in Section 4.4, see Figure 3.7. The existential quantifier elimination is often shown in the same form as in this example, but the single conjunction elimination rule is different from the usual presentation. Not only does this presentation combine the two usual rules into one, but it also makes the elimination rule for conjunction look more similar to the disjunction elimination rule.

3.1 Rule equivalence

In comparing the usual forms of an inference rules and the schematic ones,[2] one may wonder whether they are equivalent. The equivalence is most difficult to see in the case of elimination rules, since introduction rules already fit the schema, without modifications.

We proceed to show intuitively the equivalence between the usual elim-

[2]For example, take the usual presentation of the conjunction elimination rule and the one in Figure 3.7.

$$\dfrac{\dfrac{\Pi}{\varphi \wedge \psi}}{\varphi} \qquad\qquad \dfrac{\dfrac{\Pi}{\varphi \wedge \psi} \quad [\varphi]}{\varphi}$$

Figure 3.8: Equivalent sub-deductions using the original rule for conjunction elimination and its schematic version

$$\dfrac{\dfrac{\Pi_1}{\varphi \to \psi} \quad \dfrac{\Pi_2}{\varphi}}{\psi} \qquad\qquad \dfrac{\dfrac{\Pi_1}{\varphi \to \psi} \quad \dfrac{\Pi_2}{\varphi} \quad [\psi]}{\psi}$$

Figure 3.9: Equivalent sub-deductions using the original rule for implication elimination and its schematic version

ination and their schematic versions. For each discharged hypothesis of the schematic rule we have a regular elimination rule, and instead of deriving a common result for each discharging premise, we simply derive the discharged hypothesis, which can then be used as a premise to other rules. In the case of the conjunction elimination rule in Figure 3.7, we get the two usual rules $\phi \wedge \psi \ / \ \phi$ and $\phi \wedge \psi \ / \ \psi$. Similarly, for the schematic implication elimination rule (Figure 3.3) we get the regular elimination rule, by taking the final result χ from the rule to be the discharged hypothesis instead: $\phi \to \psi, \phi \ / \ \psi$.

We can see this equivalence more clearly by noting we can derive the original rules from their schematic versions. For example, whenever we use the (original) conjunction elimination rule in a deduction, we can substitute it for its schematic version without jeopardizing the correctness of the deduction. In Figure 3.8, Π is a sub-deduction leading to the conclusion $\varphi \wedge \psi$; the rule for $\wedge$E is then applied. On the left side, the original rule is applied, while on the right side the schematic version is chosen; by choosing to derive the left conjunct, we end up with equivalent sub-deductions, meaning that any deduction containing the left sub-deduction could be changed to have the right sub-deduction instead, preserving the original deduction. Figure 3.9 shows the same transformation, but for the implication elimination rule.

3.2 Correctness & completeness of GLF

After wondering about the equivalence between the usual deductive rules and their schematic versions, it is natural to wonder if the implementation of a logical system in GLF is indeed equivalent to the original. To demonstrate the equivalence between the system and its implementation in GLF, we state and prove the following two theorems:

Theorem 1 (Correctness of GLF)**.** *If a system* $\mathfrak{S}$ *whose rules are schematic is correct with respect to logic* $\mathfrak{L}$*, then its GLF implementation is also correct with respect to* $\mathfrak{L}$*.*

If the implementation of $\mathfrak{S}$ in GLF is correct, we have that any deductions in this implementation are valid in $\mathfrak{L}$. To show this, we will demonstrate that deductions in the GLF implementation are equivalent to the ones in the original system $\mathfrak{S}$, and since $\mathfrak{S}$ is correct so is the GLF implementation.

Rule application in GLF Any $\mathfrak{S}$ rule must have the form of one of our rule schemas Figures 3.4 and 3.5, but there is a general procedure for rule application so that we do not need to distinguish them for this purpose. To apply a rule in GLF we need a rule description, an indication of which existing deductions are to be the rule's premises, and the formula to be concluded.

A rule description in GLF is equivalent to a description like the one in a textbook (or the ones in Figure 3.7, for example), except in the case of provisos or label constructors we may need to supply an implementation of the function that checks the proviso or the one that verifies the label, respectively.[3] For example, in Figure 3.6 we need to provide an implementation that can check whether $x \notin l$ holds, and one that can verify that the conclusion's label matches the expression $l \setminus m$.

For the rule's premises, one must indicate which deductions are to be which premises. Recall Definition 2; here we need only worry about the set of assumptions that a conclusion depends on, and the conclusion itself; we may ignore the rest of the formula-tree.

Given a rule description, the deductions-as-premises, and the rule's conclusion-to-be, we may attempt to apply the rule. First, the premises are checked against the rule description. For example, in the case of an elimination rule, the major premise is checked to have as its principal connective the required connective. When checking a premise against the rule descriptions, any named sub-formulas are bound: again in the case of the elimination

[3] More details about the rule description are in Chapter 5, specially Figure 5.7 and related figures.

rule schema, the major premise may have ${}_iB_k$ and ${}_iA^l_m$ as sub-formulas, and when it does, these are bound to their actual formulas and whenever they reappear in the rule descriptions (e.g., in the minor premises) their actual formulas must also match. We also check the rule's conclusion against the rule description, including its label. Again, any named sub-formulas are checked against previous references and stored for comparison with later references.

For example, take the conjunction elimination rule in Figure 3.7. Their rule descriptions stipulate what the main connective of the major premise is, and binds its two operands to the names (φ and ψ). If a derivation concluding $A \wedge B$ is taken to be the major premise in the case of the conjunction elimination rule, φ is bound to A and ψ to B. The discarding premises descriptions say that the formulas bound to φ and ψ must be discarded from the derivations taken as premises, and their conclusions must be bound to the name χ. In the case of a corresponding disjunction elimination rule, the description would stipulate that both discarding premises must conclude the same χ, a condition that is checked before the rule application can continue. If the description checks out, all that remains is to check the conclusion of the new derivation.[4] If an elimination rule follows the rule schema, the conclusion of the new derivation is easy to check, since it can be any labelled formula. For introduction rules like the ones in Figure 3.6, the conclusion must match the formula described, matching any sub-formulas appearing in the premises exactly, with the same reasoning applying for its labels. In the case of the universal quantifier introduction, the formulas bound to φ in the premise and in the conclusion must match, as must the labels bound to l. Finally, we take the union of the premises' updated assumptions (the updated assumptions already have the required assumptions discarded; e.g., in the conjunction elimination example A and B are removed from the only discarding premise), and we then produce a new deduction from these assumptions to the conclusion.

A pseudocode for rule application is shown in Algorithm 1. Note that when a CHECK call fails, rule application fails. CHECK always takes a map of names to formulas (Γ) as argument, and returns an updated version of it. It is in this map that named sub-formulas are inserted as of their first appearance, and it is from this map that references to named formulas are taken from to check that they match later appearances. Before rule application can proceed, we must also check any proviso the rule might have, which is done by CHECKPROVISO. If the proviso is not fulfilled, rule application fails. When the proviso (if any) is fulfilled, UPDATEDISCARDASSUMPTIONS returns

[4] Although checking the result when we can construct it from the premises is redundant, here we describe the general case where we can not always construct the conclusion but need it to be chosen and supplied by the user.

the new deduction's assumptions by discarding any assumptions from the premises that the rule description stipulates, and calculating the set union of the results (resulting in Υ). We are then finally able to return the new deduction concluding κ from the updated set of assumptions.

Algorithm 1 Pseudo-code for rule application

```
function applyRule(ℜ, 𝔇, κ)
    ▷ ℜ: rule description ◁
    ▷ 𝔇: deductions selected as the rule's premises ◁
    ▷ κ: conclusion formula ◁
    Γ ← {}
    for all δ ∈ 𝔇 do
        Γ ← Check(ℜ, δ, Γ)
    Γ ← Check(ℜ, κ, Γ)
    CheckProviso(ℜ, 𝔇, Γ)
    Υ ← updateDiscardAssumptions(ℜ, 𝔇, Γ)
    return Υ ⊢ κ                    ▷ New deduction
```

From this explanation we see that applying a $\mathfrak{S}$ rule in GLF is equivalent to applying the same rule outside of it (e.g., on a pen-and-paper proof). Given this equivalence, to complete the proof of correctness we need only show that GLF deductions are equivalent to regular $\mathfrak{S}$ deductions. We do so by induction on the deduction tree (recall Definition 2). The base cases are the deductions' assumptions (which are always correct, since anything may be assumed) and axioms. In the latter case we may take axioms to be rules without premises, so rule application proceeds as normal. There are no premises to check, but the conclusion's description is matched against the actual conclusion formula, and the application only succeeds if they match. Because rule application does not depend on the details of the rule schemas, the fact that axioms do not adhere to them (naturally, since they are not rules) is no problem.

Now on to the inductive case: we have a sequence of deductions we know to be correct that are to be premises to a rule application that will result in a new deduction. If the premises match the rule's description as previously described, the rule application is correct, and we obtain a new deduction that is correct. □

Theorem 2 (Completeness of GLF)**.** *If a system $\mathfrak{S}$ whose rules are schematic is strongly complete with respect to logic $\mathfrak{L}$, then its GLF implementation is also strongly complete with respect to $\mathfrak{L}$.*

To see that the implementation of $\mathfrak{S}$ in GLF is strongly complete with respect to $\mathfrak{L}$, it suffices to show that any $\mathfrak{L}$ formula φ that is logically entailed by a set of assumptions Γ has a GLF deduction $\Gamma \vdash \varphi$. Since we know $\mathfrak{S}$ itself is strongly complete, then any logical entailment of $\mathfrak{L}$ has a corresponding

deduction in $\mathfrak{S}$. Given such a deduction in $\mathfrak{S}$, we can turn it into a GLF deduction by repeating its steps. First, we make in GLF all the assumptions made in the $\mathfrak{S}$ deduction (from the set of assumptions Γ), and apply any axioms used in the original $\mathfrak{S}$ deduction. Starting from the assumptions and axioms (the deduction's formula-tree's leaves), we apply the same rules of the original deduction in GLF. Given the correctness of GLF (Theorem 1), we derive the same conclusions as the sub-deductions in the original deduction do, and so we finally obtain the same conclusion φ — showing that any logical entailment of $\mathfrak{L}$ has a corresponding GLF deduction. □

Rules that do not fit in the rule schemas — such as the infamous case of classical *reduction ad absurdum* — can be managed in an *ad hoc* way. Although not schematic, these rules can be defined in and applied by GLF with no problems, since rule application does not presuppose the rule schemas (see Algorithm 1 again). We show how to define axioms and rules — including non-schematic ones — in the implementation which is described in Chapter 5, and we discuss cases of non-schematic rules in the next chapter, where we present a series of labelled systems for various logics.

4
Systems

To show that our framework is robust, being able to express systems for several logics, we use it to implement systems for the following non-exhaustive list of logics: first-order logic, ultrafilter logic, filter logic, CTL, Kleisler logic, CTL*, iALC. Most of these logics are presented by Rentería [5], and were chosen to show a diversity of quantifiers that takes the labelling system to its limits.

In what follows we give an overview of these logics, explaining their syntax and semantics, and briefly discussing how their rules fit the rule schemas from Chapter 3. Except where otherwise noted, we follow Rentería in our presentation, with some changes for clarity and homogeneity, and already defining the labels using our framework.

$$\phi, \psi ::= \bot \mid P\ \tau^* \mid \phi \to \psi \mid \phi \wedge \psi \mid \forall v\ \phi$$
$$\tau ::= v \mid f\ \tau^*$$

Figure 4.1: First-order logic syntax

4.1 First-order logic

First-order logic (FOL) is central to modern logic; in fact, when we think of the word 'logic' without any qualifications, we are usually thinking of first-order logic. Ferreirós [52] provides an explanation of why this is so, focusing more on the historical contingencies that helped create this state of affairs than on the meta-theoretical properties that make FOL so interesting (with it "being the only quantificational system that is proof-theoretically well-behaved and sound", as Ferreirós puts it).

A good introduction to (mathematical) logic, including first-order logic, is the one by Enderton [53, §2]. Here we will simply show an overview of the syntax and semantics of a complete subset of FOL (e.g., we omit the $\exists$ quantifier and the $\vee$ connective, which can be nevertheless defined in terms of the included quantifier and connectives).

Syntax First-order logic syntax is shown in the formal grammar in Figure 4.1. Do note that τ is the symbol for terms, which are either variables v or function applications. Functions are denoted by the symbol f, and the number of terms following f must match its arity for the term to be well-formed. P is any predicate symbol (and for the formula to be well-formed the number of terms following it must match its arity).

Semantics One can divide FOL syntax into the logical (e.g., connectives) and non-logical symbols (e.g., functions). To 'give meaning' to a FOL formula we need to give meaning to its non-logical symbols, and assign a domain of discourse that establishes what the terms refer to and what quantifiers quantify over. A model $\mathcal{M}$ for FOL is a tuple $(D, \mathfrak{F}, \mathfrak{P})$ where D is a non-empty set (e.g., the set of natural numbers), $\mathfrak{F}$ is function that maps a n-ary function symbol f to a function $\mathfrak{F}(f) : D^n \to D$ (e.g., in one model $\mathfrak{F}(+)$ maps the function symbol '+' to arithmetic addition, while in another it maps to string concatenation), and $\mathfrak{P}$ is a function that maps a n-ary predicate symbol P to a subset of the n-ary product of the domain, i.e., $\mathfrak{P}(P) : D^n$ (e.g., in a model

$\mathcal{M}'$, $\mathfrak{P}(\leq)$ may represent the 'less than or equal' relation on natural numbers, while in another it may represent the sub-string relation among strings).

The final necessary step to be able to determine if a formula is true is to have a variable assignment μ that associates each variable with an element of the domain D. This variable assignment can be extended (we will call this extension $\bar{\mu}$) to handle arbitrary terms: $\bar{\mu}(f(t_1, \ldots, t_n))$ reduces to $\mathfrak{F}(f)(\bar{\mu}(t_1), \ldots, \bar{\mu}(t_n))$, with each $\bar{\mu}(t_i)$ (and thus $\bar{\mu}(f(t_1, \ldots, t_n))$ as a whole) evaluating to an element of D.

We now have everything we need to provide a semantics for first-order logic. A FOL formula ϕ is true in a model $\mathcal{M}$ under a variable assignment μ iff:

($\bot$) $\mathcal{M} \nvDash_\mu \bot$

(Atomic) $\mathcal{M} \vDash_\mu P(t_1, \ldots, t_n)$ iff $(\bar{\mu}(t_1), \ldots, \bar{\mu}(t_n)) \in \mathfrak{P}(P)$.

($\rightarrow$) $\mathcal{M} \vDash_\mu \phi_1 \rightarrow \phi_2$ iff $\mathcal{M} \nvDash_\mu \phi_1$ or $\mathcal{M} \vDash_\mu \phi_2$, or both.

($\wedge$) $\mathcal{M} \vDash_\mu \phi_1 \wedge \phi_2$ iff both $\mathcal{M} \vDash_\mu \phi_1$ and $\mathcal{M} \vDash_\mu \phi_2$.

($\forall$) $\mathcal{M} \vDash_\mu \forall x \phi$ iff $\mathcal{M} \vDash_{\mu_i'} \phi$ for every possible μ_i' such that μ_i' is same as μ except for the the value assigned to x, which may be any element of D.

Labelled Natural Deduction system This system (presented by Rentería [5, §2]) uses the labels to keep track of the free variables introduced by the hypotheses of the proof, so every hypothesis must be labelled with a list of its free variables.

The diagrams representing the rules for the labelled deduction system can be seen in Figure 4.2. The schematic versions that differ from the Rentería presentation of the same system are in Figure 4.3; the only non-schematic rule is the classical *reductio ad absurdum* $\bot$ rule, which neither introduces nor eliminates a connective. Note that any hypotheses must be labelled properly when they are introduced (i.e., their labels must include all and only their free variables), and that $\setminus$ and $\uplus$ are multiset difference and multiset union, respectively. Additionally, the following restriction applies to the $\forall E$ rule: $\varphi[x \leftarrowtail t]$ denotes the substitution of x for t in φ, which can be done iff t is free for x; if that is not the case, the rule is not applicable.

$$\frac{\begin{array}{c}[\neg\varphi : u]\\ \vdots\\ \bot : v\end{array}}{\varphi : v \setminus u}\,\bot \qquad \frac{\begin{array}{c}[\varphi : u]\\ \vdots\\ \psi : v\end{array}}{\varphi \to \psi : v \setminus u}\,{\to}\mathrm{I} \qquad \frac{\varphi \to \psi : v \quad \varphi : u}{\psi : u \uplus v}\,{\to}\mathrm{E}$$

$$\frac{\varphi : u \quad \psi : v}{\varphi \wedge \psi : u \uplus v}\,{\wedge}\mathrm{I} \qquad \frac{\varphi \wedge \psi : u}{\varphi : u}\,{\wedge}\mathrm{E}_1 \qquad \frac{\varphi \wedge \psi : u}{\psi : u}\,{\wedge}\mathrm{E}_2$$

$$x \notin u\,\frac{\varphi : u}{\forall x\varphi : u}\,\forall\mathrm{I} \qquad \frac{\forall x\varphi : u}{\varphi[x \leftarrowtail t] : u}\,\forall\mathrm{E}$$

Figure 4.2: Natural deduction rules for First-order logic

$$\frac{\varphi \to \psi : v \quad \varphi : u \qquad \begin{array}{c}[\psi : u \uplus v]\\ \vdots\\ \chi : m\end{array}}{\chi : m}\,{\to}\mathrm{E} \qquad \frac{\varphi \wedge \psi : u \qquad \begin{array}{c}[\varphi : u]\,[\psi : u]\\ \vdots\\ \chi : m\end{array}}{\chi : m}\,{\wedge}\mathrm{E}$$

$$\frac{\forall x\varphi : u \qquad \begin{array}{c}[\varphi[x \leftarrowtail t] : u]\\ \vdots\\ \chi : m\end{array}}{\chi : m}\,\forall\mathrm{E}$$

Figure 4.3: Schematic versions of non-schematic rules for FOL system

$$\phi ::= \bot \mid P\ \tau^* \mid \phi \to \phi \mid \phi \wedge \phi \mid \forall v\, \phi \mid \nabla v\, \phi$$
$$\tau ::= v \mid f\ \tau^*$$

Figure 4.4: Ultrafilter logic syntax

4.2 Ultrafilter logic

Ultrafilter logic is a logic for generic reasoning proposed by Veloso [54], which extends first-order logic with a new quantifier, the quasi-universal, represented by the ∇ symbol. The quasi-universal's intuitive meaning is of 'for almost all', as in the natural language sentence 'almost all birds can fly'. The presentation of ultrafilter logic given here follows Rentería [5, §3], who introduced the labelled Natural Deduction system for this logic.

Syntax Ultrafilter logic syntax is shown in the formal grammar in Figure 4.4 (it is the same as for first-order logic Figure 4.1, but with the addition of a case for $\nabla x\, \phi$). Do note that τ denotes terms, which are either variables (denoted by v) or function applications (functions are denoted by f, and a valid function application must have f following by a number of terms matching its arity).

Semantics The semantics of ultrafilter logic is the same as for first-order logic (see Section 4.1), except we need the concept of an ultrafilter $\mathcal{F}$ in the definition, and add a case for formulas of the form $\nabla x \varphi$.

Following the definition of filter and ultrafilter given by Goldblatt [55]:

Definition 3. *A filter $\mathcal{F}$ on a nonempty set I is a nonempty collection of subsets of I satisfying:*

- *if $A, B \in \mathcal{F}$, then $A \cap B \in \mathcal{F}$.*
- *if $A \in \mathcal{F}$ and $A \subseteq B \subseteq I$ then $B \in \mathcal{F}$.*

A *proper* filter does not contain the empty subset, or else it is equal to the power-set of the underlying set.

Definition 4. *An ultrafilter is a proper filter that satisfies:*

- *for any $A \subseteq I$, either $A \in \mathcal{F}$ or $A^C \in \mathcal{F}$, where $A^C = I - A$.*

Given the definition of ultrafilter, a model for first-order logic $\mathcal{M}$, and an ultrafilter $\mathcal{F}$, we may give the following semantics for a formula $\nabla x\, \varphi$:

(∇) $(\mathcal{M}, \mathcal{F}) \vDash \nabla x \varphi$ holds iff the extension of φ in $(\mathfrak{U}, \mathcal{F})$ is in $\mathcal{F}$, with $\mathcal{F}$ being the ultrafilter defined over the domain D of the model $\mathcal{M}$.

Labelled Natural Deduction system In this system, formulas are labelled with lists of (optionally) marked variables.[1] An unmarked variable is written x, while a marked one is written $\bar{x}$. These labels must follow two invariants: any variable must occur at most once, and any variable in the label should occur free in the associated formula.

This labelling system allows us to 'register' the order in which the quantifiers were eliminated, and the variable marks allow us to know when the variable occurring free in the formula originates from the quasi-universal quantifier (∇) or the universal quantifier elimination, so that we may reintroduce them only when allowed.

Figure 4.5 shows the rules for the labelled Ultrafilter Natural Deduction system, while Figure 4.6 shows the schematic versions of the non-schematic elimination rules in the original system. Do note that we take l, m, n to range over labels, while x, y range over label elements (i.e., marked or unmarked variables).

Special attention should be given to rules $\forall E_3$ and $\forall E_4$. These two rules are the same, except for the mark in the new variable entering the conclusion's label. Compare these two rules to the the ∇E_1 rule: it is a single rule, and the variable introduced in the conclusion's label is marked, with no unmarked counterpart. The reason for this discrepancy is clear from the introduction rules for the quasi-universal and universal quantifiers: we may introduce a universal quantifier when we have a corresponding unmarked variable in the premises' label, and we may introduce the quasi-universal from a marked one. Since the semantics of ultrafilter logic guarantee that quasi-universality is entailed from universality but not the contrary, we must have those two universal elimination rules for the universal quantifier with marked and unmarked variable variants (meaning we can derive "almost all spiders have eight legs" from "all spiders have eight legs"), but the single elimination rule for the quasi-universal quantifier prevents us from deriving universality from quasi-universality (meaning we can't derive "all birds can fly" from "almost all birds can fly").

These (quasi-)universal elimination rules also illustrate the kind of changes we have to make to elimination rules in the original system rules to adapt them to our schema. To reach their form in Figure 4.6 they are modified in the same way as the conjunction elimination was to modified to fit our rule schemas, as explained in Chapter 3.

There are two rules that are not schematic. The $\bot$ rule (essentially the same rule as in the FOL system from Section 4.1), and the X rule. The X rule

[1] Syntactically, we can think of variable marks as unary operators.

$$\frac{\varphi : l, x}{\forall x\, \varphi : l} \forall \mathrm{I}_1 \qquad x \notin \mathrm{FV}(\varphi) \frac{\varphi : l}{\forall x\, \varphi : l} \forall \mathrm{I}_2$$

$$\frac{\forall x\, \varphi : l}{\varphi : l} \forall \mathrm{E}_1 \qquad \frac{\forall x\, \varphi : l}{\varphi[x \leftarrow\!\shortmid c] : l} \forall \mathrm{E}_2$$

$$x \in \mathrm{FV}(\varphi) \frac{\forall x\, \varphi : l}{\varphi(y) : l, y} \forall \mathrm{E}_3 \qquad x \in \mathrm{FV}(\varphi) \frac{\forall x\, \varphi : l}{\varphi : l, \bar{y}} \forall \mathrm{E}_4$$

$$\frac{\varphi : l, \bar{x}}{\nabla x\, \varphi : l} \nabla \mathrm{I}_1 \qquad x \notin \mathrm{FV}(\varphi) \frac{\varphi : l}{\nabla x\, \varphi : l} \nabla \mathrm{I}_2$$

$$\frac{\nabla x\, \varphi : l}{\varphi(y) : l, \bar{y}} \nabla \mathrm{E}_1 \qquad x \notin \mathrm{FV}(\varphi) \frac{\nabla x\, \varphi : l}{\varphi : l} \nabla \mathrm{E}_2$$

$$\frac{\varphi : l \quad \psi : m}{\varphi \wedge \psi : l \sqcup m} \wedge \mathrm{I} \qquad \frac{\varphi \wedge \psi : l}{\varphi : l \cap \mathrm{FV}(\varphi)} \wedge \mathrm{E}_1 \qquad \frac{\varphi \wedge \psi : l}{\psi : l \cap \mathrm{FV}(\psi)} \wedge \mathrm{E}_2$$

$$\frac{\begin{array}{c}[\varphi : l] \\ \vdots \\ \psi : m\end{array}}{\varphi \to \psi : l \sqcup u} \to \mathrm{I} \qquad \frac{\varphi \to \psi : l \quad \varphi : l \cap \mathrm{FV}(\varphi)}{B : l \cap \mathrm{FV}(\psi)} \to \mathrm{E}$$

$$\frac{\begin{array}{c}[\neg\varphi : l] \\ \vdots \\ \bot\end{array}}{\varphi : l} \bot \qquad \frac{\varphi : l, m, x, n}{\varphi : l, x, m, n} \mathrm{X}$$

Figure 4.5: Natural Deduction rules for Ultrafilter labelled deductive system

$$\frac{\begin{array}{c}[\varphi : l]\\ \vdots\\ \forall x\,\varphi : l \qquad \chi : m\end{array}}{\chi : m}\,\forall\mathrm{E}_1 \qquad \frac{\begin{array}{c}[\varphi[x \leftarrow\!\!\shortmid\, c] : l]\\ \vdots\\ \forall x\,\varphi : l \qquad \chi : m\end{array}}{\chi : m}\,\forall\mathrm{E}_2$$

$$x \in \mathrm{FV}(\varphi)\,\frac{\begin{array}{c}[\varphi(y) : l, y]\\ \vdots\\ \forall x\,\varphi : l \qquad \chi : m\end{array}}{\chi : m}\,\forall\mathrm{E}_3 \qquad x \in \mathrm{FV}(\varphi)\,\frac{\begin{array}{c}[\varphi : l, \bar{y}]\\ \vdots\\ \forall x\,\varphi : l \qquad \chi : m\end{array}}{\chi : m}\,\forall\mathrm{E}_4$$

$$\frac{\begin{array}{c}[\varphi(y) : l, \bar{y}]\\ \vdots\\ \nabla x\,\varphi : l \qquad \chi : m\end{array}}{\chi : m}\,\nabla\mathrm{E}_1 \qquad x \notin \mathrm{FV}(\varphi)\,\frac{\begin{array}{c}[\varphi : l]\\ \vdots\\ \nabla x\,\varphi : l \quad \chi : m\end{array}}{\chi : m}\,\nabla\mathrm{E}_2$$

$$\frac{\begin{array}{c}[\varphi : l \cap \mathrm{FV}(\varphi)]\,[\psi : l \cap \mathrm{FV}(\psi)]\\ \vdots\\ \varphi \wedge \psi : l \qquad \chi : m\end{array}}{\chi : m}\,\wedge\mathrm{E} \qquad \frac{\begin{array}{c}[\psi : l \cap \mathrm{FV}(\psi)]\\ \vdots\\ \varphi \to \psi : l \quad \varphi : l \cap \mathrm{FV}(\varphi) \qquad \chi : m\end{array}}{\chi : m}\,\to\mathrm{E}$$

Figure 4.6: Schematic versions of non-schematic rules for Ultrafilter system

is a structural rule, not an actual logic rule, so it not fitting our rule schemas is expected.

A few additional clarifications to Figure 4.5 are in order. $\mathrm{FV}(\varphi)$ denotes the free variables of formula φ, while $\varphi[x \leftarrow\!\!\shortmid\, c]$ denotes the substitution of variable x for term c in φ. The $\cap$ operator is list intersection (similar to multi-set intersection, but keeping the elements' order), while the $\overline{\cup}$ operator is similar to list union, but rearranging the order of the labels' elements is allowed as long as the resulting order respects the original ones (that is, in $l \,\overline{\cup}\, m$ if an element comes before another in l, it must still come before it in $l \,\overline{\cup}\, m$).

We must also state that rule $\forall\mathrm{I}_1$ demands that the variable x do not occur free in any undischarged hypotheses on which $\varphi : l, x$ depends. This proviso could be turned into a local condition on the labels if we were to make them more complex, but we present the system as given by Rentería.

4.3 Filter logic

Filter logic is similar to ultrafilter logic (see Section 4.2), but it uses filters (see Definition 3) instead of ultrafilters to give the semantics of a new quantifier (also represented by the ∇ symbol). The intuitive meaning of this new quantifier in natural language is that of 'for most' or 'generally', and so we no longer call it the quasi-universal in a Filter logic setting. Filter logic was proposed by Veloso & Veloso [56], while the labelled natural deduction system presented here is based on the one by Rentería [5, §4].

Syntax The syntax for Filter logic is the same as the one for ultrafilter logic (see Figure 4.4).

Semantics The semantics for Filter logic and Ultrafilter logic are also the same, except for the formulas whose principal connective is the ∇ quantifier:

(∇) $(\mathfrak{U}, \mathcal{F}) \vDash \nabla x\varphi$ holds iff the extension of φ in $(\mathfrak{U}, \mathcal{F})$ is in $\mathcal{F}$, with $\mathcal{F}$ being a filter defined over the universe of structure $\mathfrak{U}$.

Labelled Natural Deduction system Filter logic uses the same kind of labels as ultrafilter logic does, and its rules are the same as those of ultrafilter logic, except we need an additional proviso to the $\perp$ and the implication introduction rules (please refer to Figure 4.7): the rules are only applicable if the hypotheses' label does not contain any marked variables. This proviso is needed because marked variables are associated with ∇ quantification, and due to its different semantics in Filter logic, we do have that $\neg\nabla x\varphi$ implies $\nabla x\neg\varphi$ anymore. We can see intuitively why this is so: if we have that it is false that almost all individuals φ, then we must have that for almost all individuals $\neg\varphi$; but if we have that it is false that generally individuals are φ, we don't necessarily have that generally $\neg\varphi$.

The schematic versions of the elimination rules for the Filter logic system would look the same as the ones of the Ultrafilter logic system, so we omit them. Likewise, we can make analogous comments about the non-schematic nature of the $\perp$ and X rules.

$$\frac{\varphi : l, x}{\forall x\, \varphi : l}\,\forall \mathrm{I}_1 \qquad x \notin \mathrm{FV}(\varphi)\,\frac{\varphi : l}{\forall x\, \varphi : l}\,\forall \mathrm{I}_2$$

$$\frac{\forall x\, \varphi : l}{\varphi : l}\,\forall \mathrm{E}_1 \qquad \frac{\forall x\, \varphi : l}{\varphi[x \leftarrow\!\shortmid c] : l}\,\forall \mathrm{E}_2$$

$$x \in \mathrm{FV}(\varphi)\,\frac{\forall x\, \varphi : l}{\varphi(y) : l, y}\,\forall \mathrm{E}_3 \qquad x \in \mathrm{FV}(\varphi)\,\frac{\forall x\, \varphi : l}{\varphi : l, \bar{y}}\,\forall \mathrm{E}_4$$

$$\frac{\varphi : l, \bar{x}}{\nabla x\, \varphi : l}\,\nabla \mathrm{I}_1 \qquad x \notin \mathrm{FV}(\varphi)\,\frac{\varphi : l}{\nabla x\, \varphi : l}\,\nabla \mathrm{I}_2$$

$$\frac{\nabla x\, \varphi : l}{\varphi(y) : l, \bar{y}}\,\nabla \mathrm{E}_1 \qquad x \notin \mathrm{FV}(\varphi)\,\frac{\nabla x\, \varphi : l}{\varphi : l}\,\nabla \mathrm{E}_2$$

$$\frac{\varphi : l \quad \psi : m}{\varphi \wedge \psi : l \sqcup m}\,\wedge \mathrm{I} \qquad \frac{\varphi \wedge \psi : l}{\varphi : l \cap \mathrm{FV}(\varphi)}\,\wedge \mathrm{E}_1 \qquad \frac{\varphi \wedge \psi : l}{\psi : l \cap \mathrm{FV}(\psi)}\,\wedge \mathrm{E}_2$$

$$\not\exists \bar{x} \in l\,\frac{\begin{array}{c}[\varphi : l] \\ \vdots \\ \psi : m\end{array}}{\varphi \to \psi : l \sqcup u}\,\to \mathrm{I} \qquad \frac{\varphi \to \psi : l \quad \varphi : l \cap \mathrm{FV}(\varphi)}{B : l \cap \mathrm{FV}(\psi)}\,\to \mathrm{E}$$

$$\not\exists \bar{x} \in l\,\frac{\begin{array}{c}[\neg\varphi : l] \\ \vdots \\ \bot\end{array}}{\varphi : l}\,\bot \qquad \frac{\varphi : l, m, x, n}{\varphi : l, x, m, n}\,\mathrm{X}$$

Figure 4.7: A labelled deductive system for Filter logic

$$\phi, \psi ::= \bot \mid P\ \tau^* \mid \phi \to \psi \mid \phi \vee \psi \mid \exists y\ \phi \mid \mathcal{Q} y\ \phi \mid y = x$$
$$\tau ::= v \mid f\ \tau^*$$

Figure 4.8: Keisler logic syntax

4.4 Keisler logic

Keisler logic extends first-order logic with a quantifier $\mathcal{Q}$, where $\mathcal{Q}\, x\, \phi(x)$ means that there are uncountably many x such that $\phi(x)$ holds. Keisler logic was first proposed by Mostowski [57], with Keisler [58] providing a completeness proof for a simple axiomatization. The presentation given here is again based on Rentería [5, §6], who originally introduced the labelled Natural Deduction system for Keisler logic for a fragment with only the existential and the $\mathcal{Q}$ quantifiers.

Syntax Save for the $\mathcal{Q}$ quantifier, the syntax of Keisler logic follows that of first-order logic, and it is shown in the formal grammar in Figure 4.8. Compared to the FOL system in Section 4.1, we follow Rentería in substituting the disjunction connective for the conjunction, and the existential quantifier for the universal. Do note that x and y denote variables.

Semantics The semantics of Keisler logic is the same as for first-order logic (see Section 4.1), except we add cases for formulas of the forms $\phi \vee \phi$, $\exists y\ \phi$, $\mathcal{Q} x\, \varphi$, and $y = x$, and remove those for $\phi \wedge \psi$ and $\forall v\ \phi$.

Given a model $\mathcal{M}$ for first-order logic and a variable assignment μ that associates each variable of a formula with an element of the domain of the model, we may give the following semantics for a formula of the form ϕ in Keisler logic:

($\bot$) $\mathcal{M} \nvDash_\mu \bot$

(Atomic) $\mathcal{M} \vDash_\mu P(t_1, \ldots, t_n)$ iff $(\bar{\mu}(t_1), \ldots, \bar{\mu}(t_n)) \in \mathfrak{P}(P)$.

($\to$) $\mathcal{M} \vDash_\mu \phi_1 \to \phi_2$ iff $\mathcal{M} \nvDash_\mu \phi_1$ or $\mathcal{M} \vDash_\mu \phi_2$, or both.

($\vee$) $\mathcal{M} \vDash_\mu \phi_1 \vee \phi_2$ iff either $\mathcal{M} \vDash_\mu \phi_1$ or $\mathcal{M} \vDash_\mu \phi_2$, or both.

($\exists$) $\mathcal{M} \vDash_\mu \exists x \phi$ iff $\mathcal{M} \vDash_{\mu_i'} \phi$ for a μ_i' that is same as μ except for the the value assigned to x, which may be any element of D.

($\mathcal{Q}$) $\mathcal{M} \vDash_\mu \mathcal{Q}\, x\, \phi$ iff $\mathcal{M} \vDash_{\mu'_i} \phi$ for an uncountable number of different μ'_i such that μ'_i is same as μ except for the the value assigned to x, which may be any element of the model's domain.

($=$) $\mathcal{M} \vDash_\mu y = x$ iff $\mu(y)$ is the same as $\mu(x)$.

Labelled Natural Deduction system Similarly to the Ultrafilter logic system, in this system formulas are labelled with lists of (optionally) marked variables, however there are no invariants over the labels as there are for the Ultrafilter system. An unmarked variable is written x, while a marked variable is written x^*. Note that whenever the label is an empty list, we omit it.

Informally we may think of unmarked variables as having an existential extension, while marked variables have an uncountable extension. This intuition is made formal by the rules for the existential and the uncountably-many quantifiers: we introduce the existential from an unmarked variable in the label, and when we eliminate it we introduce an unmarked variable in the label. Conversely, we introduce the uncountably-many quantifier from a marked variable in the label, and when we eliminate it we introduce a marked variable in the label.

Figure 4.10 shows the rules for the labelled Natural Deduction system for Keisler logic, while Figure 4.9 shows the schematic versions of the elimination rules. The Ax. and $\aleph$ rules both derive axioms in the original (axiomatic) deductive system for Keisler logic, hence why they do not fit our rule schemas. The $\aleph$ rule derives from the axiom $\mathcal{Q}\, y \exists x \varphi \to \exists x\, \mathcal{Q}\, y \varphi \vee \mathcal{Q}\, x \exists y \varphi$, and in the rule the first premise matches the antecedent of the axiom, while the conclusion matches the first disjunct. The other rules that do not fit the schemas are the classical RAA rule (for the same reasons as the $\bot$ rules from the FOL, Ultrafilter, and Filter systems), and the $\star$ rule which is structural.

The $\to$E rule has a special proviso: the variables in the label l must not occur free in the hypotheses of the sub-deduction Π.

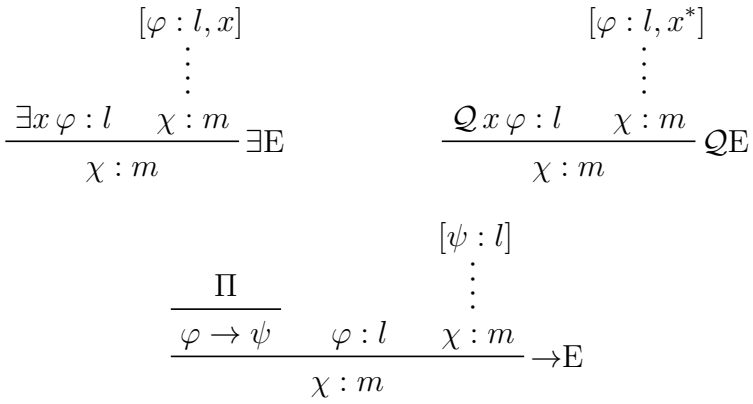

Figure 4.9: Schematic versions of non-schematic rules for Keisler system

$$\frac{}{(\mathcal{Q}x\,(x = y \vee x = z)) \to \bot}\,\text{Ax.}$$

$$\frac{\varphi : l, x}{\exists x\,\varphi : l}\,\exists\text{I} \qquad \frac{\exists x\,\varphi : l}{\varphi : l, x}\,\exists\text{E}$$

$$\frac{\varphi : l, x^*}{\mathcal{Q}\,y\,\varphi(y) : l}\,\mathcal{Q}\text{I} \qquad \frac{\mathcal{Q}\,x\,\varphi : l}{\varphi : l, x^*}\,\mathcal{Q}\text{E}$$

$$\not\exists x.x^* \in l\,\frac{\begin{array}{c}\varphi : l \\ \vdots \\ \psi : l\end{array}}{\varphi \to \psi : l}\,{\to}\text{I} \qquad \frac{\begin{array}{cc}\Pi & \\ \hline \varphi \to \psi & \varphi : l\end{array}}{\psi : l}\,{\to}\text{E}$$

$$\frac{\varphi : l}{\varphi \vee \psi : l}\,{\vee}\text{I} \qquad \frac{\varphi \vee \psi : l \quad \begin{array}{c}\varphi : l \\ \vdots \\ \chi : m\end{array} \quad \begin{array}{c}\psi : l \\ \vdots \\ \chi : m\end{array}}{\chi : m}\,{\vee}\text{E}$$

$$k \subseteq l, \not\exists x.x^* \in l\,\frac{\begin{array}{c}[\varphi \to \bot : l] \\ \vdots \\ \bot\end{array}}{\varphi : k}\,\text{RAA}$$

$$\frac{\varphi : l, y^*, x \quad \begin{array}{c}[\mathcal{Q}\,x \exists y \varphi] \\ \vdots \\ \bot\end{array}}{\varphi : l, x, y^*}\,\aleph \qquad \frac{\varphi : l \quad \begin{array}{c}[\varphi] \\ \vdots \\ \psi\end{array}}{\psi : l}\,\star$$

Figure 4.10: Natural Deduction rules for Keisler labelled deductive system

$$\phi, \psi ::= \bot \mid \mathbf{p} \mid \phi \to \psi \mid [\forall \mathcal{X}]\, \phi \mid [\exists \mathcal{G}]\, \phi \mid \exists [\phi \sim \psi]$$

Figure 4.11: CTL syntax

4.5 CTL

CTL (for *Computation Tree Logic*) is a logic meant to reason about temporal conditions introduced by Clarke and Emerson [59]. It is mostly used for formal verification of safety-critical software, being able to guarantee that it has the properties it should have, for example that train traffic control software prevents collisions and allows trains to pass.[2]

The presentation given here is based on Rentería [5, §5], who originally introduced the labelled Natural Deduction system for CTL, and which we modify to fit our rule schemas.

Although CTL and CTL* (to be presented in Section 4.6) are propositional logics, many of their connectives are quantifier-like, as we will see in their semantics.

Syntax CTL syntax is shown in the formal grammar in Figure 4.11, where $\mathbf{p}$ is any atomic formula.

Semantics The semantics of CTL can be given as follows: a model $\mathcal{M}$ is a triple $(S, \Rightarrow, L)$, where S is a set of states, $\Rightarrow$ is a relation between states ($\Rightarrow \subseteq S \times S$) determining when one state is succeeded from another, and L is a function from S to the power-set of the atomic formulas of CTL, determining which atomic formulas are true in which states. We require that $\forall s \in S. \exists r \in S. s \Rightarrow r$, that is, that every state have at least one successor state. A CTL formula ϕ is entailed from a model $\mathcal{M}$ in a state $s \in S$ by recursion over ϕ:

$\bot$**.** $(\mathcal{M}, s) \nvDash \bot$

Atomic. When p is an atomic formula of CTL, $(\mathcal{M}, s) \vDash p$ iff $p \in L(s)$

$\to$**.** $(\mathcal{M}, s) \vDash \phi_1 \to \phi_2$ iff $(\mathcal{M}, s) \nvDash \phi_1$ or $(\mathcal{M}, s) \vDash \phi_2$

$[\forall \mathcal{X}]$**.** $(\mathcal{M}, s) \vDash [\forall \mathcal{X}]\, \phi$ iff for every r such that $(s, r) \in \Rightarrow$ we have that $(\mathcal{M}, r) \vDash \phi$

[2] We need both properties since it's easy to obtain zero collisions by keeping all trains stopped. In the general case we call these safety and liveness properties.

$[\exists\mathcal{G}]$. $(\mathcal{M}, s) \vDash [\exists\mathcal{G}]\,\phi$ iff there is an infinite sequence of states $(s_k)_{k=0}^{\infty}$ with $s_0 = s$ and $(s_k, s_{k+1}) \in \Rightarrow$ such such that $(\mathcal{M}, s_k) \vDash \phi$.

$\exists[\sim]$. $(\mathcal{M}, s) \vDash \exists[\phi \sim \psi]$ iff there is a finite sequence of states $(s_k)_{k=0}^{j}$ with $s_0 = s$ and $(s_k, s_{k+1}) \in \Rightarrow$ such that $(\mathcal{M}, s_j) \vDash \phi$ and $(\mathcal{M}, s_i) \vDash \phi_1$ for every $0 \leq i < j$.

Labelled Natural Deduction system The labels used in our deductive system CTL are non-empty lists of symbols. Informally, the labels represent a sequence of states: if l is a label, the label $l + a$ represents a successor state of l. Similarly, $l + b$ is a successor state of l too, and it might or might not be the same successor state. Intuitively, we may say a labelled formula $\varphi : l$ means that $(\mathcal{M}, l) \vDash \varphi$.

The original rules for the CTL system are in Figure 4.13, while the rules that may be changed to be turned schematic are shown in Figure 4.14.

Rule $\exists+$ has a special proviso: the label elements a and b must not appear in neither k nor l, nor in the labels of any other open hypotheses of the discharging premises.[3] Rules $\mathcal{G}=$, $\mathcal{G}$Ind and $\exists$E also have special provisos that are discussed in Section 4.5.1, because they need the notion of sub-derivation which is given there.

As for the non-schematic rules, there is the classical *reductio ad absurdum* rule $\bot$E rule. The $\mathcal{G}$Ind rule can be seen as both an introduction or an elimination rule, but does not fit either schema, as is the case of $\mathcal{G}=$.

4.5.1 Sub-derivation

CTL rules $\mathcal{G}=$, $\mathcal{G}$Ind and $\exists$E involve sub-derivations (see Figure 4.13). The concept of sub-derivation used in this system is not the usual one, but the one as defined in Rentería [5, §5.4.4]. While the usual definition of sub-derivation only takes a complete sub-deduction above a certain rule result as a sub-derivation, sub-derivations as defined in Rentería may be any fragment of a deduction as long as no rule proviso is violated. Figure 4.12 shows a deduction and two of its valid sub-derivations; the latter one is invalid according to the usual definition of sub-derivation, but is valid according to the definition we use here.

These three rules involving sub-derivations have special provisos relating to which hypotheses they might contain. For instance in the case of the $\mathcal{G}=$,

[3] This proviso is similar to that of the $\mathcal{G}+$ and $\forall$I rules, and it is not shown along with the rule as in those two cases for lack of horizontal space in the page.

$$\dfrac{\dfrac{\dfrac{A \wedge C}{A \quad B}}{A \wedge B}}{B \to A \wedge B} \qquad \dfrac{\dfrac{A \wedge C}{A \quad B}}{A \wedge B} \qquad \dfrac{A \quad B}{A \wedge B}$$

Figure 4.12: A deduction and two valid sub-derivations of it

there must exist a sub-derivation of D deriving $\psi : l$ whose undischarged hypotheses are all of the form $\varphi : l$. For the $\mathcal{G}$Ind rule, we have that there must exist a sub-derivation of D deriving $[\forall \mathcal{X}](\varphi \to \bot) \to \bot : l$ whose undischarged hypotheses are all be of the form $\varphi : l$. Finally, in the case of the $\exists$E rule, we require that there be a sub-derivation of D_1 deriving $\chi : l$ whose undischarged hypotheses are all in the set $\{\varphi : l, \chi : l + a\}$; and a sub-derivation of D_2 deriving $\chi : l + a$ whose undischarged hypotheses are all of the form $\chi : l + a$. Additionally, if the sub-derivations of D_1 and D_2 are D_1 and D_2 themselves, then we may discharge their hypotheses.

Rules including provisos on sub-derivations do not fit our rule schemas, but Rentería says these provisos are not necessary for obtaining the soundness and the completude of the logical system, and so we do not find this to be too grave a problem. The GLF implementation of CTL does not enforce the sub-derivation provisos.

$$a \notin \bigcup_{i=1}^{l} l_i \; \frac{\begin{array}{c}\varphi_1 : l_1, \ldots, \varphi_n : l_n \\ \vdots \\ \varphi : l + a\end{array}}{[\forall \mathcal{X}]\, \varphi : l}\, \forall \mathrm{I} \qquad \frac{[\forall \mathcal{X}]\, \varphi : l}{\varphi : l + a}\, \forall \mathrm{E}$$

$$\frac{[\exists \mathcal{G}]\, \varphi : l}{\varphi : l}\, \mathcal{G}\mathrm{E} \qquad \frac{[\exists \mathcal{G}]\, \varphi : l + a \qquad \varphi : l}{[\exists \mathcal{G}]\, \varphi : l}\, \mathcal{G}-$$

$$a \notin k \cup \bigcup_{i=1}^{l} l_i \; \frac{[\exists \mathcal{G}]\, \varphi : l \qquad \begin{array}{c}\varphi_1 : l_1, \ldots, \varphi_n : l_n, [[\exists \mathcal{G}]\, \varphi : l + a] \\ \vdots \\ \chi : k\end{array}}{\chi : k}\, \mathcal{G}+$$

$$\frac{\begin{array}{c}D \\ [\forall \mathcal{X}](\varphi \to \bot) \to \bot : l\end{array}}{[\exists \mathcal{G}]\, \varphi : l}\, \mathcal{G}\mathrm{Ind} \qquad \frac{\begin{array}{c}D \\ [\exists \mathcal{G}]\, \varphi : l \qquad \psi : l\end{array}}{[\exists \mathcal{G}]\, \psi : l}\, \mathcal{G}=$$

$$\frac{\varphi : l \qquad \psi : l + a}{\exists [\varphi \sim \psi] : l}\, \exists \mathrm{I} \qquad \frac{\exists [\varphi \sim \psi] : l \qquad \begin{array}{c}D_1 \\ \chi : l\end{array} \qquad \begin{array}{c}D_2 \\ \chi : l + a\end{array}}{\chi : l}\, \exists \mathrm{E}$$

$$\frac{\exists (\varphi \sim \psi) : l \qquad \begin{array}{c}[\varphi : l]\, [\psi : l + a] \\ \vdots \\ \chi : k\end{array} \qquad \begin{array}{c}[\varphi : l]\, [\exists (\varphi \sim \psi) : l + b] \\ \vdots \\ \chi : k\end{array}}{\chi : k}\, \exists +$$

$$\frac{\exists [\varphi \sim \psi] : l + a \qquad \varphi : l}{\exists [\varphi \sim \psi] : l}\, \exists - \qquad \frac{\begin{array}{c}[\varphi \to \bot : k] \\ \vdots \\ \bot : l\end{array}}{\varphi : k}\, \bot \mathrm{E}$$

$$\frac{\begin{array}{c}[\varphi : l] \\ \vdots \\ \psi : l\end{array}}{\varphi \to \psi : l}\, \to \mathrm{I} \qquad \frac{\varphi \to \psi : l \quad \varphi : l}{\psi : l}\, \to \mathrm{E}$$

Figure 4.13: Natural Deduction rules for CTL labelled deductive system

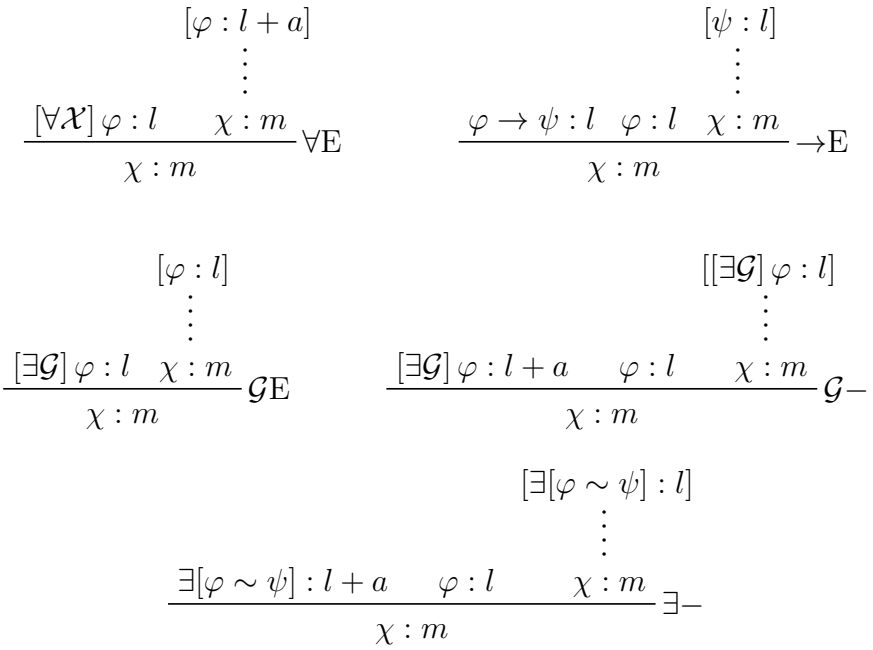

Figure 4.14: Schematic versions of non-schematic rules for CTL system

$$\phi, \psi ::= \bot \mid \mathbf{p} \mid \phi \to \psi \mid \mathcal{X}\phi \mid \phi \,\mathcal{U}\, \psi \mid \mathcal{G}\phi \mid \mathcal{E}\theta$$

Figure 4.15: CTL* syntax

4.6 CTL*

CTL* is a superset of CTL (seen in section 4.5), and is used in much the same applications. It was introduced by Emerson and Halpern [60], and fully axiomatized by Reynolds [61]. The labelled natural deduction system given here is based on the one by Rentería [5, §7].

Syntax The syntax of a CTL* formula ϕ is given by the formal grammar in Figure 4.15. Note that $\mathbf{p}$ is any propositional formula.

Semantics A model $\mathcal{M}$ for CTL* is a triple $(S, \Rightarrow, L)$, where S is a set of states, $\Rightarrow$ is a relation between states ($\Rightarrow \subseteq S \times S$) determining when one state is succeeded from another, and L is a function from S to the power-set of the atomic formulas of CTL, determining which atomic formulas are true in which states. We require that $\forall s \in S. \exists r \in S. s \Rightarrow r$, that is, that every state have at least one successor state. A path in $\mathcal{M}$ is a sequence $(s_k)_{k=0}^{\infty}$ with $(s_k, s_{k+1}) \in \Rightarrow$. A CTL* formula ϕ is entailed from a model $\mathcal{M}$ on a path ρ by recursion over ϕ:

$\bot$. $(\mathcal{M}, \rho) \nvDash \bot$

Atomic. When p is an atomic formula of CTL*, $(\mathcal{M}, \rho) \vDash p$ iff $p \in L(\rho_0)$ (where ρ_0 is the first state in the path ρ).

$\to$. $(\mathcal{M}, \rho) \vDash \phi \to \psi$ iff $(\mathcal{M}, \rho) \nvDash \phi$ or $(\mathcal{M}, \rho) \vDash \psi$

$\mathcal{X}$. $(\mathcal{M}, \rho) \vDash \mathcal{X}\phi$ iff $(\mathcal{M}, \rho_{\geq 1}) \vDash \mathcal{X}\phi$ (where $\rho_{\geq 1}$ is the same path as ρ except starting at ρ_1 instead of ρ_0).

$\mathcal{U}$. $(\mathcal{M}, \rho) \vDash \phi \,\mathcal{U}\, \psi$ iff there is an $i \geq 0$ such that $(\mathcal{M}, \rho_{\geq i}) \vDash \psi$ and for all $0 \leq j < i$ we have that $(\mathcal{M}, \rho_{\geq j}) \vDash \phi$.

$\mathcal{E}$. $(\mathcal{M}, \rho) \vDash \mathcal{E}\phi$ iff there is a path ρ' such that $\rho_0 = \rho'_0$ and $(\mathcal{M}, \rho') \vDash \phi$.

The $\mathcal{G}$ logical operator is defined syntactically, not semantically:

($\mathcal{G}$) $\mathcal{G}\phi \coloneqq \neg(\top \,\mathcal{U}\, \neg\phi)$

$$\dfrac{\mathcal{X}\varphi : a, l \quad \chi : k}{\chi : k}\,\mathcal{X}E \qquad\qquad \dfrac{\varphi \to \psi : l \quad \varphi : l \quad \chi : k}{\chi : k}\,{\to}\mathrm{E}$$

Figure 4.16: Schematic versions of non-schematic rules for CTL* system

($\neg$) $\neg\phi \coloneqq \phi \to \bot$

($\top$) $\top \coloneqq \neg\bot$

A CTL* formula ϕ is valid whenever $M, b \vDash \phi$ holds for every pair $(\mathcal{M}, b)$ where $\mathcal{M}$ is a model and b is a path on $\mathcal{M}$.

Labelled Natural Deduction system In the system for CTL*, labels represent infinite sequences of states. In the rules and in the following text, we use the symbols a, a_i to denote single states, the symbols x, x_i to denote *finite* sequences of states, and the symbols l, l_i to denote *infinite* sequences of states. Since a well-formed label always represents an infinite sequence of states, it may be formed by any number of symbols representing single states and finite sequences of states that must be followed by a symbol representing an infinite sequence of states. Label elements are separated by a comma. Intuitively, we may say a labelled formula $\varphi : l$ means that $(\mathcal{M}, l) \vDash \varphi$.

Rules IndD, IndR and KE all have special provisos specifying that the hypotheses of their discharging premises are unique (e.g., in the KE rule the only hypothesis on which $\psi : l'$ depends is $\varphi : l'$). Rule KI has a similar proviso, stating that $\mathcal{E}\mathcal{X}\varphi : l$ is the conclusion of a sub-derivation whose only hypothesis is $\varphi : l$. Finally, rule $\star$ requires that all hypotheses on which $\varphi : l$ depends have already been discharged.

Several rules of this CTL* system are not schematic. The IndD, IndR, $\bot$, p, and $\star$ rules obviously do not fit the schemas, due to their structural nature. The KE and KI rules also do not fit the schemas, since they involve two logical connectives instead of a single one. The last rule that is not schematic is the $\mathcal{U}$ rule, because it introduces a connective from another instance of itself. Note that all these rules can nevertheless be implemented in GLF, even if they do not fit the schemas; see Chapter 5 for how rules are implemented in the framework.

$$\frac{\varphi : l}{\mathcal{X}\varphi : a, l}\,\mathcal{X}I \qquad \frac{\mathcal{X}\varphi : a, l}{\varphi : l}\,\mathcal{X}E$$

$$\frac{\varphi : a, l}{\mathcal{E}\varphi : a, l'}\,\mathcal{E}I \qquad l \notin k\,\frac{\mathcal{E}\varphi : a, l' \quad \begin{array}{c}[\varphi : a, l]\\ \vdots\\ \chi : k\end{array}}{\chi : k}\,\mathcal{E}E$$

$$\frac{\psi : l}{\varphi\,\mathcal{U}\,\psi : l}\,\mathcal{U}\,I \qquad \frac{\varphi\,\mathcal{U}\,\psi : a, l \quad \begin{array}{c}[\varphi : a, l]\\ \vdots\\ \chi : k\end{array} \quad \begin{array}{c}[\varphi\,\mathcal{U}\,\psi : l]\\ \vdots\\ \chi : k\end{array} \quad \begin{array}{c}[\psi : a, l]\\ \vdots\\ \chi : k\end{array}}{\chi : k}\,\mathcal{U}\,E_1$$

$$\frac{\varphi : a, l \quad \varphi\,\mathcal{U}\,\psi : l}{\varphi\,\mathcal{U}\,\psi : a, l}\,\mathcal{U} \qquad l \notin k\,\frac{\varphi\,\mathcal{U}\,\psi : x, l \quad \begin{array}{c}[\psi : l]\\ \vdots\\ \chi : k\end{array}}{\chi : k}\,\mathcal{U}\,E_2$$

$$\frac{\begin{array}{c}\varphi : l\\ \vdots\\ \mathcal{E}\mathcal{X}\varphi : l\end{array}}{\mathcal{E}\mathcal{G}\varphi : l}\,\mathrm{KI} \qquad \frac{\mathcal{E}\mathcal{G}\varphi : l \quad \begin{array}{c}[\varphi : l']\\ \vdots\\ \psi : l'\end{array}}{\mathcal{E}\mathcal{G}\psi : l}\,\mathrm{KE}$$

$$\frac{\varphi : x, l \quad \begin{array}{c}[\varphi : a, l]\\ \vdots\\ \varphi : l\end{array}}{\varphi : l}\,\mathrm{IndD} \qquad \frac{\varphi : l \quad \begin{array}{c}[\varphi : l]\\ \vdots\\ \varphi : a, l\end{array}}{\varphi : x, l}\,\mathrm{IndR}$$

$$\frac{\begin{array}{c}[\varphi : l]\\ \vdots\\ \psi : l\end{array}}{\psi : l}\,{\to}\mathrm{I} \qquad \frac{\varphi \to \psi : l \quad \varphi : l}{\psi : l}\,{\to}\mathrm{E}$$

$$\frac{\bot : l}{\bot : l'}\,\bot \qquad \frac{p : a, l}{p : a, l'}\,\mathrm{p} \qquad \frac{\varphi : l}{\varphi : l'}\,\star$$

Figure 4.17: Natural Deduction rules for CTL* labelled deductive system

$$\phi ::= \alpha \mid x : \alpha$$
$$\alpha, \beta ::= A \mid \bot \mid \top \mid \neg\alpha \mid \alpha \sqcap \beta \mid \alpha \sqcup \beta \mid \alpha \rightharpoonup \beta \mid \exists R.\alpha \mid \forall R.\alpha$$

Figure 4.18: iALC syntax

4.7 iALC

iALC is an intuitionistic description logic designed to model and reason about legal norms/rules, presented by Haeusler *et al.* in [62, 63]. A labelled Natural Deduction system has been proposed by Alkmim [64, §2–3],[4] and shown to be sound and complete. The presentation given here is due to Alkmim's work.

Syntax An iALC formula ϕ follows the grammar in Figure 4.18. α and β are concepts, while A denotes an atomic concept, R an atomic role, and x a nominal. Note that formulas have restricted use of nominals since nominal assertions do not construct concepts, and be careful not to confuse the operator used to join formulas with nominals ($x : \varphi$) with the labelling operator ($\varphi : l$).

Semantics To define iALC we need four finite sets: N_C is the set of atomic concept names, N_R is the set of role names, N_N is the set of nominals, and Δ is the set of individuals.

For a constructive semantics of iALC, we use a structure $\mathcal{I} = (\Delta^{\mathcal{I}}, \preceq^{\mathcal{I}}, \cdot^{\mathcal{I}})$, where $\Delta^{\mathcal{I}}$ is a non-empty set of entities (each entity being representing a partially defined individual); $\preceq^{\mathcal{I}}$ is a refinement pre-ordering on $\Delta^{\mathcal{I}}$; and $\cdot^{\mathcal{I}}$ is interpretation function mapping each role $R \in N_R$ to a binary relation $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$, and each atomic concept $A \in N_C$ to a set $A^{\Delta^{\mathcal{I}}} \subseteq \Delta^{\mathcal{I}}$ which is closed under refinement (that is, we have that $x^{\mathcal{I}} \in A^{\mathcal{I}}$ and $x^{\mathcal{I}} \preceq^{\mathcal{I}} y^{\mathcal{I}}$ imply $y^{\mathcal{I}} \in A^{\mathcal{I}}$).

The semantics for iALC piggyback on those of **IK** (see Simpson [17] or Plotkin [65]), therefore a structure $\mathcal{I}$ is a model for iALC iff it satisfies the following frame conditions:

(F1) if $x \preceq x'$ and $x\,R\,y$ then $\exists y'.\, x' R\, y'$ and $y \preceq y'$;

(F2) if $y \preceq y'$ and $x\,R\,y$ then $\exists x'.\, x' R\, y'$ and $x \preceq x'$.

[4] See also Alkmim *et al.* [8], but note that this version of the system has an error in the proof of correctness of the $\exists$E, which to be corrected needed a few amendments to the system.

We extend the interpretation $\mathcal{I}$ from atomic concepts to concepts by doing:

$$
\begin{aligned}
\top^{\mathcal{I}} &= \Delta^{\mathcal{I}} \\
\bot^{\mathcal{I}} &= \varnothing \\
(\alpha \sqcap \beta)^{\mathcal{I}} &= \alpha^{\mathcal{I}} \cap \beta^{\mathcal{I}} \\
(\alpha \sqcup \beta)^{\mathcal{I}} &= \alpha^{\mathcal{I}} \cup \beta^{\mathcal{I}} \\
(\neg\alpha)^{\mathcal{I}} &= \{x \in \Delta^{\mathcal{I}} \mid \forall y, x \preceq y \rightarrow y \notin \alpha^{\mathcal{I}}\} \\
(\forall R.\alpha)^{\mathcal{I}} &= \{x \in \Delta^{\mathcal{I}} \mid \forall y(x \preceq y \rightarrow \forall z((y,z) \in R^{\mathcal{I}} \rightarrow z \in \alpha^{\mathcal{I}}))\} \\
(\exists R.\alpha)^{\mathcal{I}} &= \{x \in \Delta^{\mathcal{I}} \mid \forall y(x \preceq y \rightarrow \exists z((y,z) \in R^{\mathcal{I}} \wedge z \in \alpha^{\mathcal{I}}))\} \\
(\alpha \rightarrowtail \beta)^{\mathcal{I}} &= \{x \in \Delta^{\mathcal{I}} \mid \forall y, (x \preceq y \wedge y \in \alpha^{\mathcal{I}}) \rightarrow y \in \beta^{\mathcal{I}}\} \\
(x : \phi)^{\mathcal{I}} &= x^{\mathcal{I}} \Vdash_{\mathcal{I}} \phi^{\mathcal{I}} \qquad (\Vdash_{\mathcal{I}} \text{ is the usual satisfaction relation of Kripke models})
\end{aligned}
$$

If we define a knowledge base $\mathcal{K}$ as a pair $\langle \mathcal{T}, \mathcal{A} \rangle$, we have that a interpretation $\mathcal{I}$ satisfies the TBox $\mathcal{T}$ iff:

$$
\begin{aligned}
\mathcal{I} \Vdash \alpha \rightarrowtail \beta \quad &\text{iff} \quad \forall w \in \Delta^{\mathcal{I}}, w \in (\alpha \rightarrowtail \beta)^{\mathcal{I}} \\
\mathcal{I} \Vdash \mathcal{T} \quad &\text{iff} \quad \forall \Phi \in \mathcal{T}, \mathcal{I} \Vdash \Phi
\end{aligned}
$$

An interpretation satisfies the ABox $\mathcal{A}$ iff:

$$
\begin{aligned}
\mathcal{I} \Vdash x : \alpha \quad &\text{iff} \quad \forall x_1(x^{\mathcal{I}} \preceq x_1^{\mathcal{I}} \rightarrow x_1^{\mathcal{I}} \in \alpha^{\mathcal{I}}) \\
\mathcal{I} \Vdash x\, R\, y \quad &\text{iff} \quad \forall x_1(\forall y_1((x^{\mathcal{I}} \preceq x_1^{\mathcal{I}} \wedge y^{\mathcal{I}} \preceq y_1^{\mathcal{I}}) \rightarrow (x_1^{\mathcal{I}}, y_1^{\mathcal{I}}) \in R^{\mathcal{I}})) \\
\mathcal{I} \Vdash \mathcal{A} \quad &\text{iff} \quad \forall \Phi \in \mathcal{A},\ \mathcal{I} \Vdash \Phi
\end{aligned}
$$

If both $\mathcal{I} \Vdash \mathcal{T}$ and $\mathcal{I} \Vdash \mathcal{A}$, then $\mathcal{I}$ is a model for knowledge base $\mathcal{K}$.

Labelled Natural Deduction system In this system, formulas are labelled with lists of existential or universal restrictions (with the empty list being denoted by $\varnothing$). Unlike the other system we have seen, sub-formulas in iALC may also be labelled, but this label is always empty unless the the formula is of the form $(\alpha : l_1) \rightarrowtail (\beta : l_2) : l$; in this case, we either have that $l_1 = l_2 = \varnothing$ or that $l = \varnothing$. To prevent extraneous notation, whenever a label is empty, we omit it. We establish as a convention that the nominal operator $:$ has a lesser precedence than the label operator :, so that we can reduce the number of parentheses used in the formulas.

iALC labels form a kind of context, as becomes more evident by the

rules pertaining to the universal and existential operators. If we have that $y : \alpha : \exists Rx$ in the ABox, we also have that $x\,R\,y$ and $x : \exists R.\alpha$.

We use α, β, δ to denote concepts, x, y, z are meta-variables for nominals, R denotes a role. As usual, l, m, u denote labels. Note that when we write $l^{\forall}$ we mean that the label l only has universal restrictions as its elements, and likewise for $l^{\exists}$ — both are used as provisos in some rules.

The rules for the original system for iALC are in Figure 4.19, while the rules that can be changed to become schematic are shown in Figure 4.20.

We omit the rules for negation introduction and elimination since they can be obtained from the rules for implications, changing the consequent of the implication to be $\bot$, and aliasing $\alpha \rightarrowtriangle \bot$ to $\neg\alpha$.

For rules $\exists$I, $\exists$E, $\forall$I, and $\forall$E we use a formula of the form $x\,R\,y$ as premise. This calculus for iALC is TBox-centered, but there must be an ABox extension from which these assertions are taken.

Some rules have special provisos. For the $\exists$E rule, z must not appear in any undischarged hypotheses, while for the $\forall$I, Gen, dist, chng, and join rules, the variable y must not appear in any undischarged hypotheses.

A few rules of the present system are not schematic. Most obviously, the fist, chng, and join rules do not fit any of the schemas, primarily because they do not introduce nor eliminate a connective; they can be seen as structural rules. Similar reasoning applies to the Gen rule, although it can be thought of as introducing a restriction on the formula's label. The $\rightarrowtriangle$I and $\rightarrowtriangle$E rules which would otherwise be schematic fail being so due to their breaking of the invariant specifying exactly one label per formula. In the version of the labelled Natural Deduction system presented by Alkmim *et al.* in [8], the $\exists$I, $\exists$E, $\forall$I, and $\forall$E rules do not have the $x\,R\,y$ premise that renders them non-schematic, but still requires that assertion to hold in the ABox for the rule application to be valid, similarly to a rule proviso.

See Alkmim [64, §3] for more the details about the labelled Natural Deduction system for iALC logic, or iALC in general.

$$x \neq y \; \dfrac{\begin{array}{c}[x\,R\,y]\\ \vdots \\ y \mathbin{:} \alpha : \forall Rx, l\end{array}}{x \mathbin{:} \forall R.\alpha : l}\forall\mathrm{I} \qquad \dfrac{x \mathbin{:} \forall R.\alpha : l \quad x\,R\,y}{y \mathbin{:} \alpha : \forall Rx, l}\forall\mathrm{E}$$

$$\dfrac{x\,R\,y \quad y \mathbin{:} \alpha : \exists Rx, l}{x \mathbin{:} \exists R.\alpha : l}\exists\mathrm{I} \qquad x \neq y \neq z \; \dfrac{x \mathbin{:} \exists R.\alpha : l \qquad \begin{array}{c}[y \mathbin{:} \alpha : \exists Rx, l]\,[x\,R\,y]\\ \vdots \\ z \mathbin{:} \beta : m\end{array}}{z \mathbin{:} \beta : m}\exists\mathrm{E}$$

$$\dfrac{\begin{array}{c}[x \mathbin{:} \alpha : l]\\ \vdots \\ x \mathbin{:} \beta : m\end{array}}{x \mathbin{:} (\alpha : l) \multimap (\beta : m)}\multimap\mathrm{I} \qquad \dfrac{x \mathbin{:} (\alpha : l) \multimap (\beta : m) \qquad x \mathbin{:} \alpha : l}{x \mathbin{:} \beta : m}\multimap\mathrm{E}$$

$$l^{\forall}\,\dfrac{x \mathbin{:} \alpha : l \quad x \mathbin{:} \beta : l}{x \mathbin{:} \alpha \sqcap \beta : l}\sqcap\mathrm{I} \qquad l^{\forall}\,\dfrac{x \mathbin{:} \alpha \sqcap \beta : l}{x \mathbin{:} \alpha : l}\sqcap\mathrm{E}_1 \qquad l^{\forall}\,\dfrac{x \mathbin{:} \alpha \sqcap \beta : l}{x \mathbin{:} \beta : l}\sqcap\mathrm{E}_2$$

$$l^{\exists}\,\dfrac{x \mathbin{:} \alpha : l}{x \mathbin{:} \alpha \sqcup \beta : l}\sqcup\mathrm{I}_1 \qquad l^{\exists}\,\dfrac{x \mathbin{:} \beta : l}{x \mathbin{:} \alpha \sqcup \beta : l}\sqcup\mathrm{I}_2$$

$$l^{\exists}\,\dfrac{x \mathbin{:} \alpha \sqcup \beta : l \quad \begin{array}{c}[x \mathbin{:} \alpha : l]\\ \vdots \\ z \mathbin{:} \delta : m\end{array} \quad \begin{array}{c}[x \mathbin{:} \beta : l]\\ \vdots \\ z \mathbin{:} \delta : m\end{array}}{z \mathbin{:} \delta : m}\sqcup\mathrm{E} \qquad l^{\exists}\,\dfrac{x \mathbin{:} \bot : l}{z \mathbin{:} \delta : m}\mathrm{efq}$$

$$x \neq y \; \dfrac{\begin{array}{c}[x\,R\,y]\\ \vdots \\ y \mathbin{:} \alpha : l\end{array}}{y \mathbin{:} \alpha : l, \forall Rx}\mathrm{Gen} \qquad x \neq y \; \dfrac{y \mathbin{:} (\alpha : l, \forall Rx) \multimap (\beta : m, \forall Rx) \qquad x\,R\,y}{y \mathbin{:} (\alpha : l, \exists Rx) \multimap (\beta : m, \exists Rx)}\mathrm{chng}$$

$$x \neq y, l^{\forall}\,\dfrac{y \mathbin{:} \alpha \multimap \beta : l, \forall Rx}{y \mathbin{:} (\alpha : l, \forall Rx) \multimap (\beta : l, \forall Rx)}\mathrm{dist} \qquad x \neq y, l^{\forall}\,\dfrac{y \mathbin{:} (\alpha : l, \exists Rx) \multimap (\beta : l, \forall Rx)}{y \mathbin{:} \alpha \multimap \beta : l, \forall Rx}\mathrm{join}$$

Figure 4.19: Natural Deduction rules for iALC labelled deductive system

$$l^{\forall}\,\dfrac{x \,:\, \alpha \sqcap \beta : l \qquad \begin{array}{c}[x \,:\, \alpha : l]\,[x \,:\, \beta : l] \\ \vdots \\ z \,:\, \delta : m\end{array}}{z \,:\, \delta : m}\,\sqcap\mathrm{E}$$

Figure 4.20: Schematic versions of non-schematic rules for iALC system

5
Implementation

We have previous experience with the implementation of logical frameworks [6], and the work on this thesis was born from the need of formalizing this previous implementation we had developed. The logical framework herein described is implemented as a Haskell library which exposes the rule application engine, supported by a storage back-end, provided by the Postgres database management system. We use a real database management system instead of an *ad hoc* solution so we can store and query large proofs without fear of losing data due to memory or programming limitations. An HTTP proof server and a command-line interface are built on top of the library, and a graphical user interface (in the form of a web application) depends on the proof server to interact with the underlying database (through an HTTP API). Users can define their own deductive systems, and no distinction is made over those distributed with the application and those that are user-defined (e.g., there is no special support for the built-in systems, they can be redefined by users without loss of functionality). See Figure 5.1 for a diagram showing an overview of the implementation of GLF.

To implement a new deductive system using our system, one describes the syntax of its formulas (and of the formula's labels, if the system is labelled) and defines the system's rules. Rules are described in an embedded domain-specific language close to how rules are defined traditionally on paper; see Figure 5.2

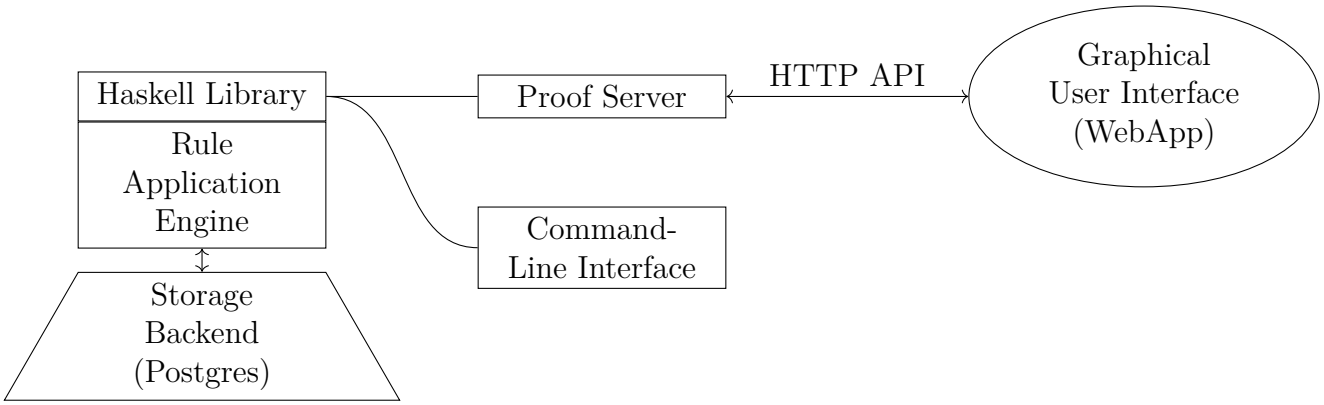

Figure 5.1: Overview of GLF implementation

```
disjunctionElimination
  = rule "∨E"
      [ premise "major" ("φ" `or` "ψ")
      , premise "left"  "γ" `discharges` ("leftHypothesis",  "φ")
      , premise "right" "γ" `discharges` ("rightHypothesis", "ψ")
      , concludes "γ"
      ]
```

Figure 5.2: Haskell code for the disjunction elimination rule

Figure 5.3: Implementation welcome screen, with user hovering over CTL system option

for the definition of disjunction elimination in a non-labelled system. Once a system is defined, it shows up in the web interface as an option the user can choose (see Figure 5.3).

Once the user chooses a system they want to prove something on, they are asked to input which deduction they would like to carry out, specifying their assumptions and the conclusion they would like to reach (see Figure 5.4). There is a help section that when active explains the system's syntax.

After inputting the new deduction information, the deduction page shows up, in a Natural Deduction style close to that of Fitch. Figure 5.5 shows an ongoing proof of the undecidability of the machine halting problem (a version of which was proved by Turing [66]) in unlabelled First-order logic. We want to show that there exists no program that can receive a program (being a program is characterized by the `.IsP` predicate) as input, and tell whether it eventually stops or not given a certain input (stopping when given a certain input is characterized by the `.Stops` predicate, whose first argument is the program and the second is its input, which may be a program too). To carry out the proof, we assume a program `.m` with the property that it always stops whenever its input program does not stop when running on itself as input.

The deduction shown in Figure 5.5 corresponds to deduction shown in

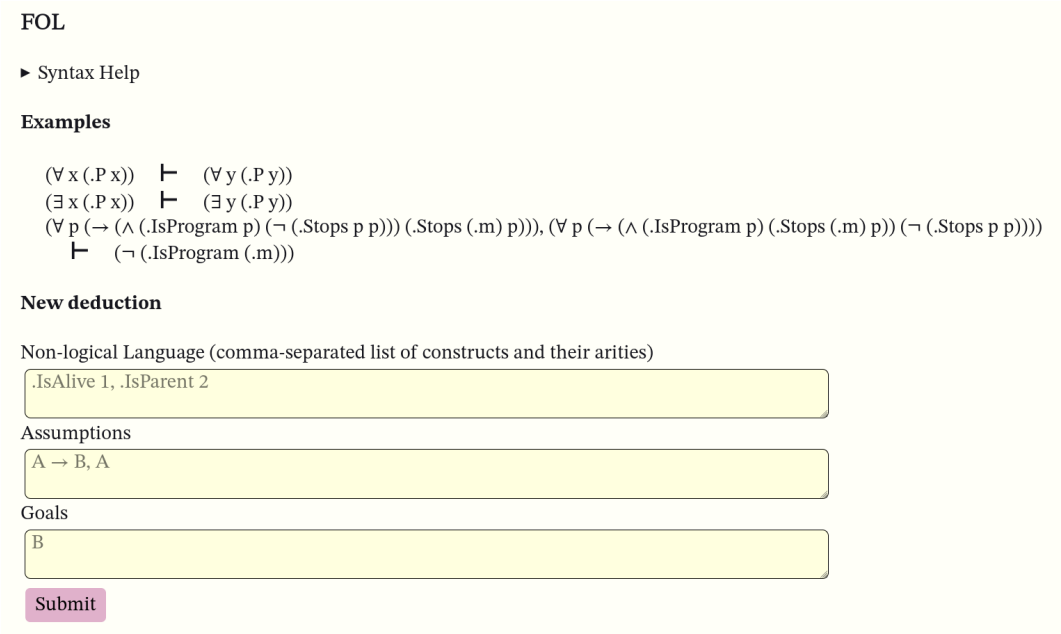

Figure 5.4: Implementation deduction input screen

figure Figure 5.6 in more traditional Gentzen-style deduction.

The initial assumptions of the proof are entered automatically, then the user may choose which rule they want to apply and to which deduction lines (please refer to Figure 5.5). Additionally the user may have to type the whole result formula (as is the case for new assumptions, or for the ∀E rule). Often nothing else is needed, e.g., in the case of the ∧I rule resulting in line 5, the user only has to specify the deduction lines 3 and 4, and the system already knows the resulting formula is the conjunction of the argument formulas, given the definition of the rule itself.

To apply a rule, the system takes the rule definition (as the example in Figure 5.2) and checks whether the premises provided by the user match (by unification) the ones specified. If so, it discards any formulas that need to be discarded, and builds the result given the specified conclusion formula.

5.1 Defining a logical system

To define a logical system in GLF, we must define its rules. As we see in Figure 5.2, defining a rule in GLF takes a declaration similar to what one would use to define a rule in a textbook. The full grammar[1] specification for rule definition is shown in Figure 5.7; it depends on the notions of **name** and

[1]Note that the grammars in this section are not parsing grammars, but generative ones intended to describe the API the user of GLF can use to describe deductive rules.

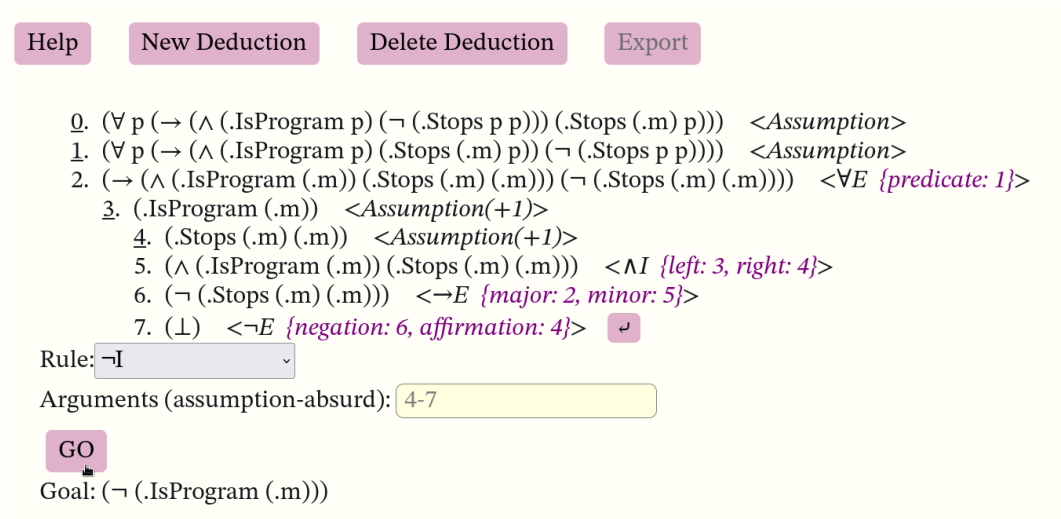

Figure 5.5: Part of the proof of the undecidability of the halting problem

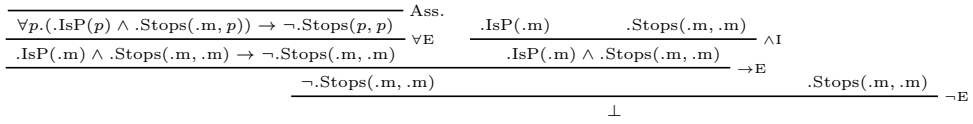

Figure 5.6: Part of the proof of the undecidability of the halting problem (Gentzen-style)

$$
\begin{aligned}
\langle rule \rangle &::= \textbf{name}\ \langle premise \rangle^*\ \langle conclusion \rangle\ \langle proviso \rangle^* \\
\langle premise \rangle &::= \textbf{name}\ \langle formulaspec \rangle\ \langle discharge \rangle^* \\
\langle discharge \rangle &::= \textbf{name}\ \langle formulaspec \rangle \\
\langle conclusion \rangle &::= \langle formulaspec \rangle \\
\langle proviso \rangle &::= \textbf{variable}\ \text{‘freeIn’}\ \langle formulaspec \rangle \\
&\quad \mid \textbf{variable}\ \text{‘freeIn’}\ \text{‘hypotheses’}\ \langle formulaspec \rangle * \\
&\quad \mid \textbf{name}\ \text{‘hypothesesMatch’}\ \langle formulaspec \rangle^* \\
&\quad \mid \text{‘not’}\ \langle proviso \rangle
\end{aligned}
$$

Figure 5.7: A grammar for the rule definition eDSL

⟨*formulaspec*⟩ ::= ‘Any’
| ‘Atomic’
| ‘Compound’ ⟨*operator*⟩? ⟨*formulaspec*⟩∗
| ‘Eval’ **function** ⟨*formulaspec*⟩∗
| ‘Substitute’ ⟨*formulaspec*⟩ ‘For’ **name** ‘In’ **name**
| ‘Named’ **name** ⟨*formulaspec*⟩
⟨*operator*⟩ ::= **name** ⟨*associativity*⟩ **integer**∗
⟨*associativity*⟩ ::= ‘Prefix’ **integer**
| ‘Postfix’ **integer**
| ‘RightInfix’ **integer**
| ‘LeftInfix’ **integer**
| ‘NoneInfix’ **integer**
| ‘NoneFixedArity’ **integer**
| ‘NoneVariableArity’ **integer**

Figure 5.8: A grammar for the formula specification eDSL

variable — which are equivalent to a Haskell string — and on the notion of a *formula specification*.[2]

As described in the grammar, a logical rule has a name, and may have any number of premises — remember that axioms are treated as rules of inference with no premises. The use of **name** for naming premises is the main difference from the definition of logical rules in GLF from the definitions usually seen on paper. Naming premises allows us to provide better error diagnostics for the user, for example saying that there was an error matching the formula provided for a premise with the formula the rule expected. A rule must have a single conclusion, and it may have any number of provisos.

The rule provisos are seen as logical primitives we implement directly. The first proviso describes a variable being free in a formula matching a certain specification; the second describes a variable being free in all the undischarged hypotheses at a certain point of a deduction, except those matching the provided formula specifications; the third one enforces that all of the undischarged hypotheses of the named premise match the provided formula specification. These three primitives (plus negation) found in Figure 5.7 are enough to implement all the systems shown in Chapter 4.

[2]In this text we treat **name** as simple string, but the actual implementation is a triple of strings: one in the ASCII character set, one in the Unicode character set, and the other being a LaTeX command that generates that name, suitable to exporting proofs to this format.

```
binaryOp :: Text -> Int -> Operator
-- | Define a binary operator that does not associate
binaryOp name prec =
  Op { name = name
     , associativity = InfixNone (Precedence prec)
     , binds = []
     }

label :: Operator
label = binaryOp ":" 500

infix 1 .:
(.:) :: FormulaSpec -> FormulaSpec -> FormulaSpec
formula .: theLabel = op label <> opn 1 formula <> opn 2 theLabel
```

Figure 5.9: Definition of label operator and helper function for formula specification

Defining a rule's premises and its conclusion involves describing the form of the related formulas. Specifying formulas in GLF is also done through an embedded domain-specific language, whose grammar is in Figure 5.8. A formula specification may say that a formula may be anything; or that a formula must be atomic; or that is a compound formulas whose operator follows a certain pattern, and whose operands match the provided formula specifications.

A logical operator is a structure of **name**, along with associativity, arity, and binding information, which is given as integers — if we say an operator binds the integer 1 then the first operand represents the variable being bound, and any other instances of the formula of that operand are considered to be occurrences of that bound formula. The associativity of an operator includes both its associativity information, but also its precedence (as an integer). In the case of non-associative operators, the integer is the arity of the operator, with those of variable arity having the integer be the lower bound on the number of operands.

See Figure 5.9 for the definition of the label operator, and an associated helper function ((`.:`), also defined as a Haskell operator) to specify formulas (the `FormulaSpec` type refers to formula specifications). We define several helper functions such as `binaryOp` to define new logical operators. More of them will be seen in Chapter 6, but they all produce the operators described in Figure 5.8. Note that we do not differentiate formulas from labels at the implementation level; labels are considered part of the formulas themselves, and constructs like label marks (like the ones from the Ultrafilter logic system in Section 4.2) are taken to be unary logical operators.

We also use helper functions to create values of type `FormulaSpec` for formula specifications. The `op` function specifies the main operator of a compound formula, while the `opn` function specifies the formula pertaining to the operator of the given index. A Haskell `Semigroup` instance for `FormulaSpec` is used to allow us to use the Haskell `<>` operator to be used to construct formula specifications, resulting in more readable syntax.

A formula specification may also be named, which allow us to refer to them afterwards, be it in provisos, or in other formulas — in the rule definition from Figure 5.2, naming the left disjunct φ allows us to refer to it and discharge it in the first discharging premise.

Finally, a formula specification may also determine that a formula must be the result of a substitution (which is checked to be valid), or the result of the evaluation of a **function**. A **function** is Haskell-defined function that takes any number of formulas and returns a new one. **function** is used to define label constructing formulas such as the ones we saw in Chapter 4.

Chapter 6 includes many examples of the definition of logical rules, which include examples of formula specification and of logical operator definition.

5.2 User interfaces

Following the advice of Thery *et al.* [67], we separate the implementation of the LF 'core' from its user interfaces. The core itself offers only one programmatic interface: its use as a Haskell library (usually by other Haskell programs, although there are foreign function interfaces for other languages). On top of this interface, we build an API served over the HTTP protocol that can be accessed from any programming language, and a somewhat limited command-line user interface (CLI). The HTTP API is the back-end of a web-based graphical user interface implemented in the Elm programming language. The core and UI code are kept in sync by code generation: not only is the API-calling Elm code generated from the Haskell API definition, but so are the type definitions in Elm derived from the Haskell ones.

This web-based interface is the interface we expect most end-users to employ, while the CLI is mainly used for administrative (e.g., starting the proof server) and testing purposes.

The web-based interface is a multi-user web application served over a secure HTTPS connection with user authentication.[3] Non-concomitant user collaboration is possible, but features like real-time collaboration have not been

[3] An instance is available at `https://glf.tecmf.inf.puc-rio.br/`, while the source code is at `https://gitlab.com/odanoburu/glf` (branch `dev`). Note that the instance often lags behind the latest developments.

implemented. The architecture is similar to that of other web-based provers like the one by Kaliszyk [68] or the one by Santos [69], differing only in the choice of technology stack.

Interface-wise GLF is very different from works like the one by Kaliszyk [68] or the Lean prover web editor, however, primarily due to the nature of Coq and of Lean as metalanguages. Both of these web interfaces for Coq and for Lean try to emulate the local (non-web) editing experience, with the difference that the web editor for version 3 of the Lean proof assistant[4] is client-only, being a duplicate — slower, less tested — implementation of Lean 3 in Javascript. The upcoming implementation of Lean version 4[5] uses a more traditional client-server implementation, but does not change the interface paradigm.

While we present a 'proof by clicking' of sorts (inspired by the work of Bertot [70]), Kaliszyk's Coq interface and the Lean web editor propose a text/programming-based interface. As previously stated, Kaliszyk's Coq web interface and the Lean web interface both strive to emulate the local Coq/Lean user interfaces, while GLF's web interface strives for a user experience closer to that of a pen-and-paper proof. Interestingly, both the Carnap [71] and the NADIA web proof assistants [72] also strive for a similar experience to traditional Natural Deduction proofs, but they use a text-based interface that is closer to Kaliszyk's or to the Lean web editor than to the 'proof by clicking' approach we take. Unlike Kaliszyk's interface for Coq or the Lean web editor, however, these text-based interfaces are not for a programming language, but for a small formal language representing the logical systems they implement.

GLF is not the only web proof assistant available following a point-and-click paradigm for its interface. There are also FitchFX,[6] Logitext,[7], the Incredible Proof Machine by Breitner [73], and others. All of these three are mainly intended to be used as pedagogic tools, with FitchFX implementing a Fitch-style Natural Deduction system for first-order logic, Logitext implementing sequent calculi for classical and intuitionistic logic, and the Incredible Proof Machine with implementations of systems for first-order logic, for Church's P_2 Hilbert system, and others. FitchFX's interface is very similar to GLF's — they are both implementation of Fitch-style Natural Deduction systems after all — while Logitext's is much more purely point-and-click since it requires less textual input from the user due to the nature of

[4]Available at `https://leanprover-community.github.io/lean-web-editor/` or `https://lean-lang.org/tutorial/`, last accessed on 2023-09-10.

[5]Available at `https://github.com/leanprover-community/lean4web`, last accessed on 2023-09-10.

[6]Available at `https://mrieppel.github.io/FitchFX/`, last accessed on 2023-09-10.

[7]Available at `http://logitext.mit.edu/main`, last accessed on 2023-09-10.

Name	Interface Paradigm	User-extensible
Kaliszyk + Coq	Text-based (programming)	Yes
NADIA	Text-based	No
The Incredible Proof Machine	Point-and-click (block-based)	Yes
lean4web	Text-based (programming)	Yes
Logitext	Point-and-click	No
FitchFX	Point-and-click	No
Carnap	Text-based	Yes

Table 5.1: Comparison between web proof assistant implementations

the sequent calculus system it implements. In Logitext, one single click is often enough to apply an inference rule, while in GLF and the other point-and-click interfaces usually require a few clicks and some text (formula) input.

The Incredible Proof Machine differs from the other point-and-click interfaces in that it does not attempt to simulate a deduction as it would be carried out on paper, but opts instead for a graph-based interface. One connects blocks (nodes) representing rules, assumptions and conclusions to create a formal proof, as if one were playing a game or deciphering a puzzle. See Breitner [73] for more details on how it works and the formalism of *port graphs* backing the idea.

See Table 5.1 for a summary of the discussion carried out in this section. Santos [69, §4] has a longer discussion of the user interfaces for interactive theorem proving in general, including some logical frameworks.

6
Compared implementations

In this chapter we describe two experiments comparing GLF to other proof assistants. In the first part we show an implementation of a system for modal logic K in four different proof assistants. This system for K is a simpler, unlabelled axiomatic system; the comparison is broader, but also shallower, focusing on the usability of the different proof assistants. In the second part we pick a more involved object of comparison: the proof of De Zolt's postulate in three-dimensional space. This theorem is not provable in ZFC, but can be proved in the Z_p system, which we then implement in the Lean proof assistant and in GLF.

$$\phi ::= \bot \mid \phi \to \phi \mid \neg\phi \mid \Box\phi \mid \Diamond\phi$$

Figure 6.1: K logic syntax

$$\frac{\varphi}{\Box\varphi}\,\Box\text{I} \qquad\qquad \frac{\varphi \to \psi \quad \varphi}{\psi}\,\text{MP}$$

$$\frac{}{\Box(\varphi \to \psi) \to (\Box\varphi \to \Box\psi)}\,\text{K}$$

$$\frac{}{\Diamond\varphi \to \neg\Box\neg\varphi}\,\text{DUAL}_1 \qquad\qquad \frac{}{\neg\Box\neg\varphi \to \Diamond\varphi}\,\text{DUAL}_2$$

Figure 6.2: Axiomatic system for K

6.1 Axiomatic K in four proof assistants

Modal logic K can be seen as classical propositional logic extended with the two *modal* notions of necessity and possibility. Regular classical logic can not represent well *modal* statements such as "it is necessarily possible that the sun may not rise tomorrow", and modal logics were created to fill this niche. In this experiment we will implement a version of an axiomatic system for K in four different proof assistants, GLF among them. First we will introduce the axiomatic system for K, then we will show the implementations in Agda (Section 6.1.1), Isabelle (Section 6.1.2), Metamath (Section 6.1.3), and GLF (Section 6.1.4). After presenting the implementations we will compare them.

The axiomatic system we will implement is a version of the one described by Zach, which is proven to be sound and complete with respect to the modal logic K in his book [74, §3,4]. The syntax of K is shown in Figure 6.1, where the unary prefix operator $\Box$ represents necessity, and the prefix unary operator $\Diamond$ represents possibility.

The K system is characterized by two inference rules: the necessitation rule (or $\Box$I) and *modus ponens* (MP), plus two axioms: K and DUAL.[1] These can be found in Figure 6.2. Additionally, we consider to be axioms of the K system any substitution instances of any tautologies.

[1]We actually break up DUAL in two separate axioms, one for each side of the bi-implication.

6.1.1 Agda

Agda is a dependently-type pure functional programming language *cum* proof assistant. It is based on Martin-Löf intuitionistic type theory [40], and was born out of Norell's PhD thesis [38]. As a programming language Agda is most similar to Haskell, Agda's implementation language. As a proof assistant, Agda is somewhat similar to the Coq and Lean provers, but it does not have a separate tactics language like them, boasting a built-in proof searcher that is sometimes able to complete proofs automatically for the user instead.

First we handle syntax. Bottom and implication[2] are already defined by Agda, so we only define negation, and the necessity and possibility operators.

```
¬_ : Set → Set
¬ A = A → ⊥
infix 3 ¬_

data ⧫_ (A : Set) : Set where

data ■_ (A : Set) : Set where
  □ :
      A
      -----
    → ■ A
```

In defining the necessity operator, we already define the necessity introduction rule (□I) as its type constructor. We then need to define the *modus ponens* rule (`mp`) and our axioms. Do note that any tautologies we use must be proved in Agda, but they are not rigorously a part of the system, and we will not be proving them before they are needed.

```
mp : ∀ {A B : Set}
  → (A → B)
  → A
    -------
  → B
mp L M = L M


-- axioms
postulate
  κ : ∀ {A B : Set}
```

[2]We actually reuse the arrow function constructor as the implication connective.

```
    ------------------------
    → ■ (A → B) → (■ A → ■ B)

  dual₁ : ∀ {A : Set}
    → ♦ A
      -----------
    → ¬ (■ (¬ A))
  dual₂ : ∀ {A : Set}
    → ¬ (■ (¬ A))
      -----------
    → ♦ A
```

While inference rules are defined as functions,[3] axioms are defined using the keyword `postulate`, which is used for this purpose. Do note that axioms have no computational content, whereas inference rules — being regular Agda functions — do.

Now that the system K implementation is done, we show a short proof as an example. Note how we needed the tautology $A \to (B \to A)$, and we proved it using Agda's built-in capabilities. We have written the proof in the most readable way we could think of, enunciating the types of the intermediate terms; the only exception is the use of the axiom K, which is instantiated to the correct type automatically, but could have been explicitly shown.

```
vacuousImplication : ∀ {A B : Set}
  → ■ A
  → ■ (B → A)
vacuousImplication {A} {B} =
  let taut : A → (B → A)
      taut = λ z _ → z
      necTaut : ■ (A → (B → A))
      necTaut = □ taut
  in mp κ necTaut
```

6.1.2 Isabelle

Isabelle [75] calls itself a generic theorem proving environment, being best known for its HOL implementation (Isabelle/HOL). Created by Paulson [21], Isabelle has gained renewed interest after the introduction of the Isar proof language by Wenzel [76] (Isabelle/Isar). Before Isar made Isabelle proofs

[3] We have previously said that the □I rule is a type constructor; in Agda and other purely functional programming languages, type constructors can be seen as little more than special functions.

more human-readable and structured, Isabelle proofs were more procedural in nature, requiring the user to know the programming language ML (Isabelle's implementation language) to expertly write them. In our formalization of K in Isabelle, we use the Isar proof language and Isabelle/Pure, Isabelle's minimal language.

Our Isabelle formalization does not use built-in logical operators, defining them from scratch instead. We start by declaring a new opaque type o for formulas, and creating a new judgement that turns them into propositions (the built-in type for meta-language truth values). This judgement is effectively embedding the object language into the meta-language. We can then proceed to describe the axiomatization of our system: we declare our logical connectives, give their operators an arity, a precedence & a fixity, and declare their respective deductive rules. An explanation of the plethora of Isabelle arrows is in order: `A ⟶ B` means that A implies B logically at the object language level, while `A ⇒ B` denotes the type of a meta language level function from type A to type B, and `A ⟹ B` denotes that from A we deduce B (at the meta-language level).

```
typedecl o

judgment
  Trueprop :: ⟨o ⇒ prop⟩  (⟨(_)⟩ 5)

subsubsection ⟨Propositional logic⟩

axiomatization (* implication *)
  imp :: ⟨o ⇒ o ⇒ o⟩  (infixr ⟨⟶⟩ 25)
  where
  mp: ⟨P ⟶ Q ⟹ P ⟹ Q⟩

axiomatization  (* bottom & negation *)
  False :: ⟨o⟩

definition Not (⟨¬ _⟩ [40] 40)
  (* syntax sugar for negation *)
  where [simp]: ⟨¬ P ≡ P ⟶ False⟩

axiomatization (* necessity and possibility *)
  box :: ⟨o ⇒ o⟩ (⟨□ _⟩ 50) and
  dia :: ⟨o ⇒ o⟩ (⟨◇ _⟩ 50)
  where
```

```
  necI: ⟨A ⟹ □ A⟩ and
  k: ⟨(□ (A ⟶ B)) ⟶ ((□ A) ⟶ (□ B))⟩ and
  ldual: ⟨¬ (□ (¬ A)) ⟹ ◇ A⟩ and
  rdual: ⟨◇ A ⟹ ¬ (□ (¬ A))⟩
```

Now that we have a complete axiomatization of K, we can show a small proof as an example:

```
lemma vacuousImplication: ⟨(□ A) ⟶ (□ (B ⟶ A))⟩
proof (rule mp)
  show ⟨(□ (A ⟶ (B ⟶ A))) ⟶ (□ A) ⟶ (□ (B ⟶ A))⟩ by (rule
   k)
next
  show ⟨□ (A ⟶ (B ⟶ A))⟩
  proof (rule necI)
    show ⟨A ⟶ (B ⟶ A)⟩ sorry
  qed
qed
```

This proof reads like a Natural Deduction proof read bottom–up, from the conclusion to the assumptions. What gives us the conclusion is the rule *modus ponens*. Applying `mp` introduces two sub-goals, which we prove in two sub-proofs. The first is a proof of $\Box(A \to (B \to A)) \to \Box A \to \Box(B \to A)$ using rule K, and the other is a proof of its antecedent using the necessitation introduction rule `necI`. For the latter sub-proof, we also needed the tautology $A \to (B \to A)$, and so we took it as an axiom (the Isabelle keyword `sorry` tells the prover to accept the absence of a proof). We could have proved the tautology if we wanted, but this would have been done outside of system K (as was done in the Agda proof in Section 6.1.1).

Constructing this proof is easier to do if one already has the proof done in pen-and-paper, but the Isabelle proof editor can help in proving it from scratch. One of the editor's interactive features that is specially helpful is the option that shows the proof state, so the user can click anywhere on the proof and see which goal is or should be proved at that point in the proof.

6.1.3 Metamath

Metamath [28] is a minimalistic logical framework and proof assistant created by Norman Megill. In keeping with this philosophy, it offers very little to the user: when implementing a system one must specify its syntax from scratch, and all it has as a built-in deductive rule is the substitution of expressions for variables. User must then codify their own axioms and inference

rules to specify their object language, and Metamath can then help construct and verify any proofs (assuming the axioms and rules the user supplied are correct).

Metamath definitions start with the definition of the symbols that will be used: constants (like bottom and the connectives) are demarcated by the keyword `$c` and meta-variables (like the φ in the definition of rule $\Box$I in Figure 6.2) are demarcated by `$v`. Metamath statements always start with a keyword and end with the `$.` symbol. Comments are anything between the `$(` and `$)` symbols.

The careful reader will notice that when declaring our constants we mixed purely logical constructs such as the connectives with purely syntactical constructs (the parentheses), but also with syntactico-semantic constructs like the notion of a well-formed formula (denoted by the `wff` symbol), and the derivation relation (denoted by `|-`). This is because core Metamath is a very minimal and flexible framework that offers little in terms of convenience[4] — almost everything is user-defined. This means that we have to define the system's formulas syntax from scratch (including parentheses and precedence), and also the derivation relation (for which we will need the notion of a well-formed formula).

We also teach Metamath about what constitutes a well-formed formula. First we say the meta-variables stand for well-formed formulas, by using the `$f` keyword. Then we go over the connectives: we state that bottom is a well-formed formula by itself (in other words, that it is a nullary connective), that the implication connective between two well-formed formulas is a well-formed formula (i.e., it is an infix binary connective), and that the necessity and possibility operators preceding a well-formed formula are well-formed formulas themselves (that is, that they are unary prefix operators). Because implementing precedence in Metamath is possible but would be too troublesome, we follow Metamath practice and don't, and so must always surround compound sub-formulas by parentheses. This simplifies the definitions and makes the system easier to use, because it needs no rewriting rules dedicated entirely to syntax, but it also means that `A -> B` is not considered a well-formed formula; similarly, due to Metamath's flexibility entailing syntax limitations, nor are `A->B` or `(A -> B)` considered well-formed formulas, since they do not have spaces separating their tokens (remember that the parentheses are also tokens).

[4]When reading a Metamath file and writing proofs one is supposed to use the **metamath** program, which does offer more user-friendly features such interactivity and displaying proofs in human-readable formats.

```
$( Declare the constant symbols we will use $)
$c -> ( ) wff |- [] <> _|_ $.

$( Declare the metavariables we will use $)
$v A B C p q $.

$( Specify properties of the metavariables $)
wa $f wff A $.
wb $f wff B $.
wc $f wff C $.
wp $f wff p $.
wq $f wff q $.

$( Define "wff" $)
wbot $a wff _|_ $.
wimp $a wff ( A -> B ) $.
wsq $a wff ( [] A ) $.
wdm $a wff ( <> A ) $.
```

We are now ready to state our axioms. For this we use the `$a` keyword, which is preceded by the axiom's name. The axioms themselves start with the turnstile symbol followed by the conclusions they may entail.

```
$( State the main axioms $)
k $a |- ( ( [] ( A -> B ) ) -> ( ( [] A ) -> ( [] B ) ) ) $.

dual1 $a |- ( ( <> A ) -> ( ( [] ( A -> _|_ ) ) -> _|_ ) ) $.

dual2 $a |- ( ( ( [] ( A -> _|_ ) ) -> _|_ ) -> ( <> A ) ) $.
```

We then define the inference rules. For each rule we create a scope with the `${, $}` pair, the introduce the rules premises with the `$e` keyword (with their labels preceding the keyword), and finally state the rule's conclusion with the `$a` keyword (later we will refer to the rule by the label preceding it).

```
$( Define the modus ponens inference rule $)
${
        min $e |- A $.
        maj $e |- ( A -> B ) $.
        mp  $a |- B $.
$}

$( necessitation introduction rule $)
${
```

```
        f   $e |- A $.
        nec $a |- ( [] A ) $.
$}
```

Now that the system K implementation in Metamath is done, we will show a short proof as an example. Note how we needed the tautology $A \to (B \to A)$, and so we state it as an axiom before the proof. As one can see, reading the raw proof is not very easy; it is actually in reverse polish notation, and **is not meant to be human-readable**. Another complicating factor is that rules and axioms have implicit arguments, for example the rule for *modus ponens* requires not only the two premises but also that they are well-formed formulas. Depending on what formulas A and B be end up being during the rule application, we need to provide their respective proofs of well-formedness. Luckily, this is all done automatically for us, but is recorded explicitly in the Metamath code. Not only the metamath program assists in constructing this proof by providing the implicit arguments of rules and axioms, but it also is capable of displaying the proof in a more readable format, by default using a Lemmon-like proof diagram (see Figure 6.3). Do note that the proof starts at index 21; this is a condensed version of the proof with only the logically-relevant steps. The other steps include the proofs of formula well-formedness we mentioned earlier, and we may of course demand metamath a complete version of the proof if we so desire. The full proof can be seen in Figure 6.4, and if read along the Metamath code it helps us understand it.

```
exhyp $a |- ( A -> ( B -> A ) ) $.

vacuousImplication $p |- ( ( [] p ) -> ( [] ( q -> p ) ) )
         $=
  wp wq wp wimp wimp wsq wp wsq wq wp wimp wsq wimp wp wq wp wimp
    wimp wp wq
  exhyp nec wp wq wp wimp k mp $.



21 exhyp     $a |- ( p -> ( q -> p ) )
22 21 nec    $a |- ( [] ( p -> ( q -> p ) ) )
27 k         $a |- ( ( [] ( p -> ( q -> p ) ) ) -> ( ( [] p ) -> (
    [] ( q -> p ) ) ) )
28 22,27 mp  $a |- ( ( [] p ) -> ( [] ( q -> p ) ) )
```

Figure 6.3: Metamath (condensed) proof of $\Box A \to \Box(B \to A)$

```
 1 wp           $f wff p
 2 wq           $f wff q
 3 wp           $f wff p
 4 2,3 wimp     $a wff ( q -> p )
 5 1,4 wimp     $a wff ( p -> ( q -> p ) )
 6 5 wsq        $a wff ( [] ( p -> ( q -> p ) ) )
 7 wp           $f wff p
 8 7 wsq        $a wff ( [] p )
 9 wq           $f wff q
10 wp           $f wff p
11 9,10 wimp    $a wff ( q -> p )
12 11 wsq       $a wff ( [] ( q -> p ) )
13 8,12 wimp    $a wff ( ( [] p ) -> ( [] ( q -> p ) ) )
14 wp           $f wff p
15 wq           $f wff q
16 wp           $f wff p
17 15,16 wimp   $a wff ( q -> p )
18 14,17 wimp   $a wff ( p -> ( q -> p ) )
19 wp           $f wff p
20 wq           $f wff q
21 19,20 exhyp  $a |- ( p -> ( q -> p ) )
22 18,21 nec    $a |- ( [] ( p -> ( q -> p ) ) )
23 wp           $f wff p
24 wq           $f wff q
25 wp           $f wff p
26 24,25 wimp   $a wff ( q -> p )
27 23,26 k      $a |- ( ( [] ( p -> ( q -> p ) ) ) -> ( ( [] p ) ->
   ( [] ( q -> p ) ) ) )
28 6,13,22,27 mp  $a |- ( ( [] p ) -> ( [] ( q -> p ) ) )
```

Figure 6.4: Metamath full proof of $\Box A \to \Box(B \to A)$

6.1.4 GLF

GLF has two distinct modes of operation: defining logical systems and making proofs using them.

As discussed in Chapter 5, defining a logical system in GLF is done in Haskell using an embedded domain-specific language (eDSL). First we define the operators we will need, giving their symbols and precedence:

```
-- declare logical connectives
implication, bottomOp, necessity, possibility :: Operator
implication = rightAssociativeOp "→" 750
bottomOp = nullaryOp "⊥"
necessity = prefixOp "□" 1000
possibility = prefixOp "◊" 1000
```

`rightAssociativeOp`, `nullaryOp` and `prefixOp` are helper functions to define logical operators, in the same spirit of `binaryOp` from Figure 5.9.

To define a rule one needs to specify the format of the formulas appearing in its premises, conclusions, and discards. This formula specification is a core part of a rule's description, allowing GLF's rule applicator to check that a rule application is valid or not. Here we define a helper function that creates a formula specification for each of the logical connectives, taking as many arguments as their connective takes. The built-in `op` function specifies the main logical connective of a formula, while the `opn` function specifies the format of its nth operand.

```
implies f g = op implication <> opn 1 f <> opn 2 g

bottom = op bottomOp

negationOf spec = spec `implies` bottom

necessarily f = op necessity <> opn 1 f

possibly f = op possibility <> opn 1 f
```

We are now ready to define our rules. A formula specification like `"p" `implies` "q"` means that the required formula must be an implication; the strings "p" and "q" do not require anything of the operands, but giving them names allows the rule applicator to check that other sub-formulas with the same name name match them. In the `modusPonens` rule, we need the major premise to be an implication, but also that the antecedent of this implication match the minor premise, and the consequent must match the rule's conclusion.

```
modusPonens, necessitation, k, dualL, dualR :: DeductiveRule

modusPonens =
  rule "mp"
    [ premise "major" ("p" `implies` "q"),
    , premise "minor" "p"
    , concludes "q"
    ]

necessitation =
  rule "□I"
    [ premise "premise" "p"
    , concludes (necessarily "p")
    ]
```

```
k =
  axiom "K"
    [ concludes (antecedent `implies` consequent) ]
  where
    antecedent = necessarily ("p" `implies` "q")
    consequent = necessarily "p" `implies` necessarily "q"

dualR =
  axiom "DUAL-r"
    [ concludes (antecedent `implies` consequent) ]
  where
    antecedent = possibly "p"
    consequent = negationOf (necessarily (negationOf "p"))

dualL =
  axiom "DUAL-l"
    [ concludes (antecedent `implies` consequent) ]
  where
    antecedent =
      negationOf $
        necessarily (negationOf "p")
    consequent = possibly "p"
```

In this code we use a helper function `axiom` that is akin to the `rule` function, but explicitly marks the rule as being an axiom. As previously explained, in GLF axioms are taken to be rules with no axioms.

Given these rules definitions, we can define the K system, and then create proofs using this system in GLF's web proof assistant (see Figure 6.5). The proof is done by sequentially selecting rules and applying them, with the user needing to specify the premises and the rule's conclusion (whenever the GLF can not determine it uniquely). When the proof is done, it can be exported to Latex, the result of which can be seen in Figure 6.6.

6.1.5 Conclusions

As one can see from the implementations of K in Agda (Section 6.1.1), Isabelle (Section 6.1.2), Metamath (Section 6.1.3) and GLF (Section 6.1.4), all systems are capable of defining this simple system without difficulties, in just a few lines of code. All these definitions are done in programming

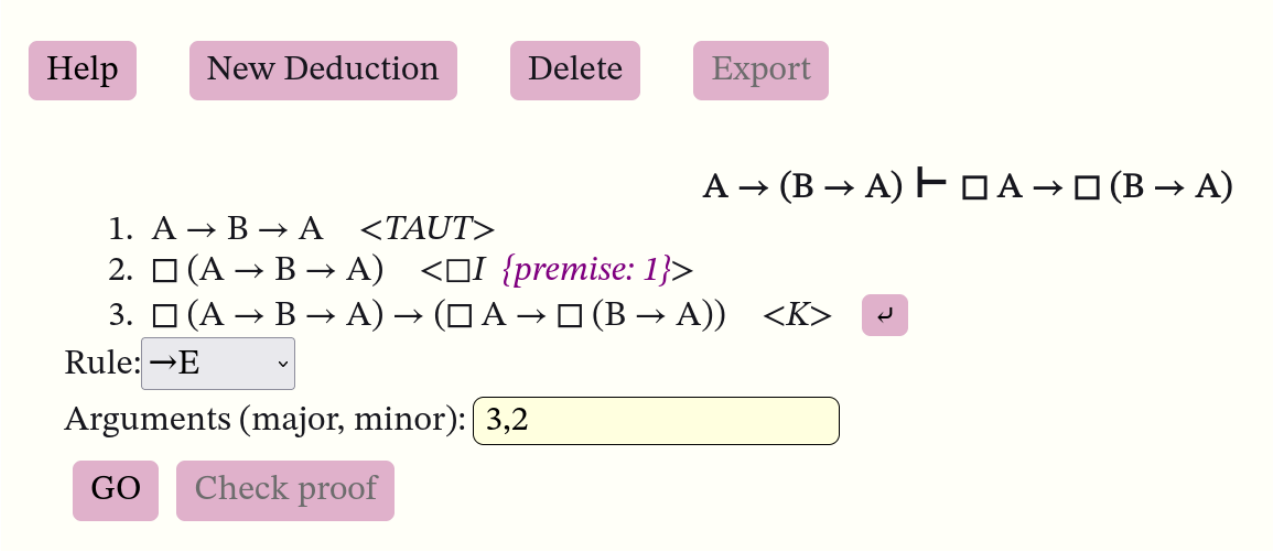

Figure 6.5: On-going proof of $\Box A \to \Box(B \to A)$

1. A → B → A	TAUT
2. □(A → B → A)	□I 1
3. □(A → B → A) → □A → □(B → A)	K
4. □A → □(B → A)	→ E 3 2

Figure 6.6: Proof of $\Box A \to \Box(B \to A)$

language environments,[5] which may not be accessible to many users. For GLF the situation changes when it comes to carrying out proofs in the system: these are not done by programming but by using the system's web interface. For the others, users must still 'program' their proofs. Even though the syntax for these proofs is not too complicated, simply setting up a language environment can be challenging for some users, with many proof assistants proving difficult to install. Metamath and Isabelle are exceptions in this respect, because they make their systems available all in one executable file or archive file, and they don't need external editors that also need setting up (external editors may still be used in both cases).

Metamath in special may be considered difficult to use by some. Despite its simplicity, its syntax limitations can make reading complicated terms very difficult. Displaying large proofs is a challenge in all systems, but specially so in Metamath with its terminal interface. Users may configure LaTeX proof

[5]In the case of GLF defining systems could be made into a standalone domain-specific language, making it more accessible, although a programming language environment would still be needed to define label functions and more complex provisos.

output, but large proofs will often need manual tweaking, making the frequent visualization of an ongoing proof prohibitive. Alas, this limitation is not deal-breaking since Metamath can easily break the proof into smaller sub-goals to be proved instead.

Isabelle and Agda both implement core logics that can then have other logical systems embedded into it, while Metamath and GLF are agnostic, having no built-in logics. This means — among other consequences — that Isabelle and Agda are more readily able to prove meta-properties of the systems they embed, using their built-in logic capabilities. Metamath and GLF on the other hand are only able to prove meta-properties by implementing an additional logical system, which then embeds the original system, with the embedder being able to prove things about the embedded system.

Another consequence of Metamath and GLF's lack of a built-in logical system is that they can not prove the tautologies that are part of the language of the K system. Both of them can assume a tautology is true, but Isabelle and Agda can prove the tautologies themselves using their built-in logical systems. Recall that in Section 6.1.1 we have proved the tautology $A \to (B \to A)$ in Agda; for the Isabelle implementation (Section 6.1.2) we have chosen to assume it was true, but we could have easily proven it. In Metamath or GLF we would have to expand the set of rules of the implemented K system (for the case of $A \to (B \to A)$, we would need the implication introduction rule) to be able to prove such tautologies. The current approach of letting the user assume tautologies risks an unknowing user assuming a non-tautology as being true, jeopardizing the correctness of the system and of its proofs.

Agda, Isabelle, and Metamath have been around for much longer than GLF, and so have accrued many features that have not been discussed here, most of which would not have been useful in the definition of a simple logical system such as this axiomatic one for K. These include automatic reasoning capabilities like proof tactics and the use of solvers (in the cases of Agda and Isabelle) and different user interfaces (Isabelle), even multiple implementations (Metamath).

6.2 Z_p and the proof of De Zolt's postulate in 3D

Here we make a comparison between the implementation of a logical system in the Lean language and in our logical framework. First we motivate the idea of a proof of De Zolt's postulate in three dimensions; then we introduce the system we will implement — Z_p, proposed by Giovannini *et al.* [9] — and show the Lean implementation. This Lean implementation is the basis for a proof of the De Zolt's postulate in three-dimensional space, which we also include. Note that while the three-dimensional version of De Zolt's postulate is not provable in ZFC due to the Banach-Tarski paradox/theorem, it is provable in the weaker type theory Z_p. Finally, we show how Z_p can be implemented in our logical framework, and then we compare both implementations and their advantages.

6.2.1 About De Zolt's postulate

The original version of De Zolt's postulate is for two dimensions. We state below De Zolt's postulate for the two-dimensional case:

Postulate 1 (De Zolt)**.** *Given a polygon $\mathcal{P}$ and a decomposition $T = \{t_1, \ldots, t_k\}$ of $\mathcal{P}$ into k polygons. Let $t_i \in T$, then $T - \{t_i\}$ is not equivalent to T in the theory of equivalence of plane polygons.*

De Zolt establishes common notions concerning equal magnitude (e.g., area of polygons) and congruence between polygons. There is an important discussion on whether one needs to attach a measure, such as the concept of *area*, to figures in order to make a mereological statement. Hartshorne [77] claims that the words 'lesser' or 'greater' in Euclidian geometry should be avoided because these imply the existence of an order relation among figures which has not yet been established. In fact, the existence of an order relation for content depends on the above postulate. He argues that there is no notice of a *purely geometric* proof of the De Zolt postulate from the definition of content area already given. On the other hand, he observes that De Zolt holds whenever a measure of area function is defined in geometry.

We agree with Baldwin [78] that a geometrical proof is one provided in one of these forms: (1) A proof in a formal language for geometry; (2) A proof about i.e., in a meta-theory (e.g. ZFC) with geometry as a defined notion, or (3) Use (2) to get (1). Under this perspective, we can observe that Euclid's proofs in his Elements are not geometrical in Baldwin's sense, because they

were written in natural language. The same reasoning applies to Hilbert's 1899 proof of De Zolt's postulate.

Again following Baldwin [79, 78], a formal proof in geometry should choose: (1) a vocabulary, under some previous conceptual analysis, of the fundamental notions, such as *point*, *line*, *incidence*, etc; (2) a logic, such as first-order logic, second-order logic, $L_{\omega_1\omega}$, dependent or even intuitionistic type theory, for example, and (3) the axioms that reflect the conceptual analysis.

There is much more in discussing Hilbert proof of De Zolt's postulate under the perspective of a formal geometric proof as provided by Baldwin and Hartshore. However, we can say that conceptually De Zolt's original postulate is related to what is known as the scissors congruence (decomposition or dissection) or equidecomposability as opposed to Hilbert's and Euclid's notions that are closer to equicomplementability, equal content or area. To compare statements, Hartshore's version of De Zolt is:

Postulate 2 (Hartshore version of De Zolt)**.** *If Q is a figure contained in another figure P, and if $P - Q$ has a non-empty interior, then P and Q do not have equal content.*

Equal content, in the version of De Zolt above, can be taken as equal area. For Hilbert and other researchers, the original version of De Zolt can be obtained by connecting the mereological and the analytical equivalences with the following theorem:

Theorem 3. *In every model of Euclidian geometry two figures are equimeasured iff they are equicomplementable iff they are equal figures.*

The theorem above holds in Euclidian plane geometry, also known as the Wallace-Bolyai-Gerwien theorem [80, 81][77, §24]. In Giovannini *et al.* [82], we can find a geometrical formal proof of the plane version of De Zolt. It is proven in the original version of De Zolt and does not need the theorem above, nor does it mention any content measure, such as area. It is a mereological formal proof using algebra as a logic for equality and fundamental notions of point, line and polygons in the basic vocabulary.

A natural continuation of the work by Giovannini *et al.* [82] is to have a proof of De Zolt's for the three-dimensional case, and indeed this is pursued in later work by the same authors [9]. A three-dimensional version of De Zolt's postulate is given [9, §5] as follows:

Postulate 3 (Three-dimensional De Zolt)**.** *Given a polyhedron $\mathcal{P}$, a decomposition Δ_p of $\mathcal{P}$, and a truncation Δ_q of Δ_p, then we have that $\Delta_p \prec \Delta_q$.*

The definitions of decomposition, truncation, and $\prec$ are formalized later in this chapter, or see [9].

Due to Dehn's counterexample [83, 84] proving that the regular tetrahedron and the cube with equal volumes are not scissors decomposable,[6] Zolt's postulate does not hold in three-dimensional and higher geometry in a theory as strong as ZFC. We can also mention the so-called ball paradoxes, Hausdorff and Banach-Tarski, that have as a consequence that any two polyhedra, with equal volumes or not, are equidecomposable and hence De Zolt cannot hold either, as both theorems above render it invalid. However, the notion of equidecomposability that is used in the ball paradoxes — which are not in fact paradoxes, but theorems — is too broad to be considered seriously in De Zolt terminology. This point was also made by Giovannini *et al.* [9].

Thus, this chapter provides a mereological and formal–geometrical proof of De Zolt's postulate in the three-dimensional case. We will be formalizing the work of Giovannini *et al.* in proposing the Z_p type system, and filling in the details of their natural language sketch of the proof of De Zolt's postulate in three dimensions (see [9, §5]). The proof is formal not only in the sense of Baldwin's definition of a formal geometrical proof but also due to its execution in the Lean proof assistant. The Lean proof uses only a basic form of recursive definition and a very weak type system, the aforementioned Z_p (Figure 6.7), to provide the fundamental geometric vocabulary. Z_p has no way to define what is geometric content, such as volume or the Dehn invariant; it does not have even the notion of Natural numbers, making our proof mereological in nature. Both the notion of polyhedron decomposition and that of the $\prec$ relation needed by the three-dimensional version of De Zolt's postulate are given by the Z_p system's rules; the definition of a truncation of a polyhedron decomposition is given in Lean in Section 6.2.5.

6.2.2 The Lean prover

The formalization of De Zolt's postulate we carry out in this chapter is done in the Lean language (version 4). In this section we give a brief overview of Lean, before we delve into the details of our proof.

Lean is a pure functional programming language designed to aid formal proofs. The theory behind Lean is type theory, like in other similar tools like Coq [34] and Agda [38]. More specifically, Lean uses a version of the Calculus of Inductive Constructions (CIC) of Coquand and Huet [35].

[6] This counterexample provided the solution to Hilbert's third problem.

Types:

$$\mathfrak{T} : \mathfrak{p}, \mathfrak{s}, \mathfrak{f}, \mathfrak{v}$$

Rules:

$$\frac{n : \mathfrak{p} \quad m : \mathfrak{p} \quad n \neq m}{\langle n, m \rangle : \mathfrak{s}}\,\mathfrak{s}_1 \qquad \frac{p : \mathfrak{s} \quad q : \mathfrak{s} \quad \neg\,\mathrm{Collinear}(p, q) \quad p \cap q : \mathfrak{p}}{\langle p : \mathfrak{s}, q : \mathfrak{s} \rangle : \mathfrak{s}}\,\mathfrak{s}_2$$

$$\frac{p : \mathfrak{s} \quad q : \mathfrak{s} \quad \mathrm{Jordan}(p \cup q)}{p \cup q : \mathfrak{f}}\,\mathfrak{f}_1 \qquad \frac{p : \mathfrak{f} \quad q : \mathfrak{f} \quad p \cap q : \mathfrak{s}}{p \cup q : \mathfrak{f}}\,\mathfrak{f}_2$$

$$\frac{p : \mathfrak{f} \quad q : \mathfrak{f} \quad \mathrm{Closed}(p \cup q)}{p \cup q : \mathfrak{v}}\,\mathfrak{v}_1 \qquad \frac{p : \mathfrak{v} \quad q : \mathfrak{v} \quad p \cap q : \mathfrak{f}}{p \cup q : \mathfrak{v}}\,\mathfrak{v}_2$$

$$\frac{p : \mathfrak{T}}{p \preceq p}\,\varepsilon_0 \qquad \frac{p : \mathfrak{T} \quad q : \mathfrak{T} \quad p \,\mathrm{cmp}\, q \quad q \neq \varepsilon}{p \prec p; q}\,\varepsilon_1$$

$$\frac{p : \mathfrak{T} \quad q : \mathfrak{T} \quad p \,\mathrm{cmp}\, q \quad p \neq \varepsilon}{q \prec p; q}\,\varepsilon_2 \qquad \mathrm{i}{=}1{,}2\,\frac{p_i : \mathfrak{T} \quad q_i : \mathfrak{T} \quad p_i \preceq q_i \quad p_1 \,\mathrm{cmp}\, p_2 \quad q_1 \,\mathrm{cmp}\, q_2}{p_1; p_2 \preceq q_1; q_2}\preceq_1$$

$$\mathrm{i}{=}1{,}2\,\frac{p_i : \mathfrak{T} \quad q_i : \mathfrak{T} \quad p_1 \prec q_1 \quad p_2 \preceq q_2 \quad p_1 \,\mathrm{cmp}\, p_2 \quad q_1 \,\mathrm{cmp}\, q_2}{p_1; p_2 \prec q_1; q_2}\prec_1 \qquad \frac{}{\varepsilon : \mathfrak{T}}\,\varepsilon$$

$$\frac{p_i : \mathfrak{T} \quad q_i : \mathfrak{T} \quad p_1 \preceq q_1 \quad p_2 \prec q_2 \quad p_1 \,\mathrm{cmp}\, p_2 \quad q_1 \,\mathrm{cmp}\, q_2}{p_1; p_2 \prec q_1; q_2}\prec_2$$

$$\frac{\langle n, m \rangle : \mathfrak{s}}{n \;\mathrm{cmp}\; m}\,\mathrm{cmp}_0 \qquad \frac{\langle p : \mathfrak{s}, q : \mathfrak{s} \rangle : \mathfrak{s}}{p \;\mathrm{cmp}\; q}\,\mathrm{cmp}_1$$

$$\frac{p \cup q : \mathfrak{f}}{p \;\mathrm{cmp}\; q}\,\mathrm{cmp}_2 \qquad \frac{p \cup q : \mathfrak{v}}{p \;\mathrm{cmp}\; q}\,\mathrm{cmp}_3$$

Figure 6.7: The type system Z_p for polyhedral mereology

Lean's sprawling mathematics library, mathlib [85] differentiates Lean from most other theorem provers. It contains more than a hundred thousand theorems and about half as many definitions, totalling more than a million lines of code contributed by 300 people. Besides the community-building efforts of mathematicians like Jeremy Avigad, Patrick Massot and Kevin Buzzard, the reason for this success is believed to be Lean's extensibility, which has been increased even more in its fourth version [42] with the addition of hygienic macros [86] inspired by the Racket programming language.

Not only has Lean been at the frontier of machine-checked mathematical proofs (for a sample of recent developments, see [87, 88, 89, 90, 91, 92]), but it has also been a platform for innovations in the field of programming languages, specially functional ones. Lean 4 introduced an improvement over the traditional *do* notation that is used as syntactic sugar for imperative-style programming [93], allowing such features as local mutation, early return, and iteration. None of these features are currently supported by the Haskell language, which introduced the idea. Another contribution to language design inspired by the Lean implementation was a new technique for reference counting in purely functional programming languages [94]. Any reference-counting implementation makes a garbage collector unnecessary, and this one improves on them by reducing the number of reference updates and providing a new memory-reclaiming algorithm for non-shared values.

6.2.3 Geometric objects in Lean

The first step in our formalization is to define the geometrical objects of the Z_p system in which De Zolt's postulate holds. Z_p is a type system with only four types: one for points, another for segments, another for faces, and finally one for volumes. The first set of rules of Z_p (see Figure 6.7) compose well-formed geometrical objects of these types; because Z_p is a very weak theory, it takes geometric concepts like collinearity and Jordan curves as given. These concepts defined outside the system are fundamental to the building of well-formed geometric objects, but are not important for the proof of De Zolt's postulate.

The first and most elementary object we will implement in Lean is the point, which is taken as given. We thus define it as an opaque Lean type (a structure with no fields, henceforth written as `Point`). This means that a `Point` is not explicitly described, and so can't 'look inside' it.

```
structure Point : Type
```

The treatment given to `Point` is exceptional: all the other geometric objects are represented in Lean by their constructions (or deconstructions, depending on how you look at it).

The type of `Segment` has a constructor for the empty `Segment`, another that constructs a `Segment` from two points and a proof that they are different from each other, and one that joins two values of `Segment` into one (`Segment.cons`).

```
inductive Segment : Type where
| empty : Segment
| s₁ : (n m : Point) → n ≠ m → Segment
| cons : Segment → Segment → Segment
```

The `Segment` type is isomorphic to a list of singleton `Segment` values, but we do not use the usual definition for such a type — with only two constructors — so that our formalization is closer to the original definition of the Z_p system. `Segment.empty` is the monomorphic version of the polymorphic ε rule for the `Segment` type. We will see in Section 6.2.4 how we will recover the polymorphism of the ε rule (and other instances of polymorphism) with the use of a Lean type class. `Segment.s₁` of course corresponds to the Z_p s_1 rule, while `Segment.cons` simply joins two `Segment` values into one, regardless of whether they can be composed together validly or not. Because `Segment.cons` does not create well-formed values of `Segment`, it does not correspond to the s_2 rule from Z_p; it is the `Segment.s₂` function that fills this role. `Segment.s₂` is defined below, after we define the necessary predicates that guarantee the well-formedness of a segment built from two other segments.[7] Because we do not need to reason about collinearity and the intersection of two segments for our proof of De Zolt's postulate, we define both predicates as being `opaque`. Also note that while the `Segment.s₂` function discards its arguments related to the well-formedness of the union of two segment values, the Lean type checker still guarantees that when it is called the necessary predicates hold, and so guarantees the correct application of the s_2 rule.

```
opaque Segment.Collinear
  : Segment → Segment → Prop

opaque Segment.HasPointIntersection
  : Segment → Segment → Prop

def Segment.s₂ : (p q : Segment)
```

[7] The reason for defining `Segment.s₂` separately from `Segment.s₁` is that including it in the definition of `Segment` would create a circularity between this type and the collinearity and intersection predicates.

```
    → ¬ (Segment.Collinear p q)
    → Segment.HasPointIntersection p q
    → Segment
| p, q, _notCollinear, _hasPointIntersection => Segment.cons p q
```

Analogous comments to the ones made above about the `Segment` type's constructors, functions, and predicates, and how they pertain to their original incarnations in Z_p also apply to the `Face` and `Volume` types that we will see shortly (these types too are isomorphic to lists).

The next object we will implement is that of geometric faces. A `Face` is isomorphic to a list of singleton `Face` values, and has a constructor for the empty `Face`, another that builds a `Face` from a pair of `Segment` values and a proof that they form a Jordan curve, and finally one that joins two values of type `Face` into one.

```
opaque Segment.IsJordan : Segment → Segment → Prop

inductive Face : Type where
| empty : Face
| f₁ : (p q : Segment)
  → Segment.IsJordan p q
  → Face
| cons : Face → Face → Face

opaque Face.HasSegmentIntersection
  : Face → Face → Prop

def f₂ : (p q : Face)
    → Face.HasSegmentIntersection p q
    → Face
| p, q, _hasSegmentIntersection => Face.cons p q
```

A `Volume` is isomorphic to a list of singleton `Volume` values, and has a constructor for the empty `Volume`, another that creates a `Volume` from a pair of `Face` values and a proof that they form a closed volume, and finally one that joins two values of type `Volume` into one.

```
opaque Face.IsClosed : Face → Face → Prop

inductive Volume where
| empty : Volume
| v₁ : (p q : Face) → Face.IsClosed p q → Volume
| cons : Volume → Volume → Volume
```

```
opaque Volume.HasFaceIntersection
  : Volume → Volume → Prop

axiom Volume.EmptyAlwaysHasFaceIntersection {v : Volume}
  : HasFaceIntersection empty v

axiom Volume.HasFaceIntersection_comm {v u : Volume}
  : HasFaceIntersection v u
  → HasFaceIntersection u v

def v₂ : (p q : Volume)
    → Volume.HasFaceIntersection p q
    → Volume
| p, q, _hasFaceIntersection => Volume.cons p q
```

We also state two facts about the `Volume.HasFaceIntersection` predicate: the empty `Volume` always has a face intersection with any volume, and that the `Volume.HasFaceIntersection` predicate is commutative, that is, if a volume has a face intersection with another volume, then the latter volume also has a face intersection with the former volume.

Here ends the definition of the geometric objects we will need; De Zolt's postulate is defined over the `Volume` type.

6.2.4 The Z_p system in Lean

We can classify the rules of the Z_p system (see Figure 6.7) in two groups: the ones related to the construction of geometrical objects, and the mereological rules.

We have already seen the definition of the geometrical objects of the Z_p system in Lean, and the rules about how they are constructed. These definitions are all monomorphic, however, while the ε rule is inherently polymorphic. The mereological rules we still need to implement are also polymorphic, so we have two implementation options: either we monomorphize them, creating one rule for each geometric object type, or we introduce a mechanism for polymorphism so that we can implement the rules as they are. Because monomorphizing all polymorphic rules would make the number of rules in the system blow up and would also we make the Lean code very repetitive, we opted for the latter alternative.

In Lean, type classes are the preferred mechanism for introducing polymorphism, also serving as a way to overload notation. We thus introduce the $\mathtt{Z}_p$ type class, which is composed of the following definitions: ε is the (poly-

morphic) empty object, `cmp` is a binary predicate stating that two objects are 'compatible', that is, we can use `join` to join them in a well-formed way. Additionally, we also give an infix notation to the `join` function (the same one used in Z_p, the ; symbol), and state that joining a Z_p object with an empty object results in the original object, a fact we will employ latter.

```
class Zₚ (a : Type u) where
  ε : a
  cmp : a → a → Prop
  join : (p : a) → (q : a) → a

infixr:80 ";" => Zₚ.join

axiom Zₚ.empty_right_join {t} [Zₚ t] {p : t}
  : p ; ε = p
```

For a type to become an instance of this type class we must provide type-specific definitions for these notions. To illustrate how this works in practice, below is the instantiation of `Volume` as part of the Z_p type class:

```
instance : Zₚ Volume where
  ε := Volume.empty
  cmp := Volume.HasFaceIntersection
  join := Volume.cons
```

The definition of ε for `Volume` is simply the value given by the `Volume.empty` constructor. `cmp` for `Volume` is defined according to the cmp_3 rule: if we have $p \,\text{cmp}\, q$, then we must have that $p \cap q \;:\; \mathfrak{f}$ (i.e., `Volume.HasFaceIntersection p q`). Finally, `join` (Z_p's ;) for `Volume` is simply Lean's `Volume.cons`. Note again that the user can build invalid geometric objects compositions using either `Volume.cons` or `join` directly; the user should always use the definitions corresponding to the Z_p rules instead, to guarantee the correct constructions.

The instantiations for the other geometrical objects are made similarly (with the appropriate predicates, according to the Z_p system rules), but they are not necessary for the proof of the De Zolt's postulate in Z_p. These instantiations can be consulted in Appendix A, where the full Lean code for the Z_p proof is shown.

Now we are ready to define the mereological rules of Z_p. $\prec$ (in Lean, `le`) and $\preceq$ (in Lean, `leq`) are defined inductively over pairs of Z_p values:

```
mutual
  variable {t} [Zₚ t]
```

```
  inductive Z_p.le : t → t → Prop where
  | ε₀ {p : t} : le p p
  | le₁ : ∀ {p₁ q₁ p₂ q₂ : t}, le p₁ q₁ → le p₂ q₂
        → (pc : cmp p₁ p₂) → (qc : cmp q₁ q₂)
        → le (p₁ ; p₂) (q₁ ; q₂)

  inductive Z_p.lt : t → t → Prop
  | ε₁ : ∀ {p q : t}, (pqc : cmp p q) → lt p (p ; q)
  | ε₂ : ∀ {p q : t}, (pqc : cmp p q) → lt q (p ; q)
  | lt₁ : ∀ {p₁ q₁ p₂ q₂ : t}, lt p₁ q₁ → le p₂ q₂
        → (pc : cmp p₁ p₂) → (qc : cmp q₁ q₂)
        → lt (p₁ ; p₂) (q₁ ; q₂)
  | lt₂ : ∀ {p₁ q₁ p₂ q₂ : t}, le p₁ q₁ → lt p₂ q₂
        → (pc : cmp p₁ p₂) → (qc : cmp q₁ q₂)
        → lt (p₁ ; p₂) (q₁ ; q₂)
end
```

The constructors of the `le` and `lt` types correspond to the Z_p deductive rules of the same name (see Figure 6.7).

With the way we implemented `join` and the cmp predicate (from the ε_1 and ε_2 rules) as part of the Z_p type class, we removed the need for a direct implementation of the cmp_i rules, as their purpose of giving a polymorphic predicate that the union of two geometrical objects is well-formed is served by the definition of `cmp` in the Z_p type class.

6.2.5 The formal proof

The final piece we need to prove De Zolt's postulate is the definition of a truncation of a geometric object (in Lean this means any value of a type which is an instance of the Z_p type class). We define truncation inductively with the following Lean code:

```
section Truncation
  variable {t} [Z_p t]

  inductive Z_p.TruncationOf : t → t → Prop where
  | t₀ {p : t} : p ≠ ε → TruncationOf ε p
  | t₁ {r s v : t} : (rv : cmp r v) → (sv : cmp s v)
    → TruncationOf r s
    → TruncationOf (r ; v) (s ; v)

end Truncation
```

A `TruncationOf` value is thus constructed recursively. The base case is that for any non-empty version of a geometric object, the empty object is a `TruncationOf` it. So for the case of a `Volume`, for any non-empty `Volume` the empty `Volume` is a `TruncationOf` it. For the inductive case, given three geometric objects, the first of which is a `TruncationOf` the second one, we have that the join of the first object with the third object is a `TruncationOf` the second one with the same third object. This is only true provided we can perform both of these joins, that is, that their results are well-formed geometric objects; this is guaranteed by the two `cmp` arguments.

We are finally ready for the statement of De Zolt's postulate:

```
theorem zolt {q p : Volume}
  (isTrunc : Z_p.TruncationOf q p)
  : Z_p.lt q p
```

That is, if p, q are values of type `Volume`, and q is a `TruncationOf` p, we have that $q \prec p$. Note that the statement of De Zolt's postulate for the three-dimensional case (Postulate 3) talks about polyhedron decompositions; in Z_p a polyhedron and its decomposition are the same thing, for the construction of the polyhedron value is given by its (de)composition.

The proof of De Zolt's postulate is by induction on the `TruncationOf` construction: in the base case, we have that q is the empty `Volume` ε, and so we use the ε_2 rule to show that $\varepsilon \prec p$. In the inductive case we have that p and q are actually $u; r$ (`join u r`) and $w; r$ (`join w r`) respectively. Moreover, we have a proof that `TruncationOf w u` holds, with which we recursively invoke Zolt's postulate to obtain $w \prec u$. With this proof and the trivial proof of $r \preceq r$ we can invoke the lt_1 rule to show that $w; r \prec u; r$ (i.e., $q \prec p$) holds. □

Or, in Lean:

```
theorem zolt {q p : Volume}
  (isTrunc : Z_p.TruncationOf q p)
  : Z_p.lt q p :=
  match isTrunc with
  | Z_p.TruncationOf.t_0 _ =>
      have pεcmp : Z_p.cmp p v_0
           := Volume.HasFaceIntersection_comm
                Volume.EmptyAlwaysHasFaceIntersection
      (Eq.subst Z_p.empty_right_join
             <| Z_p.lt.ε_2 pεcmp)
  | Z_p.TruncationOf.t_1 (r := w) (s := u) (v := r) wrcmp urcmp
    wIsTruncOfu =>
    have w_lt_u : Z_p.lt w u := zolt wIsTruncOfu
```

```
have r_le_r : Z_p.le r r := Z_p.le.ε₀
Z_p.lt.lt₁ w_lt_u r_le_r wrcmp urcmp
```

6.2.6 GLF

Our implementation of Z_p in GLF is very different from the Lean one. GLF can more closely emulate the original system, down to its rules (see Figure 6.7 on page 90).

Before implementing the rules, however, we first need to define the operators we will need. We start with those defining the types of the geometric objects, plus the empty object. To encode these types in GLF, we make them into nullary operators (which are also logical constants). We also define auxiliary functions that help write formula specifications including those types. For example, for a premise saying that an object p is a point we write `p .: point`, where the Haskell operator `.:` is the one from Figure 5.9.

```
pointType, segmentType, faceType, volumeType, emptyOp :: Operator
pointType = nullaryOp "𝔭"
segmentType = nullaryOp "𝔰"
faceType = nullaryOp "𝔣"
volumeType = nullaryOp "𝔳"
emptyOp = nullaryOp "ε"

empty, point, segment, face, volume :: FormulaSpec
empty = op emptyOp
point = op pointType
segment = op segmentType
face = op faceType
volume = op volumeType
```

Now we can define the other operators we need and the associate auxiliary helper functions for specifying formulas, including those for intersection and union. The `compatibility, compatible` operators pertain to the cmp operator in the original system definition, which we chose to represent by a ? operator.

```
intersectionOp, unionOp :: Operator
intersectionOp = binaryOp "∩" 800
unionOp = binaryOp "∪" 800

leqOp, leOp :: Operator
leqOp = binaryOp "≼" 600
leOp = binaryOp "≺" 600
```

```
compatibility :: Operator
compatibility = binaryOp "?" 650

inequality :: Operator
inequality = binaryOp "≠" 540

intersection, union :: FormulaSpec ->  FormulaSpec -> FormulaSpec
f `intersection` g = op intersectionOp <> opn 1 f <> opn 2 g
f `union` g = op unionOp <> opn 1 f <> opn 2 g

leq, le :: FormulaSpec -> FormulaSpec -> FormulaSpec
f `leq` g = op leqOp <> opn 1 f <> opn 2 g
f `le` g = op leOp <> opn 1 f <> opn 2 g

compatible :: FormulaSpec -> FormulaSpec -> FormulaSpec
compatible f g = op compatibility <> opn 1 f <> opn 2 g

(/=) :: FormulaSpec -> FormulaSpec -> FormulaSpec
a /= b = op inequality <> opn 1 a <> opn 2 b
```

We also need the following definitions of predicates and constructors:

```
segmentCons :: Operator
segmentCons = naryOp "Segment" 2

twoPointSegment, polySegment :: FormulaSpec -> FormulaSpec ->
   FormulaSpec
twoPointSegment f g = op segmentCons <> opn 1 f <> opn 2 g .: segment
polySegment f g = op segmentCons <> opn 1 f <> opn 2 g .: segment

nonCollinearPred, isClosedPred, isJordanPred :: Operator
nonCollinearPred = naryOp "NonCollinear" 2
isClosedPred = naryOp "IsClosed" 2
isJordanPred = naryOp "IsJordan" 2

nonCollinear :: FormulaSpec -> FormulaSpec -> FormulaSpec
nonCollinear f g = op nonCollinearPred <> opn 1 f <> opn 2 g

isJordan, isClosed :: FormulaSpec -> FormulaSpec
isJordan f = op isJordanPred <> opn 1 f
isClosed f = op isClosedPred <> opn 1 f
```

We can now define all the rules in Z_p:

```
s1, s2, f1, f2, v1, v2 :: DeductiveRule
s1 = rule "s1"
     [ premise "left" ("m" .: point)
     , premise "right" ("n" .: point)
     , premise "inequality" ("m" /= "n")
     , concludes (twoPointSegment "m" "n")
     ]
s2 = rule "s2"
     [ premise "left" ("p" .: segment)
     , premise "right" ("q" .: segment)
     , premise "non-collinearity" (nonCollinear "p" "q")
     , premise "pointIntersection" ("p" `intersection` "q" .: point)
     , concludes (polySegment "p" "q")
     ]
f1 = rule "f1"
     [ premise "left" ("p" .: segment)
     , premise "right" ("q" .: segment)
     , premise "jordanity" (isJordan ("p" `union` "q"))
     , concludes ("p" `union` "q" .: face)
     ]
f2 = rule "f2"
     [ premise "left" ("p" .: face)
     , premise "right" ("q" .: face)
     , premise "segmentIntersection" ("p" `intersection` "q" .:
   segment)
     , concludes ("p" `union` "q" .: face)
     ]
v1 = rule "v1"
     [ premise "left" ("p" .: face)
     , premise "right" ("q" .: face)
     , premise "isClosed" (isClosed ("p" `union` "q"))
     , concludes ("p" `union` "q" .: volume)
     ]
v2 = rule "v2"
     [ premise "left" ("p" .: volume)
     , premise "right" ("q" .: volume)
     , premise "faceIntersection" ("p" `intersection` "q" .: face)
     , concludes ("p" `union` "q" .: volume)
     ]
```

```
emptyRule :: DeductiveRule
emptyRule = rule "ε"
            [ concludes (empty .: "ty")
            , help "The empty object may be considered of any type."
            ]

e0, e1, e2 :: DeductiveRule
e0 = rule "ε0"
     [ premise "object" ("p" .: "ty")
     , concludes ("p" `leq` "p")
     ]
e1 = rule "ε1"
     [ premise "left" ("p" .: "ty")
     , premise "right" ("q" .: "ty")
     , premise "compatibility" ("p" `compatible` "q")
     , premise "compatibility" ("q" /= empty)
     , concludes ("p" `le` "p" `union` "q")
     ]
e2 = rule "ε2"
     [ premise "left" ("p" .: "ty")
     , premise "right" ("q" .: "ty")
     , premise "compatibility" ("p" `compatible` "q")
     , premise "compatibility" ("p" /= empty)
     , concludes ("q" `le` "p" `union` "q")
     ]

leqRule :: DeductiveRule
leqRule = rule "≼ "
          [ premise "left1" ("p1" .: "ty")
          , premise "left2" ("p2" .: "ty")
          , premise "right1" ("q1" .: "ty")
          , premise "right2" ("q2" .: "ty")
          , premise "leq1" ("p1" `leq` "q1")
          , premise "leq2" ("p2" `leq` "q2")
          , premise "compatibilityp" ("p1" `compatible` "p2")
          , premise "compatibilityq" ("q1" `compatible` "q2")
          , concludes ("p1" `union` "p2" `leq` "q1" `union` "q2")
          ]


le1, le2 :: DeductiveRule
```

```
le1 = rule "≺ 1"
      [ premise "left1" ("p1" .: "ty")
      , premise "left2" ("p2" .: "ty")
      , premise "right1" ("q1" .: "ty")
      , premise "right2" ("q2" .: "ty")
      , premise "le" ("p1" `le` "q1")
      , premise "leq" ("p2" `leq` "q2")
      , premise "compatibilityp" ("p1" `compatible` "p2")
      , premise "compatibilityq" ("q1" `compatible` "q2")
      , concludes ("p1" `union` "p2" `le` "q1" `union` "q2")
      ]
le2 = rule "≺ 2"
      [ premise "left1" ("p1" .: "ty")
      , premise "left2" ("p2" .: "ty")
      , premise "right1" ("q1" .: "ty")
      , premise "right2" ("q2" .: "ty")
      , premise "leq" ("p1" `leq` "q1")
      , premise "le" ("p2" `le` "q2")
      , premise "compatibilityp" ("p1" `compatible` "p2")
      , premise "compatibilityq" ("q1" `compatible` "q2")
      , concludes ("p1" `union` "p2" `le` "q1" `union` "q2")
      ]

cmp0, cmp1, cmp2, cmp3 :: DeductiveRule
cmp0 = rule "?0"
       [ premise "segment" (twoPointSegment "p1" "p2")
       ,  concludes ("p1" `compatible` "p2")
       ]

cmp1 = rule "?1"
       [ premise "segment" (polySegment "s1" "s2")
       , concludes ("s1" `compatible` "s2")
       ]
cmp2 = rule "?2"
       [ premise "face" ("p" `union` "q" .: face)
       , concludes ("p" `compatible` "q")
       ]
cmp3 = rule "?3"
       [ premise "volume" ("p" `union` "q" .: volume)
       , concludes ("p" `compatible` "q")
       ]
```

$$\frac{p : \mathfrak{T} \qquad p \neq \varepsilon}{\mathrm{TruncationOf}(\varepsilon, p)}\,\mathrm{t}_0$$

$$\frac{p : \mathfrak{T} \quad q : \mathfrak{T} \quad r : \mathfrak{T} \quad p \,\mathrm{cmp}\, r \quad q \,\mathrm{cmp}\, r \quad \mathrm{TruncationOf}(p, q)}{\mathrm{TruncationOf}(p \cup r, q \cup r)}\,\mathrm{t}_1$$

$$\frac{\begin{array}{cccc} & [w : \mathfrak{T}] & & [w : \mathfrak{T}, u : \mathfrak{T}, r : \mathfrak{T}, w \,\mathrm{cmp}\, r, u \,\mathrm{cmp}\, r, \mathrm{P}(w, u)] \\ & \vdots & & \vdots \\ \mathrm{TruncationOf}(p, q) & \mathrm{P}(w, \varepsilon) & & \mathrm{P}(w \cup r, u \cup r) \end{array}}{\mathrm{P}(p, q)}\,\mathrm{t}_\mathrm{i}$$

Figure 6.8: Additional rules for GLF proof of De Zolt

As one can observe, the definitions above follow the ones from Giovannini *et al.* [9] closely (see Figure 6.7), and indeed one can write Z_p proofs in GLF as one can write them on paper, using these definitions and the web interface.

While in Lean we chose to implement a few data types representing the geometric objects we needed, in GLF there is no such need. Lean forces us to divide our inference rules into groups pertaining to each type/relation,[8] but in GLF as in the original Z_p system, we make no distinction between the kind of rules we have; they are all implemented the same way. An advantage of the GLF definition is that they are polymorphic by definition, unlike the Lean ones which need the Z_p typeclass definitions to be polymorphic (without this polymorphism we would need one version of each polymorphic rule for each geometric object type).

The most significant difference from the Z_p implementation is that GLF is not well-suited to prove De Zolt's postulate; one has implemented Z_p, but one still misses the means to write the proof since it needs concepts from outside the system like the notion of truncation and of induction. As always when this is the case, not all is lost: we can implement another system to embed Z_p in, or somewhat equivalently one can even add the necessary definitions to a modified version of Z_p. The necessary rules for such an attempt would look like the ones in Figure 6.8.

[8]This is not a bad thing *per se*, one might even argue it helps to organize the system and make it more organized; although it might also make it less clear that the different rules are all related.

7
Conclusion

Contributions In Chapter 3 we presented a logical framework for labelled Natural Deduction systems. The framework is composed of rule schemas for introduction and elimination rules, and in Chapter 4 we have showcased seven systems that can be embedded in the framework as a way of showing in practice that the rules schemas are general enough to handle a wide variety of logical systems — including ones with unusual quantifiers. We have also shown in Section 3.2 that systems implemented in GLF preserve their correctness and completeness.

Our logical framework attempts to remedy some of the shortcomings we point out about LFs in general in Section 2.3. Most LFs have intricate meta-languages that create barriers of entry to new users. Most LFs also put up barriers to entry in their user interfaces, with them mostly consisting of programming language environments. As discussed in Section 2.3, it is often the case that formalizations done using a logical framework or proof assistant are different from the original object of formalization, raising questions about whether the formalization is equivalent to the original formulation, and making the formalization less useful/attractive to the proponents of the original object.

Our GLF framework thus employs a simple meta-language, based on the rule schemas of Chapter 3. Although GLF still needs a programming language environment to define new logical systems, once these systems are defined we are able to offer a single user interface equivalent to Fitch-style Natural Deduction proofs. We also attempt to alleviate the programming language requirement by providing the user domain-specific languages that help them define such new logical systems in a more declarative manner, as one can see in Chapter 5. Because of our choice of meta-language and of user interface — which are are natural consequences of our focus on Natural Deduction systems — we are able to formalize logical systems matching their traditional paper definitions very closely.

This proximity to the original formulations of logical systems can be seen in Chapter 6. There we contribute an implementation of a single logical system in four different logical frameworks: Agda, Isabelle, Metamath, and of course

GLF. We also contribute an formalization of the Z_p type system for polyhedral mereology in both Lean and GLF. The Lean prover implementation contains a new formal proof of a three-dimension version of De Zolt's postulate, whose original version is a core tenet of planar geometry but whose generalization to higher dimensions is inconsistent with the axioms of ZFC.

Limitations One challenge GLF faces is rules that are not schematic. One such example is the structural X rule in Ultrafilter logic (see Section 4.2), which by virtue of being structural does not fit into the introduction/elimination rule schemas. Other examples are the CTL rules involving sub-derivations, whose provisos can not be simply encoded into the formulas' labels. As Rentería [5, §5.4.4] notes, the inclusion of the sub-derivations in these rules is not necessary for the correctness and completeness of his original system (they are only added to guarantee some additional nice-to-have properties), and so we may consider a variant labelled system for CTL that removes the sub-derivations and the provisos on them without losing the correctness or completeness properties. Other provisos that are not readily supported by the framework can be included by complicating the system labels. This would be an alternative solution to the CTL sub-derivation provisos, and would also work for the special provisos of CTL* (Section 4.6), or the $\to$E rule of Keisler logic (Section 4.4).

Making a system's labelling scheme more complex has trade-offs, however. This allows us to support more provisos inside the system, but it complicates the system's rules and formulas, making it harder to use, so in our implementation we sometimes we judge it better to have a proviso implemented programatically than to have an overly complicated labelling system. We have not found a non-arbitrary criteria as to where the balance should lie, and so this judgement is done heuristically.

Finally, we must observe that the framework presented here focuses on the definition of logical systems, and its implementation is tailored to prove theorems 'inside' the object language (i.e., the logical system described by the user). Therefore the system is not well-suited to prove meta-properties about the deductive systems — even though one could always implement a better-suited system for that into the framework, and then use this newly-implemented system for this purpose. This limitation is clear in the implementations of the Modal K system and of the Z_p in Chapter 6, and is discussed there.

Another limitation of GLF that is made clear by the implementation of the Modal K system is the lack of a way of checking that the tautologies

inputted by a user are indeed tautologies. In proof assistants like Isabelle or Agda one may prove tautologies outside the K system and use them, but this is only possible in GLF if we extend the K system with more rules. This is similar to how we can not prove De Zolt's postulate using GLF's implementation of Z_p without first extending it, as the proof requires meta-reasoning that in the Lean proof is provided by Lean's built-in logical system.

Future work A natural expansion of the work of this thesis is implementing more logical systems in GLF.

We also plan to provide way of checking GLF proofs independently. The Dedukti framework [95] already has translators that can export proofs written in the HOL, Matita, and Coq (work-in-progress) proof assistants — among others — to the Dedukti language. The idea is thus to export GLF proofs to Dedukti, which will then check them, providing greater assurance of the correctness of the GLF implementation.

Another extension of GLF we plan on is the implementation of other user interfaces. More specifically, we would like to have a Gentzen-style Natural Deduction interface, supporting both backwards (from goal to hypotheses) and forwards (from hypotheses to goal) reasoning.

We also plan work towards adding semi-automatic capabilities to GLF. We have already devised a domain-specific language to describe proof tactics, and need to work out the details of the implementation and of how tactics and their definitions are to be presented to — and defined by — the users.

A welcome addition to GLF would be a way to 'splice' proofs together. This would solve the limitation we have in the implementation of the K Modal system in which we can not prove the classical tautologies that are also valid in K inside the K system: we would invoke the appropriate proofs from a classical propositional system to guarantee that the tautologies are indeed tautologies, i.e., that the user is not making a mistake assuming something that is invalid. When this splicing is done between proofs in the same system, we have an instance of *substitution* [96, p. 18]: if we have two deductions $\Gamma \vdash \varphi$ and $\varphi, \Delta \vdash \psi$, then we can splice the former into the latter to obtain a deduction $\Gamma, \Delta \vdash \psi$. Note that substitution corresponds to the cut rule in sequent calculus systems.

Bibliography

[1] PRAWITZ, D.. **Proofs and the Meaning and Completeness of the Logical Constants**, p. 25–40. Springer Netherlands, Dordrecht, 1979.

[2] GABBAY, D.. **Labelled deductive systems: A position paper**. In: LOGIC COLLOQUIUM, volumen 90, p. 66–88, 1993.

[3] GABBAY, D. M.. **Labelled Deductive Systems**. Oxford University Press, Oxford, 1996.

[4] VIGANÒ, L.. **Labelled non-classical logics**. Springer Science & Business Media, New York, 2000.

[5] RENTERÍA, C. J.. **Uma abordagem geral para quantificadores em dedução natural**. PhD thesis, Departamento de Informática, PUC-Rio, 2004.

[6] CUCONATO, B.; DE BARROS SANTOS, J. ; HAEUSLER, E. H.. **A logical framework with a graph meta-language**. CoRR, abs/2106.13843, 2021. Presented at LFMTP'21.

[7] CUCONATO, B.; DE BARROS SANTOS, J. ; HAEUSLER, E. H.. **A graph logical framework (abstract)**. In: Russo, C.; Suguitani, L. ; Passos, M. D., editors, XX BRAZILIAN LOGIC CONFERENCE BOOK OF ABSTRACTS, p. 92–94. Editora dos Autores, 2022. Presented at EBL'22.

[8] ALKMIM, B.; HAEUSLER, E. ; NALON, C.. **A labelled natural deduction system for an intuitionistic description logic with nominals**. In: DL 2022, 35TH INTERNATIONAL WORKSHOP ON DESCRIPTION LOGICS, Haifa, Israel, 2022.

[9] GIOVANNINI, E. N.; LASSALLE CASANAVE, A. ; HAEUSLER, E. H.. **Sobre el postulado de de zolt en tres dimensiones**. O que nos faz pensar, 29(49), 2021.

[10] VON PLATO, J.. **From axiomatic logic to natural deduction**. Studia Logica, 102(6):1167–1184, jun 2014.

[11] PELLETIER, F. J.; HAZEN, A.. **Natural Deduction Systems in Logic**. In: Zalta, E. N.; Nodelman, U., editors, THE Stanford ENCYCLOPEDIA OF PHILOSOPHY. Metaphysics Research Lab, Stanford University, Spring 2023 edition, 2023.

[12] PELLETIER, F. J.. **A history of natural deduction and elementary logic textbooks**. Logical consequence: Rival approaches, 1:105–138, 2000.

[13] INDRZEJCZAK, A.. **Stanisław Jaśkowski and Natural Deduction Systems**, p. 465–483. Springer International Publishing, Cham, 2018.

[14] PRAWITZ, D.. **Natural deduction: A proof-theoretical study**. Almqvist & Wiksell, Göteborg, 1965.

[15] CHURCH, A.. **Introduction to Mathematical Logic**. Princeton university Press, Princeton, 1956.

[16] DE QUEIROZ, R. J. G. B.; GABBAY, D. M.. **Labelled Natural Deduction**, p. 173–250. Springer Netherlands, Dordrecht, 1999.

[17] SIMPSON, A. K.. **The Proof Theory and Semantics of Intuitionistic Modal Logic**. PhD thesis, University of Edinburgh, College of Science and Engineering, School of Informatics, 1994.

[18] VAN BENTHEM, J.. **Proofs, Labels and Dynamics in Natural Language**, p. 31–41. Springer Netherlands, Dordrecht, 1999.

[19] ANDERSON, A. R.; BELNAP, N. D.. **Entailment: The Logic of Relevance and Neccessity, Vol. I**. Princeton, N.J.: Princeton University Press, 1975.

[20] ANDERSON, A. R.; BELNAP, N. D.. **The pure calculus of entailment**. The Journal of Symbolic Logic, 27(1):19–52, 1962.

[21] PAULSON, L. C.. **Isabelle: A generic theorem prover**, volumen 828. Springer Science & Business Media, 1994.

[22] PFENNING, F.. **The practice of logical frameworks**. In: COLLOQUIUM ON TREES IN ALGEBRA AND PROGRAMMING, p. 119–134. Springer, 1996.

[23] BARWISE, K. J.. **Axioms for abstract model theory**. Annals of Mathematical Logic, 7(2-3):221–265, 1974.

[24] GABBAY, D. M.. **What is a Logical System?**, p. 179–216. Oxford University Press, Inc., USA, 1994.

[25] MESEGUER, J.. **General logic**. In: Ebbinghaus, H.-D.; Fernandez-Prida, J.; Garrido, M.; Lascar, D. ; Artalejo, M. R., editors, LOGIC COLLOQUIUM'87, volumen 129 de **Studies in Logic and the Foundations of Mathematics**, p. 275–329. Elsevier, 1989.

[26] WOLTER, U.; MARTINI, A. ; HAUSLER, E. H.. **Towards a uniform presentation of logical systems by indexed categories and adjoint situations**. Journal of Logic and Computation, 25(1):57–93, Sep 2012.

[27] Huet, G.; Plotkin, G., editors. **Logical frameworks**. Cambridge University Press, Cambridge, 1991.

[28] MEGILL, N. D.; WHEELER, D. A.. **Metamath: A Computer Language for Pure Mathematics**. Lulu Press, Morrisville, North Carolina, 2019. `http://us.metamath.org/downloads/metamath.pdf`.

[29] BRUIJN, DE, N.. **A survey of the project Automath**, p. 141–161. Studies in logic and the foundations of mathematics. North-Holland Publishing Company, Netherlands, 1994.

[30] GEUVERS, H.. **Proof assistants: History, ideas and future**. Sadhana, 34(1):3–25, Feb 2009.

[31] HARPER, R.; HONSELL, F. ; PLOTKIN, G.. **A framework for defining logics**. Journal of the ACM (JACM), 40(1):143–184, 1993.

[32] PFENNING, F.; SCHÜRMANN, C.. **System description: Twelf — a meta-logical framework for deductive systems**. Lecture Notes in Computer Science, p. 202–206, 1999.

[33] BERTOT, Y.. **A short presentation of Coq**. In: Mohamed, O. A.; Muñoz, C. ; Tahar, S., editors, THEOREM PROVING IN HIGHER ORDER LOGICS, p. 12–16, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[34] TEAM, T. C. D.. **The coq proof assistant, version 8.7.0**, Oct. 2017.

[35] COQUAND, T.; HUET, G.. **The calculus of constructions**. Information and Computation, 76:95–120, 1988.

[36] COQUAND, T.; PAULIN, C.. **Inductively defined types**. In: Martin-Löf, P.; Mints, G., editors, COLOG-88, p. 50–66, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.

[37] BRADY, E.. **Idris, a general-purpose dependently typed programming language: Design and implementation**. Journal of functional programming, 23(5):552, 2013.

[38] NORELL, U.. **Towards a practical programming language based on dependent type theory**. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, 9 2007.

[39] MARTIN-LÖF, P.. **An intuitionistic theory of types: Predicative part**. In: Rose, H.; Shepherdson, J., editors, LOGIC COLLOQUIUM '73, volumen 80 de **Studies in Logic and the Foundations of Mathematics**, p. 73–118. Elsevier, 1975.

[40] MARTIN-LÖF, P.; SAMBIN, G.. **Intuitionistic type theory**, volumen 9. Bibliopolis Naples, 1984.

[41] DE MOURA, L.; KONG, S.; AVIGAD, J.; VAN DOORN, F. ; VON RAUMER, J.. **The Lean theorem prover (system description)**. In: INTERNATIONAL CONFERENCE ON AUTOMATED DEDUCTION, p. 378–388. Springer, 2015.

[42] MOURA, L. D.; ULLRICH, S.. **The lean 4 theorem prover and programming language**. In: Platzer, A.; Sutcliffe, G., editors, AUTOMATED DEDUCTION – CADE 28, p. 625–635, Cham, 2021. Springer International Publishing.

[43] BUZZARD, K.; COMMELIN, J. ; MASSOT, P.. **Formalising perfectoid spaces**. Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, Jan 2020.

[44] GONTHIER, G.. **The four colour theorem: Engineering of a formal proof**. Lecture Notes in Computer Science, p. 333–333, 2007.

[45] GOUËZEL, S.; SHCHUR, V.. **A corrected quantitative version of the morse lemma**. Journal of Functional Analysis, 277(4):1258–1268, 2019.

[46] GOWERS, T.. **What makes mathematicians believe unproved mathematical statements?** Annals of Mathematics and Philosophy, 1(1), 2023.

[47] ALLAIS, G.. **Typing with leftovers - a mechanization of intuitionistic multiplicative-additive linear logic**. 2018.

[48] HAEUSLER, E. H.. **Propositional logics complexity and the subformula property**. In: Lago, U. D.; Harmer, R., editors, PROCEEDINGS TENTH INTERNATIONAL WORKSHOP ON DEVELOPMENTS IN COMPUTATIONAL MODELS, DCM 2014, VIENNA, AUSTRIA, 13TH JULY 2014, volumen 179 de **EPTCS**, p. 1–16, 2014.

[49] HAEUSLER, E. H.. **Prova automática de teoremas em dedução natural: uma abordagem abstrata**. PhD thesis, Departamento de Informática, PUC-Rio, 1990.

[50] CHI, W. H.. **Esquemas abstratos para dedução natural, cálculo de sequentes e lambda cálculo tipificado**. Master's thesis, Departamento de Informática — PUC-Rio, 1991.

[51] SCHROEDER-HEISTER, P.. **A natural extension of natural deduction**. The Journal of Symbolic Logic, 49(4):1284–1300, 1984.

[52] FERREIRÓS, J.. **The road to modern logic—an interpretation**. Bulletin of Symbolic Logic, 7(4):441–484, 2001.

[53] ENDERTON, H.. **A Mathematical Introduction to Logic**. Elsevier Science, 2001.

[54] VELOSO, P. A. S.. **On a logic for 'almost all' and 'generic' reasoning**. Manuscrito: Revista Internacional de Filosofia, 25(1):191–271, 2002.

[55] GOLDBLATT, R.. **Lectures on the hyperreals: an introduction to nonstandard analysis**, volumen 188. Springer Science & Business Media, New York, 1998.

[56] VELOSO, P. A. S.; VELOSO, S. R. M.. **On 'Most' and 'Representative': Filter Logic and Special Predicates**. Logic Journal of the IGPL, 13(6):717–728, 11 2005.

[57] MOSTOWSKI, A.. **On a generalization of quantifiers**. Fundamenta Mathematicae, 44(1):12–36, 1957.

[58] KEISLER, H. J.. **Logic with the quantifier "there exist uncountably many"**. Annals of Mathematical Logic, 1(1):1–93, 1970.

[59] CLARKE, E. M.; EMERSON, E. A.. **Design and synthesis of synchronization skeletons using branching time temporal logic**. In: WORKSHOP ON LOGIC OF PROGRAMS, p. 52–71. Springer, 1981.

[60] EMERSON, E. A.; HALPERN, J. Y.. **"sometimes" and "not never" revisited: On branching versus linear time (preliminary report)**. In: PROCEEDINGS OF THE 10TH ACM SIGACT-SIGPLAN SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES, POPL '83, p. 127–140, New York, NY, USA, 1983. Association for Computing Machinery.

[61] REYNOLDS, M.. **An axiomatization of full computation tree logic**. The Journal of Symbolic Logic, 66(3):1011–1057, 2001.

[62] HAEUSLER, E. H.; DE PAIVA, V. ; RADEMAKER, A.. **Intuitionistic logic and legal ontologies**. In: PROCEEDINGS OF THE 2010 CONFERENCE ON LEGAL KNOWLEDGE AND INFORMATION SYSTEMS: JURIX 2010, JURIX 2010, p. 155–158, NLD, 2010. IOS Press.

[63] HAEUSLER, E. H.; RADEMAKER, A.. **On how kelsenian jurisprudence and intuitionistic logic help to avoid contrary-to-duty paradoxes in legal ontologies**. In: Almeida, E.; Costa-Leite, A. ; Freire, R., editors, SEMINáRIO LóGICA NO AVIãO, Brasília, 2019.

[64] DE ALKMIM, B. P.. **Law and Order(ing): Providing a Natural Deduction System and Non-monotonic Reasoning to an Intuitionistic Descripotion Logic**. PhD thesis, Departamento de Informática — PUC-Rio, 2023.

[65] PLOTKIN, G.; STIRLING, C.. **A framework for intuitionistic modal logics: Extended abstract**. In: PROCEEDINGS OF THE 1986 CONFERENCE ON THEORETICAL ASPECTS OF REASONING ABOUT KNOWLEDGE, TARK '86, p. 399–406, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc.

[66] TURING, A. M.. **On computable numbers, with an application to the entscheidungsproblem**. Proceedings of the London Math. Society, s2-42(1):230–265, 1937.

[67] THERY, L.; BERTOT, Y. ; KAHN, G.. **Real theorem provers deserve real user-interfaces**. ACM SIGSOFT Software Engineering Notes, 17(5):120–129, 1992.

[68] KALISZYK, C.. **Web interfaces for proof assistants**. Electronic Notes in Theoretical Computer Science, 174(2):49–61, 2007. Proceedings of the 7th Workshop on User Interfaces for Theorem Provers (UITP 2006).

[69] SANTOS, J. D. B.. **Infraestrutura para provadores interativos de teoremas na web**. Master's thesis, Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio), 2010. in Portuguese.

[70] BERTOT, Y.; KAHN, G. ; THERY, L.. **Proof by pointing**. Lecture Notes in Computer Science, 789:141–160, 1994.

[71] LEACH-KROUSE, G.. **Carnap: An open framework for formal reasoning in the browser**. In: Quaresma, P.; Neuper, W., editors, PROCEEDINGS 6TH INTERNATIONAL WORKSHOP ON THEOREM PROVING COMPONENTS FOR EDUCATIONAL SOFTWARE, GOTHENBURG, SWEDEN, 6 AUG 2017, volumen 267 de **Electronic Proceedings in Theoretical Computer Science**, p. 70–88. Open Publishing Association, 2018.

[72] VASCONCELOS, D.; PAULA, R. ; MENEZES, M.. **Nadia - natural deduction proof assistant**. In: ANAIS DO XXX WORKSHOP SOBRE EDUCAçãO EM COMPUTAçãO, p. 427–438, Porto Alegre, RS, Brasil, 2022. SBC.

[73] BREITNER, J.. **Visual theorem proving with the incredible proof machine**. In: Blanchette, J. C.; Merz, S., editors, INTERACTIVE THEOREM PROVING, p. 123–139, Cham, 2016. Springer International Publishing.

[74] ZACH, R.. **Boxes and Diamonds: An Open Introduction to Modal Logic**. Independently Published, 2019.

[75] WENZEL, M.. **The Isabelle/Isar Reference Manual**, 2022.

[76] WENZEL, M. M.. **Isabelle/Isar — a versatile environment for human-readable formal proof documents**. PhD thesis, Fakultät für Informatik der Technischen Universität München, 2002.

[77] HARTSHORNE, R.. **Geometry: Euclid and Beyond**. Springer-Verlag, 2000.

[78] BALDWIN, J.. **What is a geometric proof: Reflections on de zolt's axiom**. In: PROCEEDINGS OF THE 17TH CLMPST, 2023.

[79] BALDWIN, J. T.. **Model Theory and the Philosophy of Mathematical Practice: Formalization without Foundationalism**. Cambridge University Press, 2018.

[80] WALLACE, W.; LOWRY, J.. **Question 269**, volumen 3 de **New Series of the Mathematical Repository**. W. Glendinning, London, 1814.

[81] GERWIEN, P.. **Zerschneidung jeder beliebigen anzahl von gleichen geradlinigen figuren in dieselben stücke.** Journal für die reine und angewandte Mathematik (Crelles Journal), 1833(10):228–234, Jan. 1833.

[82] GIOVANNINI, E. N.; HAEUSLER, E. H.; LASSALLE-CASANAVE, A. ; VELOSO, P. A. S.. **De Zolt's Postulate: An Abstract Approach**. The Review of Symbolic Logic, 15(1):197–224, Sept. 2019.

[83] DEHN, M.. **Ueber raumgleiche polyeder**. Nachrichten von der Gesellschaft der Wissenschaften zu Göttingen, Mathematisch-Physikalische Klasse, 1900:345–354, 1900.

[84] DEHN, M.. **Ueber den rauminhalt**. Mathematische Annalen, 55(3):465–478, Sept. 1901.

[85] MATHLIB COMMUNITY, T.. **The lean mathematical library**. In: PROCEEDINGS OF THE 9TH ACM SIGPLAN INTERNATIONAL CONFERENCE ON CERTIFIED PROGRAMS AND PROOFS, CPP 2020, p. 367–381, New York, NY, USA, 2020. Association for Computing Machinery.

[86] ULLRICH, S.; DE MOURA, L.. **Beyond notations: Hygienic macro expansion for theorem proving languages**. CoRR, abs/2001.10490, 2020.

[87] MEHTA, B.. **Formalising sharkovsky's theorem (proof pearl)**. In: PROCEEDINGS OF THE 12TH ACM SIGPLAN INTERNATIONAL CONFERENCE ON CERTIFIED PROGRAMS AND PROOFS, CPP 2023, p. 267–274, New York, NY, USA, 2023. Association for Computing Machinery.

[88] DE FRUTOS-FERNÁNDEZ, M. I.. **Formalizing the Ring of Adèles of a Global Field**. In: Andronick, J.; de Moura, L., editors, 13TH INTERNATIONAL CONFERENCE ON INTERACTIVE THEOREM PROVING (ITP 2022), volumen 237 de **Leibniz International Proceedings in Informatics (LIPIcs)**, p. 14:1–14:18, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[89] DILLIES, Y.; MEHTA, B.. **Formalising Szemerédi's Regularity Lemma in Lean**. In: Andronick, J.; de Moura, L., editors, 13TH INTERNATIONAL CONFERENCE ON INTERACTIVE THEOREM PROVING (ITP 2022), volumen 237 de **Leibniz International Proceedings in Informatics (LIPIcs)**, p. 9:1–9:19, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[90] BAANEN, A.; BEST, A. J.; COPPOLA, N. ; DAHMEN, S. R.. **Formalized class group computations and integral points on mordell elliptic curves**. In: PROCEEDINGS OF THE 12TH ACM SIGPLAN INTERNATIONAL CONFERENCE ON CERTIFIED PROGRAMS AND PROOFS, CPP 2023, p. 47–62, New York, NY, USA, 2023. Association for Computing Machinery.

[91] CLUNE, J.. **A formalized reduction of keller's conjecture**. In: PROCEEDINGS OF THE 12TH ACM SIGPLAN INTERNATIONAL CONFERENCE ON CERTIFIED PROGRAMS AND PROOFS, CPP 2023, p. 90–101, New York, NY, USA, 2023. Association for Computing Machinery.

[92] VAN DOORN, F.; MASSOT, P. ; NASH, O.. **Formalising the h-principle and sphere eversion**. In: PROCEEDINGS OF THE 12TH ACM SIGPLAN INTERNATIONAL CONFERENCE ON CERTIFIED PROGRAMS AND PROOFS, CPP 2023, p. 121–134, New York, NY, USA, 2023. Association for Computing Machinery.

[93] ULLRICH, S.; DE MOURA, L.. **'do' unchained: Embracing local imperativity in a purely functional language (functional pearl)**. Proc. ACM Program. Lang., 6(ICFP), 8 2022.

[94] ULLRICH, S.; DE MOURA, L.. **Counting immutable beans: Reference counting optimized for purely functional programming [in press]**. In: 31ST SYMPOSIUM ON IMPLEMENTATION AND APPLICATION OF FUNCTIONAL LANGUAGES, 2019.

[95] ASSAF, A.; BUREL, G.; CAUDERLIER, R.; DELAHAYE, D.; DOWEK, G.; DUBOIS, C.; GILBERT, F.; HALMAGRAND, P.; HERMANT, O. ; SAILLARD, R.. **Dedukti: a logical framework based on the $\lambda\pi$-calculus modulo theory**, 2016. Manuscript available at `http://www.lsv.fr/~dowek/Publi/expressing.pdf`.

[96] NEGRI, S.; VON PLATO, J. ; RANTA, A.. **Structural Proof Theory**. Cambridge University Press, 2001.

A
Lean code for De Zolt's postulate proof

```
class Z_p (a : Type u) where
  ε : a
  cmp : a → a → Prop
  join : (p : a) → (q : a) → a

infixr:80 ";" => Z_p.join

axiom Z_p.empty_left_join {t} [Z_p t] {p : t}
  : ε ; p = p
axiom Z_p.empty_right_join {t} [Z_p t] {p : t}
  : p ; ε = p

structure Point : Type

inductive Segment : Type where
| empty : Segment
| s₁ : (n m : Point) → n ≠ m → Segment
| cons : Segment → Segment → Segment

opaque Segment.IsCollinear
  : Segment → Segment → Prop

opaque Segment.HasPointIntersection
  : Segment → Segment → Prop

def Segment.s₂ : (p q : Segment)
    → ¬ (Segment.IsCollinear p q)
    → Segment.HasPointIntersection p q
    → Segment
| p, q, _notCollinear, _hasPointIntersection => Segment.cons p q

instance : Z_p Segment where
  ε := Segment.empty
  cmp := λ p q =>    ¬ (Segment.IsCollinear p q)
```

```
                ∧ Segment.HasPointIntersection p q
  join := Segment.cons

opaque Segment.IsJordan : Segment → Segment → Prop

inductive Face : Type where
| empty : Face
| f₁ : (p q : Segment)
  → Segment.IsJordan p q
  → Face
| cons : Face → Face → Face

opaque Face.HasSegmentIntersection
  : Face → Face → Prop

def f₂ : (p q : Face)
    → Face.HasSegmentIntersection p q
    → Face
| p, q, _hasSegmentIntersection => Face.cons p q

instance : Zₚ Face where
  ε := Face.empty
  cmp := Face.HasSegmentIntersection
  join := Face.cons

opaque Face.IsClosed : Face → Face → Prop

inductive Volume where
| empty : Volume
| v₁ : (p q : Face) → Face.IsClosed p q → Volume
| cons : Volume → Volume → Volume

opaque Volume.HasFaceIntersection
  : Volume → Volume → Prop

axiom Volume.EmptyAlwaysHasFaceIntersection {v : Volume}
  : HasFaceIntersection empty v

axiom Volume.HasFaceIntersection_comm {v u : Volume}
  : HasFaceIntersection v u
  → HasFaceIntersection u v
```

```
def v₂ : (p q : Volume)
    → Volume.HasFaceIntersection p q
    → Volume
| p, q, _hasFaceIntersection => Volume.cons p q

instance : Zₚ Volume where
  ε := Volume.empty
  cmp := Volume.HasFaceIntersection
  join := Volume.cons

mutual
  variable {t} [Zₚ t]

  inductive Zₚ.le : t → t → Prop where
  | ε₀ {p : t} : le p p
  | le₁ : ∀ {p₁ q₁ p₂ q₂ : t}, le p₁ q₁ → le p₂ q₂
        → (pc : cmp p₁ p₂) → (qc : cmp q₁ q₂)
        → le (p₁ ; p₂) (q₁ ; q₂)

  inductive Zₚ.lt : t → t → Prop
  | ε₁ : ∀ {p q : t}, (pqc : cmp p q) → lt p (p ; q)
  | ε₂ : ∀ {p q : t}, (pqc : cmp p q) → lt q (p ; q)
  | lt₁ : ∀ {p₁ q₁ p₂ q₂ : t}, lt p₁ q₁ → le p₂ q₂
        → (pc : cmp p₁ p₂) → (qc : cmp q₁ q₂)
        → lt (p₁ ; p₂) (q₁ ; q₂)
  | lt₂ : ∀ {p₁ q₁ p₂ q₂ : t}, le p₁ q₁ → lt p₂ q₂
        → (pc : cmp p₁ p₂) → (qc : cmp q₁ q₂)
        → lt (p₁ ; p₂) (q₁ ; q₂)
end

section Truncation
  variable {t} [Zₚ t]

  inductive Zₚ.TruncationOf : t → t → Prop where
  | t₀ {p : t} : p ≠ ε → TruncationOf ε p
  | t₁ {r s v : t} : (rv : cmp r v) → (sv : cmp s v)
    → TruncationOf r s
    → TruncationOf (r ; v) (s ; v)

end Truncation
```

```
open Z_p

theorem zolt {q p : Volume}
  (isTrunc : TruncationOf q p)
  : lt q p :=
  match isTrunc with
  | TruncationOf.t₀ _ =>
      have pεcmp : cmp p ε
           := Volume.HasFaceIntersection_comm
                Volume.EmptyAlwaysHasFaceIntersection
      (Eq.subst empty_right_join
             <| lt.ε₂ pεcmp)
  | TruncationOf.t₁ (r := w) (s := u) (v := r) wrcmp urcmp
    wIsTruncOfu =>
    have w_lt_u : lt w u := zolt wIsTruncOfu
    have r_le_r : le r r := le.ε₀
    lt.lt₁ w_lt_u r_le_r wrcmp urcmp
```