A Beginner's Guide to Logical Relations for a Logical Framework

Navid Roux
FAU Erlangen-Nuremberg, Germany

Abstract

Logical relations are an established proof technique for deriving meta-level theorems of formal systems. For example, they have been used to prove strong normalization, type safety, and correctness of compiler optimizations in the setting of various type theories and lambda calculi. Theories of logical relations have been stated for a wide range of formal systems.

To formalize formal systems themselves, logical frameworks have been sucessfully used, particularly shining when specifying syntax and deduction (e.g. using judgements-as-types and higher-order abstract syntax). And module systems over logical frameworks allow to modularly identify, translate, and combine full hierarchies of formal systems.

How to represent logical relations in such systems is an ongoing research question. In "Logical Relations for a Logical Framework" (*ACM Transactions on Computational Logic (2013)*), Rabe and Sojakova present a theory of logical relations applicable in the setting of logical frameworks that, when instantiated for formal systems formalized therein, gives a reasonable notion of logical relations within those systems.

We provide a detailed exposition of a special case, accessible to everyone familiar with basics of formalizing in logical frameworks. Our exposition is embedded in a coherent narrative spanning from concretely motivating logical relations (by walking through a proof of strong normalization for the simply-typed lambda calculus) up to concretely modeling a corresponding language feature for the Mmt module system over the Edinburgh Logical Framework. The mechanics of our derived language feature are novel and forgo the need of introducing any logical relation-specific language primitive.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Type theory

Keywords and phrases logical relations,logical framework,module system,representing meta theorems,mmt

Supplementary Material A corresponding talk, slides, and other material can be found at https://gl.kwarc.info/supervision/seminar/-/tree/master/WS2021/logrels.

Acknowledgements I am thankful *i*) to my advisor Florian Rabe for sparking my interest and steadily deepening my knowledge on logical relations, and *ii*) to Paolo G. Giarrusso and Miëtek Bak for discussing several questions of mine in the `##dependent` IRC channel (on dependent type theory) on Freenode.

Contents

Submitted 2021-03-22 as a seminar paper for the kwarc seminar. Last updated 2021-03-22.

1 Introduction

The use of type theories has become pervasive in building foundations of formal systems, proof assistants, and type systems of programming lanaguages. Formalizing type theories in a deduction system themselves allows us to identify, translate, and combine type theories (e.g. via pushouts) in the same way as for logics [Rab17]. Apart from mechanically verifying existing type theories this way, we can also use formalizations "as data" to rapidly prototype full-blown formal systems [MR19].

Almost always, the usefulness of a type theory stands and falls with the meta theorems it admits. This makes the representation and mechanical verification of meta theorems for formalized type theories an ongoing research topic. For example, verification for type theories on which proof assistants are built can constitute one step to increase trust in them. (Even more so if we can extract executable type checkers from such formalizations [Soz+19].) In particular, settling a good way to represent meta theorems is crucial to reason about how these behave under meta operations offered by the deduction system (e.g. translation, combination). For instance, the deduction system might guarantee that the combination of two type theories admits a meta theorem X if both admitted X individually (for X ranging over some class of meta theorems).

In this paper, we focus on the task of representing one specific class of meta theorems, which we motivate in the following. Consider System F, a minimalistic type theory that extends the simply-typed lambda calculus with type polymorphism. Its syntax is given by:

$$\begin{array}{lcll} t & ::= & x \mid \lambda x.\ t \mid s\ t & \text{terms ("programs")} \\ T & ::= & \alpha \mid \forall \alpha.\ T \mid T_1 \to T_2 & \text{types} \end{array}$$

It admits these famous meta theorems:

- *Strong Normalization:* every well-typed program terminates (in the sense of β-reduction)
- *Type Safety*: well-typed program only ever reduce to programs that are well-typed, too
- *Theorems for Free* [Wad89]: e.g. all well-typed programs of type $\forall \alpha.\ \alpha \to \alpha$ are equal to the identity program $\lambda x.\ x$ (for some reasonable notion of equality)

We can consider the lambda calculus and its many variants as archetypes of the core of every functional programming language. In this light, these theorems have real-world significance. The first two theorems imply that any well-typed program terminates with a well-typed value, i.e. "runs to completion without exceptions." And the third theorem guarantees that the semantics of "generic" programs is independent from the type for which they are instantiated; they cannot inspect the type at runtime.

The theorems above are instances of a common pattern for meta-theoretical properties in type theories: they all assert something about well-typed programs only, and, as it turns out, for proving them it is useful to (re)phrase them in the form "if a program t types against T, then $P_T(t)$ holds." Here, $P_T(\cdot)$ is a predicate (relation) on terms of type T for every type T. Corresponding proofs of theorems phrased in such a way typically perform induction on the typing derivation of the well-typedness assumption. The technique of formulating and proving statements in this form is known as *proof by logical relation* or *Tait's method*. Many type theories admit this technique in one way or another.

Our goal is to increase tool support for representing meta theorems and corresponding proofs of the form above, ideally in a way that is generic in the type theory. This means users formalizing type theories should be spared from tediously formalizing a theory of logical relations (in their conrete setting) and instead should be able to derive such a theory

by instantiating a generic notion provided for by the deduction system. In particular, we are interested in solutions that are *i*) stated in a framework that is as general as possible (uncommitting to any logic), and are *ii*) nicely behaving with modular (non-monolithic) formalizations of formal systems. For these reasons, we focus on the task of representing logical relations in module systems based on logical frameworks such as Mmt [RK13] and Twelf [PS]. These combine the following traits that make them especially suitable for our purposes:

- usage of comparatively *weak meta logics* such as the Edinburgh Logical Framework LF [HHP93]
- formalization of knowledge (e.g. syntax) *modularly* and *declaratively* into theories (signatures)
- *open-world assumption*

A weak meta logic ensures we are committing only to a minimal set of assumptions, thus can faithfully represent a broad range of formal systems including many variants of logics, type theories, and set theories. And together with the modular and open-world nature of formalizations, this lets us achieve highly modular formalization hierarchies of formal systems [Cod+; Rab17].

On the other hand, these traits collectively make representation of meta theorems a challenging task. For example, since we do not represent the syntax of a type theory as an inductive type (which would be standard in proof assistants), it is unclear how to state meta theorems quantifying over terms.

Driving Questions Is there a sweet spot between the general, modular, and declarative approaches of LF-inspired module systems and the usual inductive types-inspired approaches of proof assistants? Can we extend the former with an idiomatic way to encode proofs by logical relation?

Contributions Our main contribution is that we provide a coherent composition and narrative spanning from very concretely motivating logical relations to very concretely modeling them in a declarative and modular manner in Mmt; in a way that is accessible to students who are familiar with Mmt- or LF-style formalizations. Concretely:

- We introduce and motivate logical relations by carefully walking through a proof of strong normalization of the simply-typed lambda calculus (Section 2.1).
- We distill the lessons learned into a pattern of how proofs by logical relation look like in general (Section 2.2).
- We recap Mmt/LF, a module system over LF, with a focus on driving themes such as generality and modularity in formalizations, thereby putting our goal into perspective (Section 3).
- We review the approach of Rabe and Sojakova [RS13] for modeling proofs by logical relation for a logical framework; special-cased to Mmt/LF with an emphasis on examples first (Sections 4.1 and 4.2) followed by formal definitions (Section 4.3) and a short evaluation (Section 4.4).

In the last point, we employ a novel presentation of the material which avoids the need to introduce a language primitive in Mmt that is specific to logical relations. The idea was provided by Florian Rabe inspired by recent joint work on a framework for metaprogramming facilities in Mmt [RR20], and implemented for [RR21]. To the best of our knowledge, this paper is the first exposition of this idea.

Related Work To the best of our knowledge, there is neither a previous treatment of logical relations in Mmt (apart from the orignal source [RS13]) nor a longer coherent and student-accessible narrative. We refer to [RS13] for related work on the representation of logical relations. We indicate sources of the materials in the respective sections.

2 Logical Relations

Logical relations are an informal class of proof methods commonly used to prove meta theorems of type theories. By informal we mean that the concrete incarnation of the concept "logical relation" may differ from type theory to type theory and also from proof to proof. Below is a non-exhaustive compilation of meta-theoretic properties that are usually shown via proofs by logical relation [Ahm13].

- Strong normalization: There is no infinite path $t_1 \rightsquigarrow t_2 \rightsquigarrow \dots$ where $\rightsquigarrow$ is some reduction relation on terms.
- Type safety: If a term t has type T and $t \rightsquigarrow t'$, then t' also has type T.
- Program equivalences
  - Correctness of programs and compiler optimizations
  - Parametricity and free theorems [Wad89]: in type theories admitting such "free theorems," we can generate theorems that terms have to obey by just looking at their type. For example, in System F all terms t of type $\forall\alpha.\ \alpha \to \alpha$ obey: for all terms $f\colon A \to A'$, we have $f \circ t_A = t_{A'} \circ f$, where $\circ$ is function composition. With a small bit of more work, we can show that all such t are equal to $\lambda x.\ x$ for some reasonable notion of equality [Pfe18].
  - Representation independence [Mit86]: overlapping with the previous point, programs should not depend on specific implementations (or terms) of abstract datatypes (or types). Instead, programs with either implementation should be observationally equivalent.
  - Information flow control [RG18]: type theories have been proposed that allow to label the type of variables (e.g. with *public* and *secret*) with the idea to restrict certain information flow. For example, to lessen certain *side channel attacks* (of cryptographic programs), we might forbid outputs labeled as public to "noticeably" depend on inputs labeled as secret.

Note that the concept of logical relations is not limited to type theory. We refer to [HRR14] for a modern take on logical relations as applied on broad fields of mathematics and computer science.

In Section 2.1, following the exposition of [Zil12][1], we consider strong normalization of the simply-typed lambda calculus (STLC) as one concrete example to motivate the necessity of logical relations and to also flesh out some light technical details lest logical relations remain an abstract concept for the reader. Readers not interested in such a detailed exposition may safely skip to Section 2.2, where we summarize logical relations to an extent necessary to understand later sections.

[1] Further explanations have been incorporated from [Ahm13], for which typeset notes can be found at [Sko19].

2.1 Strong Normalization of STLC

We first quickly review the syntax and semantics of the variant of STLC that we chose for our purposes.

▶ **Definition 1** (Lambda Calculus). *The **terms** and **substitutions** of the lambda calculus are given by*

$$\begin{array}{lcll} t & ::= & x \mid \lambda x.\ t \mid s\ t & \textit{terms} \\ \gamma & ::= & \cdot \mid \gamma, x \mapsto t & \textit{substitutions} \end{array}$$

We employ (but not define for brevity) the usual capture-avoiding substitution application and write $t\gamma$ for the substitution γ applied on t.

*The **operational semantics** is given by*

$$\frac{}{(\lambda x.\ s)\ t \rightsquigarrow s[x \mapsto t]}\ (\textit{OP-BETA}) \qquad \frac{t \rightsquigarrow t'}{\lambda x.\ t \rightsquigarrow \lambda x.\ t'}\ (\textit{OP-ABS})$$

$$\frac{s \rightsquigarrow s'}{s\ t \rightsquigarrow s'\ t}\ (\textit{OP-APP1}) \qquad \frac{t \rightsquigarrow t'}{s\ t \rightsquigarrow s\ t'}\ (\textit{OP-APP2})$$

Let $\mathcal{SN} = \{t \mid \nexists \text{ infinite path } t = t_1 \rightsquigarrow t_2 \rightsquigarrow \ldots\}$ be the set of strongly normalizing terms.

▶ **Definition 2** (Simply-Typed Lambda Calculus). ***Types** and **contexts** are given by*

$$\begin{array}{lcll} T & ::= & B \mid T_1 \to T_2 & \textit{types} \\ \Gamma & ::= & \cdot \mid \Gamma, x\colon T & \textit{contexts} \end{array}$$

*where B is an unspecified set of base types. We write $\Gamma \vdash t\colon T$ for the **typing judgement** relating terms, types, and contexts defined by the rules*

$$\frac{(x\colon T) \in \Gamma}{\Gamma \vdash x\colon T}\ (\textit{TP-VAR}) \qquad \frac{\Gamma, x\colon T_1 \vdash s\colon T_2}{\Gamma \vdash \lambda x.\ s\colon T_1 \to T_2}\ (\textit{TP-ABS})$$

$$\frac{\Gamma \vdash s\colon T_1 \to T_2 \qquad \Gamma \vdash t\colon T_1}{\Gamma \vdash s\ t\colon T_2}\ (\textit{TP-APP})$$

*Given a typing judgement, we call the proofs of it (there may be arbitrarily many, incl. none) the possible **typing derivations** of the typing judgement.*

We overload the notation above and write $\vdash \gamma\colon \Gamma$ if for all $(x\colon T) \in \Gamma$ the substitution γ is defined at x and $\Gamma \vdash x\gamma\colon T$.

Above and in the following, we informally make use of Barendrengt's convention [UBN07] and assume that bound variable identifiers are generally unique. In particular, we treat contexts as sets of typed variables and substitutions as maps.

We now state the theorem that we will spend proving the rest of this section.

▶ **Theorem 3** (Strong Normalization). $\Gamma \vdash t\colon T \implies t \in \mathcal{SN}$

We proceed in three steps. First, we *attempt* a naive induction on the the typing derivations (the antecedence), which will get stuck in one case. Second, we attempt induction over a strengthened formulation (employing our first logical relation), which partially solves the problem, but makes another case unprovable. Third, we strengthen the formulation a final time to arrive at something provable and from which strong normalization can be recovered as a corollary.

2.1.1 Attempt #1: Naive Induction

Let us first make explicit all quantifiers in the theorem statement above lest to hide subtle details.

▶ **Formulation 1.** *For all terms t, contexts Γ, and typing derivations of $\Gamma \vdash t\colon T$, we have $t \in \mathcal{SN}$.*

This reformulation makes clear that we can in fact induct on the typing derivation (as a universally quantified object) – which before were hidden as the antecedence of an implication. This will be the only time that we are that explicit for clarity. In the statements to come, we will suppress the clutter again.

▶ Proof Attempt (of Formulation 1). We induct on typing derivations.

- case TP-VAR: necessarily $t = x$ for some variable x, and $t \in \mathcal{SN}$ immediately by definition of $\mathcal{SN}$.
- case TP-ABS: we have $t = \lambda x.\ s$ and the judgement

$$\frac{\Gamma, x\colon T_1 \vdash s\colon T_2}{\Gamma \vdash \lambda x.\ s\colon T_1 \to T_2} \quad \text{(TP-ABS)}$$

  By induction hypothesis we have $s \in \mathcal{SN}$, hence $(\lambda x.\ s) \in \mathcal{SN}$ immediately by definition of $\mathcal{SN}$.
- case TP-APP: we have $t = s_1\ s_2$ and the judgement

$$\frac{\Gamma \vdash s_1\colon T_1 \to T_2 \qquad \Gamma \vdash s_2\colon T_1}{\Gamma \vdash s_1\ s_2\colon T_2} \quad \text{(TP-APP)}$$

  By induction hypotheses we have $s_1 \in \mathcal{SN}$ and $s_2 \in \mathcal{SN}$. But it is unclear how we can combine them to show the desired goal of $s_1\ s_2 \in \mathcal{SN}$. **We are stuck.**

2.1.2 Attempt #2: Strengthening the Induction by a Logical Relation

To fix the proof, we might try a very common (folklore) technique to fix stuck inductions: instead of trying to show the desired theorem X, we strengthen X to X' (i.e. $X' \implies X$) and show X' via induction. This way, in the induction proof for X' we have access to strengthened induction hypotheses, which hopefully remedy any stuckness that previously existed.

In general, there is no systematic way to choose X', and in fact it may require intuition from a domain expert, much like indefinite integration of complicated functions. If we choose X' too weak, we may again be stuck. And if we choose X' to be too strong, it may become unnecessarily hard to prove or even unprovable. Nonetheless, a good first strategy is to bake the missing proof steps right into X'. And if we get stuck again, we just iterate this strategy until we hopefully arrive at a fixpoint. This is what we will do in Formulations 2 and 3.

In our case, we failed to show $s_1\ s_2 \in \mathcal{SN}$ in the induction case of the typing derivation for function applications (TP-APP). Hence, let us take the desired theorem ($\Gamma \vdash t\colon T \implies t \in \mathcal{SN}$) and strengthen by *exactly* the missing step.

▶ **Formulation 2.** $\Gamma \vdash t\colon T \implies P_T(t)$ *where for every type T, $P_T(\cdot)$ is a relation on terms of type T given by*

$$P_B(b) := b \in \mathcal{SN}$$
$$P_{T_1 \to T_2}(s) := s \in \mathcal{SN} \text{ and } (\forall t.\ P_{T_1}(t) \implies P_{T_2}(s\ t))$$

This is our first formulation involving a logical relation. We introduce some terminology in a second; before let us look at what we have done. Overall, we strengthened $t \in \mathcal{SN}$ to $P_T(t)$. For terms b of base type B, we see that we have not performed any strengthening compared to the previous formulation. But for terms s of function type, in addition to claiming $s \in \mathcal{SN}$, we now also claim the term $s\ t$ to be in the relation (at its type) for all terms t that are in the relation (at a type compatible with the parameter type of s).

Terminology The type-indexed family P of relations is called a *logical relation*. In general, given a type theory (e.g. STLC), a **logical relation** is **a typed-indexed family P of unary relations that for every type T specify a relation $P_T(\cdot)$ on terms of type T**. Be sure to take this as an informal definition only; flavors of logical relations vary in the literature and are often fit to the type theory under consideration. Note that the formulation above is not our desired theorem (Theorem 3), but rather supposed to be an intermediate theorem (if provable) following the specific pattern "if a term t types against T, then t is in the relation at its type." An intermediate theorem of such a form is called **Basic Lemma** (also known as parametricity, abstraction theorem, or fundamental theorem).

Often, desired theorems can be recovered as corollaries from Basic Lemmas and that is also the case here. Since $t \in \mathcal{SN}$ is mentioned as-is by the logical relation itself, we could get strong normalization as an easy corollary if the formulation was provable (it is not). Let us try proving it, again by induction on typing derivations. In the proof attempt below, we will observe a common phenomenon when strengthening inductions: the case that one intended to fix becomes provable, but other cases become harder to prove if not unprovable.

▶ Proof Attempt (of Formulation 2). We induct on typing derivations. Compared to the previous proof attempt, we reordered the cases for didactical reasons.

- case TP-APP: we have $t = s_1\ s_2$ and the judgement

$$\frac{\Gamma \vdash s_1 \colon T_1 \to T_2 \qquad \Gamma \vdash s_2 \colon T_1}{\Gamma \vdash s_1\ s_2 \colon T_2} \quad \text{(TP-APP)}$$

  By induction hypotheses we have $P_{T_1}(s_1)$ and $P_{T_2}(s_2)$. The former in particular yields $P_{T_2}(s_1\ s_2)$ concluding this case.
- case TP-VAR: necessarily $t = x$ for some variable x and $t \in \mathcal{SN}$. If T is a base type, we are done since $P_T(t)$ only amounts to $t \in \mathcal{SN}$. Otherwise, let $T = T_1 \to T_2$ be a function type. We need to show $P_{T_2}(x\ s)$ for all terms s with $P_{T_1}(s)$. For the moment, let us restrict to the case where T_2 is a base type. Then, apart from $t \in \mathcal{SN}$ (which we have already discussed) our proof goal would amount to $P_{T_2}(x\ s) \iff (x\ s) \in \mathcal{SN}$, which is immediate since x is a variable and the argument in the function application is strongly normalizing by assumption $P_{T_1}(s)$.
  However T_2 needs not be a base type: in general it is of the form $T_2 = T_2^1 \to \ldots \to T_2^n$ for some types T_2^i where $1 \leq i \leq n$ and T_2^n is a base type. In this general case, after unfolding the definition of $P\ n$ times, we need to show

$$P_{T_2^n}(x\ s^1\ \ldots\ s^n) \iff (x\ s^1\ \ldots\ s^n) \in \mathcal{SN}$$

  for terms s^i with $P_{T_2^i}(s^i)$. This holds by an argument analogous to above: x is a variable and all arguments in the function application are strongly normalizing by assumptions $P_{T_2^i}(s^i)$.
- case TP-ABS: we have $t = \lambda x.\ s$ and the judgement

$$\frac{\Gamma, x \colon T_1 \vdash s \colon T_2}{\Gamma \vdash \lambda x.\ s \colon T_1 \to T_2} \quad \text{(TP-ABS)}$$

By induction hypothesis we have $P_{T_2}(s)$. We need to show $P_{T_1 \to T_2}(\lambda x.\ s)$. The left conjunct of this proof goal, $(\lambda x.\ s) \in \mathcal{SN}$, follows from the induction hypothesis. For the second conjunct, let r be a term with $P_{T_1}(r)$. We need to show $P_{T_2}((\lambda x.\ s)\ r)$. Let us *optimistically assume* that we had $P_{T_2}(s[x \mapsto r]) \implies P_{T_2}((\lambda x.\ s)\ r)$, i.e. assume that our logical relation was stable under β-expansion and it sufficed to show $P_{T_2}(s[x \mapsto r])$.(In general, this is a reasonable side claim to make for logical relations in the setting of STLC.) But even that we are unable to show. Concretely, we are **deviating from our induction hypothesis $P_{T_2}(s)$ by a substitution. We are stuck.**

2.1.3 Attempt #3: Closing the Logical Relation under Substitution

Again, let us apply the technique of strengthening the induction to resolve the point at which we got stuck. This time, our failure suggests to *close the logical relation under substitutions*.

To do so, we first need an auxiliary definition. Let P be a logical relation, i.e. for every type T a relation $P_T(\cdot)$ on terms of type T. We homomorphically extend this family to contexts and substitutions (which bear a relationship similar to types and terms) and define for every context Γ and compatible substitutions $\vdash \gamma \colon \Gamma$:

$$P_\Gamma(\gamma) := \forall\, (x \colon T) \in \Gamma.\ \ P_T(x\gamma)$$

▶ **Theorem/Formulation 3.** *$\Gamma \vdash t \colon T$ and $P_\Gamma(\gamma) \implies P_T(t\gamma)$ for all substitutions with $\vdash \gamma \colon \Gamma$ and where*

$$P_B(b) := b \in \mathcal{SN}$$
$$P_{T_1 \to T_2}(s) := s \in \mathcal{SN} \text{ and } (\forall t.\ P_{T_1}(t) \implies P_{T_2}(s\ t))$$

It turns out that this final formulation is sufficient in remedying any stuckness and thus provable. To recover strong normalization as a corollary, we instantiate with the identity substitution on Γ (which is in P_Γ). We skip proving these claims at this point and refer interested readers to Appendix A.

In the next section, we summarize our findings on the necessity and patterns of logical relations.

2.2 Logical Relations: Summary

Why are logical relations needed? In many type theories, we can observe meta theorems of the form "$\Gamma \vdash t \colon T \implies X$."[2] It may very well happen that a *naive* induction on typing derivations gets stuck. In these cases, we may have luck **strengthening the induction by utilizing stronger, type-indexed assertions**. This way, we gain access to stronger induction hypotheses in the proof (but also stronger proof goals). We call the family of type-indexed assertions a **logical relation**. And we call proofs using a logical relation **proofs by logical relation**.

Patterns of Proofs by Logical Relation We distill our methodology to prove strong normalization in the last section into the following *patterns*. These patterns are deliberately kept informal to best convey a general and high-level understanding of the technique behind proofs by logical relation.

[2] e.g. strong normalization, type safety, correctness of compiler optimizations, program equivalences, ...

For readers interested in formal definitions, we refer to the citations in [RS13, Sect. 1]. In particular, we mention [BJP12] which defines logical relations not for a specific type theory, but for a large class, thus aligning well with the spirit of this exposition.

▶ **Pattern 1** (Type Theory). *A type theory is a triple of* ***terms***, ***types***, *and a ternary* ***typing judgement*** *between contexts (which are lists of typed variables), terms, and types.*

We write t, T, Γ for terms, types, and contexts, and $\Gamma \vdash t\colon T$ for the typing judgement of t having type T in context Γ.

▶ **Pattern 2** (Proofs by Logical Relation). *Proofs by logical relation over some type theory following Pattern 1 proceed as follows.*

i) ***Define a logical relation*** *P: for every type T, $P_T(\cdot)$ is a relation on terms of type T*
ii) ***Prove the Basic Lemma***

$$\Gamma \vdash t\colon T \implies P_T(t)$$

"if a term is well-typed, then it is in the relation at its type"

by induction on typing derivations.
iii) ***Recover desired theorem*** *as corollary*

The remaining parts of this paper are structured as follows: In Section 3, we recap Mmt/LF, the pairing of the Edinburgh Logical Framework and the Mmt module system, as means to modularly define formal systems. Then, in Section 4 we develop an approach for representing proofs by logical relation in Mmt/LF in a way that is idiomatic to Mmt/LF.

3 Logical Frameworks: Mmt/LF

In this section, we shortly review the general idea of logical frameworks (we assume the reader's familiarity) and then present Mmt/LF: a pairing of a module system and a logical framework. This is the system for which we will develop a representation method of proofs by logical relation in the next section.

Logical frameworks are an approach for defining logics in a convenient, boilerplate-free way. Consider the following central ingredients that almost all logics and type theories possess [HHP93; Pfe01]:

- syntax (terms, variables, contexts)
- binding and substitution (e.g. functions, function application, β-reduction, capture-avoiding substitution; similarly: quantifiers, quantifier elimination)
- syntax checking (incl. type checking)
- proof syntax (incl. schematic proof rules, proof rules with side conditions, proof terms, variables, contexts)
- proof checking

Suppose we wanted to define some given logic, where by "define" we mean that we either formalize it in a formal system or even implement it using a programming language. Without a logical framework, we would need to define all of the above, which would be a time-consuming and error-prone task. And if we wanted to define a whole atlas of logics [Cod+], we would be facing a hopeless endeavor.

"A logical framework provides a means to define (or present) a logic as a signature in a higher-order type theory," ([Wik19]) in a way that the above ingredients emerge as mere

instantiations almost for free. Let us denote the logical framework's type theory by LF. For many logics L, we can reduce defining a logic's syntax (incl. binding and substitution) to defining an LF-signature Σ_L with constants of suitable type (incl. higher-order functions and function application, respectively). The logic's syntax is thus represented by well-typed Σ_L-terms, and we can do similarly for representing proof rules and terms. This lets us recover syntax, proof, and type checking for L as instantiations of syntax and type checking of Σ_L. And the latter is already given (i.e. defined/implemented) by LF. Overall, for defining many logics (with the ingredients above), it suffices to simply give a suitable LF-signature.

In other words, a **logical framework offers general notions that when instantiated yield adequate definitions for formal systems**. The endeavor of this paper is in a similar spirit: we extend the pairing of a logical framework and the Mmt module system (which we explain below) with a general notion of proofs by logical relation. And when instantiated to specific logics (or type theories) T, this yields adequate representation methods of T-proofs by logical relation.

Mmt/LF The Edinburgh Logical Framework [HHP93] (to which we refer by LF) is a dependent type theory suitable to define a wide variety of formal systems including many variants of first- and higher-order logic and set and type theory [Cod+]. Mmt/LF is the combination of LF with the Mmt module system [RK13], which induces a language of modular *theories* and *views* encapsulating LF-signatures and LF-signature morphisms, respectively. Although in this paper we stay purely theoretical, we note that Mmt is also a suitable choice implementation-wise since its reference implementation is built around offering a generic API for rapid prototyping [MR19; RR20]. In fact, the representation methods we propose in this paper have already been implemented as part of joint work of Florian Rabe and the author [RR21]. In the following, we simply write Mmt to mean Mmt/LF.

Below, we describe Mmt and LF in parallel. The primary way to formalize knowledge in Mmt is via Mmt-*theories*. (Generally, if necessary, we prepend words with "Mmt-" to disambiguate from meta concepts. However, if clear from context, we leave out such prefixes.)

▶ **Definition 4** (Theory). *The grammar for theories, terms, and contexts is*

Thy	$::=$	$\textbf{theory}\ T = \{Decl^*\}$	*theory definition*
$Decl$	$::=$	$\textbf{include}\ T \mid c\colon A\,[= t]$	*declarations in a theory*
f, t, A, B	$::=$	$\mathtt{type} \mid \Pi x{:}A.\ B \mid$ $c \mid x \mid f\ t \mid \lambda x{:}A.\ t$	*terms*
Γ	$::=$	$\cdot \mid \Gamma, x\colon A$	*contexts*

Here, T, c stand for identifiers of theories and constants, and x for identifiers of variables. We write $A \to B$ as an abbreviation for $\Pi x\colon A.\ B$ if x does not occur freely in B.

Theories are effectively lists of constant declarations $c\colon A\,[= t]$, which all have a type and, optionally, a definiens component. Include declarations allow to modularly build up hierarchies of theories.

We can use theories to represent type theories. Below, we formalize a theory `PL` representing propositional logic, our running example of a type theory represented in Mmt. In the next section, we will consider proofs by logical relation over `PL` and also show how definitions and theorems generalize to arbitrary theories. Our choice of such a rudimentary type theory is deliberate to enable an accessible presentation. We refer to [RS13] for a discussion of STLC and the representation of corresponding logical relations.

▶ **Example 5** (Syntax of Prop. Logic)**.** Propositional logic can be formalized in Mmt as follows.

$$\textbf{theory } \texttt{PL} = \left\{ \begin{array}{l} \texttt{prop}\colon \texttt{type} \\ \quad \neg\colon \texttt{prop} \to \texttt{prop} \\ \quad \wedge\colon \texttt{prop} \to \texttt{prop} \to \texttt{prop} \end{array} \right\}$$

There is no type of atoms; instead in theories extending PL (i.e. via includes) atoms are represented as constants $a\colon \texttt{prop}$.

The example above demonstrates common themes of Mmt: **knowledge is formalized *declaratively*, *modularly*, and with an *open-world assumption***. For example, any theory extending PL can add another production for the syntax of propositional logic by stating a declaration with return type prop. This is in contrast to proof assistants such as Coq, Isabelle/HOL, Agda, and Lean where inductive datatypes and functions are commonly used to represent syntax and proofs of formal systems. Arguably, in those proof assistants knowledge is formalized *inductively*, *monolithically*, and with a *closed-world assumption*. This makes extending formalizations for the sake of building large hierarchies cumbersome (e.g. see [Boi04] for a problem description and a proposed workaround). But in Mmt, doing so is a natural and supported workflow as the example below demonstrates.

Both approaches have their merits and we mentioned them for the following reason: When conceiving of a way to represent logical relations, be it in Mmt or some proof assistant, it is important to custom-fit to yield the most idiomatic integration into the underlying sytem.

▶ **Example 6** (Extending Prop. Logic to First-Order Logic)**.** We can extend PL with defined connectives to get PLExt.

$$\textbf{theory } \texttt{PLExt} = \left\{ \begin{array}{l} \textbf{include } \texttt{PL} \\ \vee\colon \texttt{prop} \to \texttt{prop} \to \texttt{prop} \\ \qquad = \lambda p\colon \texttt{prop}.\ \lambda q\colon \texttt{prop}.\ \neg(\neg p \vee \neg q) \\ \supset\colon \texttt{prop} \to \texttt{prop} \to \texttt{prop} \\ \qquad = \lambda p\colon \texttt{prop}.\ \lambda q\colon \texttt{prop}.\ \neg p \vee q \\ \Leftrightarrow\colon \texttt{prop} \to \texttt{prop} \to \texttt{prop} \\ \qquad = \lambda p\colon \texttt{prop}.\ \lambda q\colon \texttt{prop}.\ (p \supset q) \wedge (q \supset p) \end{array} \right\}$$

Yet another extension with suitable undefined constants yields a toy representation of first-order logic FOL.

$$\textbf{theory } \texttt{FOL} = \left\{ \begin{array}{l} \textbf{include } \texttt{PLExt} \\ \texttt{ind}\colon \texttt{type} \\ \quad \forall\colon (\texttt{ind} \to \texttt{prop}) \to \texttt{prop} \\ \quad \doteq\colon \texttt{ind} \to \texttt{ind} \to \texttt{prop} \end{array} \right\}$$

In the sequel, whenever we write theory, we mean a well-typed theory. For the purpose of this paper, precise definitions of typing rules or well-formedness conditions are irrelevant, and we appeal to the reader's intuition (e.g. concerning function types). We refer interested readers to [Mül19, Sect. 4.2 – 4.3]. For our presentation, it suffices to note that for a constant within a theory to be well-formed, it is necessary that its identifier is unique in scope and that type and definiens are closed terms over all previously declared and included symbols.

Concerning well-typedness, there are two judgements that we will make use of:

- $\Gamma \vdash_T A$ inh.: term A is *inhabitable*, i.e. may appear as a type of constants and terms
- $\Gamma \vdash_T t\colon A$: term t has type A

In both cases, the judgements are relative to a context Γ over the theory T. When contexts are empty, we simply omit them, such as in $\vdash \texttt{type}$ inh. (a typing rule built-in into Mmt). A well-formed constant $c\colon A\,[= t]$ in a theory T is well-typed if $\vdash_T A$ inh. and $\vdash_T t\colon A$ hold.

Another important Mmt primitive are *views*, which represent compositional translations between theories.

▶ **Definition 7** (View). *The grammar for views is*

View	::=	**view** $v\colon S \to T = \{Ass^*\}$	*view definition*
Ass	::=	**include** $v \mid c := t$	*assignments in a view*

Here, in addition to Definition 4, v stands for identifiers of views.

We treat (well-formed) views as functions and write $v(c)$ to refer to the term assigned to c. Given a view $v\colon S \to T$, we define its homomorphic extension $\overline{v}$ as a function translating S- to T-terms and -contexts:

$$\begin{array}{ccc} & \overline{v}(c) = v(c) & \\ \overline{v}(\texttt{type}) = \texttt{type} & \overline{v}(\Pi x\colon A.\ B) = \Pi x\colon \overline{v}(A).\ \overline{v}(B) & \\ \overline{v}(x) = x \quad \overline{v}(f\ t) = \overline{v}(f)\ \overline{v}(t) & & \overline{v}(\lambda x\colon A.\ t) = \lambda x\colon \overline{v}(A).\ \overline{v}(t) \\ & \overline{v}(\cdot) = \cdot \quad \overline{v}(\Gamma, x\colon A) = \overline{v}(\Gamma), x\colon \overline{v}(A) & \end{array}$$

In the sequel, we write v for $\overline{v}$.

A view $v\colon S \to T$ is

- well-formed if it maps all undefined constants (and nothing more) declared and included in S
- well-typed if for all assignments $c := t$ to constants $c\colon E$, we have $\vdash_T \overline{v}(t)\colon \overline{v}(E)$

There are multiple guiding intuitions for thinking about the meaning of views [Rab17, Sect. 3][Rou19, Sect. 3][MMT]. For the purpose of this paper, it suffices to think about views from the following software engineering perspective. In addition to representing formal systems, we can think of **theories as classes/interfaces**, and if they contain undefined constants, we can consider them to be abstract classes. And we can interpret a view $v\colon S \to T$ as an implementation of the abstract parts of S by means of the T-terms given in v. In other words, **a view $S \to T$ witnesses that (and says how) T implements the interface of S.**

Lastly, we present a formalization `PLND` of a toy fragment of an intuitionistic natural deduction calculus for our theory of extended propositional logic from Example 6. We do so for two reasons. First, this formalization demonstrates the judgements-as-types pattern of which we will make heavy use of later when modeling logical relations in Mmt. Second, we will also use the proof calculus in some concrete examples of logical relations.

$$\textbf{theory } \texttt{PLND} = \left\{\begin{array}{l}\textbf{include } \texttt{PLExt}\\ \Vdash\colon \texttt{prop} \to \texttt{type}\\ \neg\neg_{\texttt{I}}\colon \Pi p\colon \texttt{prop}.\ \Vdash p \to \Vdash \neg\neg p\\ \wedge_{\texttt{I}}\colon \Pi p\, q\colon \texttt{prop}.\ \Vdash p \to \Vdash q \to \Vdash p \wedge q\\ \wedge_{\texttt{EL}}\colon \Pi p\, q\colon \texttt{prop}.\ \Vdash p \wedge q \to \Vdash p\\ \wedge_{\texttt{ER}}\colon \Pi p\, q\colon \texttt{prop}.\ \Vdash p \wedge q \to \Vdash q\\ \vee_{\texttt{IL}}\colon \Pi p\, q\colon \texttt{prop}.\ \Vdash p \to \vdash p \vee q\\ \vee_{\texttt{IR}}\colon \Pi p\, q\colon \texttt{prop}.\ \Vdash q \to \vdash p \vee q\\ \vee_{\texttt{E}}\colon \Pi p_1\, p_2\, q\colon \texttt{prop}.\ \Vdash p_1 \vee p_2 \to\ (\Vdash p_1 \to \Vdash q) \to (\Vdash p_2 \to \Vdash q) \to \Vdash q\\ \ldots\end{array}\right\}$$

Here, we model the provability of a proposition $\vdash_{\texttt{PLND}} p\colon \texttt{prop}$ by the inhabitedness of $\Vdash p$. This way, natural deduction rules correspond to undefined constants with function types that "axiomatically" give inhabitants of $\Vdash p$ for certain p. This is an instance of the judgement-as-types pattern [HHP93] phrased in the setting of Mmt:

▶ **Pattern 3** (Judgements as Types). *Let $T_1, \ldots, T_n$ be Mmt-types (e.g. constant declarations $T_i\colon \texttt{type}$). We can represent an n-ary judgement (or relation) on terms of those types as follows.*

i) Introduce a new constant declaration $\texttt{judg}\colon T_1 \to \ldots \to T_n \to \texttt{type}$,
e.g. $\Vdash\colon \texttt{prop} \to \texttt{type}$
Let $t_1, \ldots, t_n$ be terms of type $T_1, \ldots, T_n$, respectively. The type $\texttt{judg}\ t_1\ \ldots\ t_n$ *is*
- *the type of all proofs of $t_1, \ldots, t_n$ being in the relation,*
- *thus inhabited if and only if "$t_1, \ldots, t_n$ are in the relation"*

ii) Encode derivation rules as (dependent) function types, with return type $\texttt{judg}\ \ldots$,
e.g. $\texttt{trueI}\colon \Vdash \texttt{true}$ *or* $\wedge_{\texttt{EL}}\colon \Pi p\, q\colon \texttt{prop}.\ \Vdash p \wedge q \to \Vdash p$

4 Representing Logical Relations in Mmt/LF

We now aim at combining the previous sections on formalizations in Mmt and on use of logical relations to prove meta theorems in type theories. Concretely, we strive for a **declarative and modular representation of proofs by logical relation** in Mmt. We closely follow Rabe and Sojakova [RS13], and the material in this section is essentially a special case of their approach.[3]

Notably, we employ a novel presentation of the material by fleshing out an idea communicated to the author by Florian Rabe. This way, to represent logical relations, apart from theories and views we do not need any new language primitives (as was the case in [RS13]). This idea was inspired by recent joint work on a framework of metaprogramming facilities in Mmt [RR20], in which we the representation of logical relations can be handled as a single metaprogramming operator. As such it has been implemented for but not disccused in [RR21].

In this section we proceed in four steps:

[3] In their terminology, we only consider logical relations along a single identity morphism.

- We walk through representing proofs by logical relation over propositional logic acting as a toy type theory (Section 4.1).
- We discuss two exemplary meta theorems about propositional logic in the representation method just developed (Section 4.2).
- We generalize to the case of proofs by logical relation over arbitrary MMT-theories representing arbitrary type theories (Section 4.3).
- We shortly evaluate our approach regarding MMT's themes of modularity (Section 4.4).

Note that in the first two points we deliberately opted for propositional logic as an exemplary (but rudimentary) type theory to keep our presentation accessible. We refer interested readers to [RS13, Ex. 4.6] for a discussion of how strong normalization of STLC can be represented as a formalized proof by logical relation (in the essentially same approch as ours).

4.1 Representing Logical Relations over Prop. Logic

Recall the theory PL from Example 5 (reprinted below) formalizing propositional logic. Throughout this section, keep in mind that by PL we always refer to the MMT-theory below; and not to the concept of propositional logic itself.

$$\textbf{theory } \texttt{PL} = \left\{ \begin{array}{l} \texttt{prop}: \texttt{type} \\ \quad \neg: \texttt{prop} \to \texttt{prop} \\ \quad \wedge: \texttt{prop} \to \texttt{prop} \to \texttt{prop} \end{array} \right\}$$

Our goal in this section is to define a suitable notion of logical relations over PL in a way that generalizes to arbitrary MMT-theories representing arbitrary type theories. We proceed in two steps. First, we make explicit how we can interpret PL as a type theory matching Pattern 1. This lets us recall Pattern 2 (describing proofs by logical relation in general) and draw motivation for modeling it step-by-step in MMT, which is what we do in the second step.

PL as a Type Theory PL behaves like a (rudimentary) type theory according to Pattern 1:

(i) the syntax is the set of all well-typed MMT-terms generated by $\neg$, $\wedge$ (and free variables)
(ii) the set of types is a singleton and only contains prop
(iii) the typing judgement between terms and types is given by the full relation: all terms of the syntax type against prop[4]

Observe how all three ingredients to interpreting PL as a type theory can be read off the theory definition of PL itself: First, the syntax is the set of MMT-well-typed MMT-terms generated by all constant declarations that are not types themselves. Second, the set of types is the set of all constant declarations that are types. And third, the typing judgement is inherited from MMT; after all the syntax only contains well-typed terms by construction. In fact, in a similar way, we can interpret all MMT-theories as type theories.

Having interpreted PL as a type theory, we can now refer to Pattern 2 (reprinted and adapted below) to assess how we should define proofs by logical relation over PL. In the next step, we instantiate the pattern for PL seen as a type theory and model it step-by-step in MMT.

[4] This might seem trivial, but it just is a confirmation of the adequacy of LF and the shallow embedding used to formalize propositional logic.

i) Define a logical relation, i.e. a type-indexed family of relations: for every type T, a relation $P_T(\cdot)$ on terms of type T
ii) Prove the Basic Lemma "$\Gamma \vdash t\colon T \implies P_T(t)$" by induction on typing derivations
iii) Recover the desired theorem as a corollary

Modeling proofs by logical relation over `PL` How can we model the pattern above in the formal setting of Mmt? (And how can we model it for arbitrary Mmt-theories interpreted as type theories?) Speaking figuratively, we are looking for an input method in Mmt for users to state a proof by logical relation over `PL`. Keep in mind that the approach we come up with should ideally be similar to how theories and views are stated, and in particular be declarative and modular – to best align with Mmt's themes.

For step 1, we seek a way to model the requirement to state a relation (of suitable form) for every type in the type theory at hand. For `PL` seen as a type theory we only have the type `prop`, and we can model the relation itself using judgements-as-types (cf. Pattern 3): relations on terms of type `prop` can be modeled by Mmt-terms of Mmt-type $\texttt{prop} \to \texttt{type}$. For the requirement to state such a relation, we introduce a **theory** $\texttt{PL}_\texttt{r} = \{\textbf{include}\ \texttt{PL}, \ldots\}$, which we build up in the following. The idea is to see $\texttt{PL}_\texttt{r}$ as an "interface theory" and to represent (proofs by) logical relations over `PL` as views with domain $\texttt{PL}_\texttt{r}$. To be precise in our language, we refer to views with domain $\texttt{PL}_\texttt{r}$ as Mmt-*logrels*, which distinguishes them from logical relations as a meta concept. We will see later that Mmt-logrels inherit by construction the declarative and modular nature of views. We now add the following undefined constant to $\texttt{PL}_\texttt{r}$:

$$\texttt{prop}_\texttt{r}\colon \texttt{prop} \to \texttt{type}$$

This forces Mmt-logrels to give an assignment to $\texttt{prop}_\texttt{r}$, thus capturing the requirement to state a relation on terms of type `prop`.

For step 2, we seek a way to encode proofs of the Basic Lemma. Since we assume the proof is done using induction on typing derivations, we can reduce our task to encoding proofs of the individual cases of the induction. (Although intuitive, the fact that these necessarily combine into an overall proof requires a non-trivial proof later.) Recall that for `PL`, when seen as a type theory, the typing judgement of `PL` is given by Mmt's own typing judgement. Hence, all possible ways to form a typing derivation for `PL`, when seen as a type theory, exactly correspond to all the ways to construct Mmt-terms that from Mmt's perspective type gainst `prop`. And a finite description of those ways can be read off the definition of `PL`: they are generated by the set of constants with return type `prop` (here: $\neg_\texttt{r}$, $\wedge_\texttt{r}$). Continuing the spirit of judgements-as-types, we add the requirements below to our interface theory $\texttt{PL}_\texttt{r}$, each synthesized for each constructor:

$$\begin{aligned}
\neg_\texttt{r}\colon &\Pi p\colon \texttt{prop}.\ \Pi p^*\colon \texttt{prop}_\texttt{r}\ p.\ \ \texttt{prop}_\texttt{r}\ (\neg p)\\
\wedge_\texttt{r}\colon &\Pi p\colon \texttt{prop}.\ \Pi p^*\colon \texttt{prop}_\texttt{r}\ p.\\
&\qquad \Pi q\colon \texttt{prop}.\ \Pi q^*\colon \texttt{prop}_\texttt{r}\ q.\ \ \texttt{prop}_\texttt{r}\ (p \wedge q)
\end{aligned}$$

For example, the first constant puts the following requirement onto implementors (i.e. views out of $\texttt{PL}_\texttt{r}$): given an arbitrary proposition p, an arbitrary proof p^* of p being in the relation assigned to $\texttt{prop}_\texttt{r}$ (the induction hypothesis), you need to give a proof of $\neg p$ being in the relation, too. In total, the two requirements above can also be seen as the condition of the relation assigned to $\texttt{prop}_\texttt{r}$ being a congruence wrt. all of `PL`'s constructors.

Step 3 is about using the Basic Lemma to recover some desired theorem. Since how to model this best in Mmt is still an open research question at time of writing, we will not touch upon Step 3 at all in this paper.

Formal Definitions We collect all mentioned ideas and requirements in the definition below.

▶ **Definition 8** (MMT-Logrel over PL). *An* MMT*-logrel over* PL *is a view with domain* $\texttt{PL}_\texttt{r}$*, which is defined as*

$$\textbf{theory } \texttt{PL}_\texttt{r} = \left\{ \begin{array}{l} \textbf{include } \texttt{PL} \\ \texttt{prop}_\texttt{r} : \texttt{prop} \to \texttt{type} \\ \quad \neg_\texttt{r} : \Pi p : \texttt{prop}.\ \Pi p^* : \texttt{prop}_\texttt{r}\ p.\ \ \texttt{prop}_\texttt{r}\ (\neg p) \\ \quad \wedge_\texttt{r} : \Pi p : \texttt{prop}.\ \Pi p^* : \texttt{prop}_\texttt{r}\ p. \\ \qquad\quad \Pi q : \texttt{prop}.\ \Pi q^* : \texttt{prop}_\texttt{r}\ q.\ \ \texttt{prop}_\texttt{r}\ (p \wedge q) \end{array} \right\}$$

Overall, MMT-logrels over PL represent stating a logical relation over PL seen as a type theory *and* giving proofs of individual cases for the (intended) induction on typing derivations. This makes MMT-logrels more than just the formalized counterpart of logical relations, which is another reason we opted for "logrel" in our naming convention. The codomain X of an MMT-logrel $\texttt{PL}_\texttt{r} \to X$ is the *semantic domain* in which the relations over types of PL are stated and in which the individual cases of the Basic Lemma are proved.

Below, we foreshadow the MMT-Basic Lemma for PL which shows that the cases provided for in views MMT-logrels in fact combine to a larger result. We will introduce the missing notation in the section after next, where we give a generalized version of the theorem anyway.

▶ **Theorem 9** (MMT-Basic Lemma for PL). *It holds*

$$\Gamma \vdash_{\texttt{PL}} p : \texttt{prop} \implies r(\Gamma) \vdash_{\texttt{PL}_\texttt{r}} r(p) : \texttt{prop}_\texttt{r}\ p$$

for some function $r(\cdot)$ *on* MMT*-terms and -contexts. In particular, if* $R : \texttt{PL}_\texttt{r} \to X$ *is an* MMT*-logrel over* PL*, then*

$$\Gamma \vdash_{\texttt{PL}} p : \texttt{prop} \implies R(r(\Gamma)) \vdash_X R(r(p)) : R(\texttt{prop}_\texttt{r})\ p$$

The theorem establishes that if a term p types against prop, then there is an inhabitant of $R(\texttt{prop}_\texttt{r})\ p$. By the judgement-as-types idiom, this means that p is in the relation represented by $R(\texttt{prop}_\texttt{r})$, i.e. precisely the relation the MMT-logrel R assigned to $\texttt{prop}_\texttt{r}$.

This concludes our walk through representing proofs by logical relation for PL. Below in form of a pattern, we reiterate the approach we developed and invite the reader to compare it with Pattern 2.

▶ **Pattern 4** (MMT-Proofs by Logical Relation for PL). *Proofs by logical relation over* PL *are formalized in* MMT *as follows.*

i) ***Define a logical relation:*** *declare a (yet partial) view* $R : \texttt{PL}_\texttt{r} \to X$ *for some theory* X *with an assignment to* $\texttt{prop}_\texttt{r}$
ii) ***Prove the Basic Lemma:*** *fill out the missing assignments of* R *for* $\neg_\texttt{r}$ *and* $\wedge_\texttt{r}$*; the Basic Lemma follows by Theorem 9*
iii) ***Recover desired theorem****: within* MMT *formalization not yet possible!*

4.2 Examples of Logical Relations over Prop. Logic

Let us look at two examples of meta theorems for (intuitionistic) propositional logic, which are both amenable to be proven via MMT-logrels.

▶ **Theorem 10** (Tertium Non Datur). *If "a or not a" for all atoms a, then "p or not p" for all propositions p.*

▶ **Theorem 11** (Double Negation Elimination). *If "a iff. not not a" for all atoms a, then "p iff. not not p" for all propositions p.*

Proofs of Theorems 10 and 11. Both claims follow by a straightforward inductions on the syntax of propositional logic. For readers interested in fully formalizing these proofs (e.g. by completing the Mmt-logrels given below), we have collected the individual induction cases in Fitch notation in Appendix B. ◀

We deliberately refrained from using mathematical symbols (like $\wedge$, $\neg$) in the statements above to emphasize that these statements happen on the meta level and *not* on the level of our `PL` formalization within Mmt.

To understand why these theorems are amenable to proofs by logical relation, observe that the consequence of each implication is an assertion on propositions that is universally quantified over all propositions. For example, we can understand Theorem 10 as the assertion that, given some assumption on atoms, all propositions are in in the relation that contains propositions p for which "p or not p" is provable. This gives us hope that with our formalization of propositional logic as `PL`, we can recover (representative modelings of) the theorems above as instantiations of Basic Lemmas for `PL`. In the following, we represent suitable Mmt-logrels over `PL` to achieve this.

Let us start by modeling Theorem 10 as an Mmt-logrel, i.e. as a view with domain $\texttt{PL}_\texttt{r}$ (cf. Figure 1). To ease reading of Figure 1, we annotated in gray the expected types of view assignments and omitted inferrable types and arguments.

First, we need to answer two somewhat interdependent questions: what do we assign to $\texttt{prop}_\texttt{r} : \texttt{prop} \to \texttt{type}$ and what codomain do we choose? Above we already expressed the desired relation in natural language. In particular, we talked about provability of propositions, which our formalization (and codomain) also would need to do. For that reason, recall the theory `PLND` from Section 3 which extends `PL` with some derived connectives ($\vee$, $\supset$, $\Leftrightarrow$) and a natural deduction calculus ($\Vdash : \texttt{prop} \to \texttt{type}, \ldots$). Using the latter, we can encode the relation that exactly contains those propositions for which "p or not p" is provable by the Mmt-term $\lambda p : \texttt{prop}.\ \Vdash p \vee \neg p$.

Second, we need to give the remaining assignments acting as proofs of the Basic Lemma's individual cases. The concrete nature of these assignments is irrelevant for our purposes since our main point in this section is the modeling of meta theorems as such. Hence, we only show the case for $\neg_\texttt{r}$ in Figure 1 and omit the one for $\wedge_\texttt{r}$ entirely.

Using `TND` and Theorem 9, we get the below result acting as the formalized variant of Theorem 10.

▶ **Corollary 12** (Formalized Tertium Non Datur). *We have*

$$\Gamma \vdash_{\texttt{PL}} p : \texttt{prop} \implies \texttt{TND}(r(\Gamma)) \vdash_{\texttt{PLND}} \texttt{TND}(r(p)) : \Vdash p \vee \neg p$$

In other words, for every proposition p in `PL` there is a witness of $p \vee \neg p$ being provable (in the sense of $\Vdash$)

▶ Remark 13 (Implicit Condition on Atoms). Note that Corollary 12, in contrast to Theorem 10, does not list any assumption on atoms. This is because our formalization of propositional logic encodes atoms as constants of type `prop` (cf. Example 5). As there are no such constants in `PL`, there is no such condition on atoms; neither in the corollary nor in

$$
\begin{array}{l}
\textbf{view}\ \texttt{TND}\colon \texttt{PL}_{\texttt{r}} \to \texttt{PLND} = \{ \\
\quad \textbf{include}\ \texttt{PL} \\
\quad \texttt{prop}_{\texttt{r}} = \lambda p\colon \texttt{prop}.\ \Vdash p \vee \neg p \\
\qquad\quad : \texttt{prop} \to \texttt{type} \\
\qquad \neg_{\texttt{r}} = \lambda p.\ \lambda p^*.\ \vee_{\texttt{E}} \_\ \_\ \_\ p^*\ (\lambda \mathfrak{p}.\ \vee_{\texttt{IR}} \_\ \_\ (\neg_{\texttt{I}}\ \mathfrak{p}))\ (\lambda \mathfrak{np}.\ \vee_{\texttt{IL}} \_\ \_\ \mathfrak{np}) \\
\qquad\quad : \Pi p\colon \texttt{prop}.\ \Pi p^*\colon\ \Vdash p \vee \neg p.\ \Vdash (\neg p) \vee \neg(\neg p) \\
\qquad \wedge_{\texttt{r}} = \ldots \\
\qquad\quad : \Pi p\colon \texttt{prop}.\ \Pi p^*\colon\ \Vdash p \vee \neg p.\ \Pi q\colon \texttt{prop}.\ \Pi q^*\colon\ \Vdash q \vee \neg q.\ \Vdash (p \wedge q) \vee \neg(p \wedge q) \\
\}
\end{array}
$$

Figure 1 Tertium Non Datur modeled as an MMT-logrel

$$
\begin{array}{l}
\textbf{view}\ \texttt{DNE}\colon \texttt{PL}_{\texttt{r}} \to \texttt{PLND} = \{ \\
\quad \textbf{include}\ \texttt{PL} \\
\quad \texttt{prop}_{\texttt{r}} = \lambda p.\ \Vdash p \Leftrightarrow \neg\neg p \\
\qquad\quad : \texttt{prop} \to \texttt{type} \\
\qquad \neg_{\texttt{r}} = \lambda p.\ \lambda p^*.\ \Leftrightarrow_{\texttt{I}} \_\ \_\ (\lambda \mathfrak{np}.\ \neg_{\texttt{I}}\ \mathfrak{np})\ (\lambda \mathfrak{nnnp}.\ \neg_{\texttt{I}} \_\ \lambda \mathfrak{p}.\ \bot_{\texttt{I}} \_\ (\Leftrightarrow_{\texttt{ER}} \_\ \_\ p^*\ \mathfrak{p})\ \mathfrak{nnnp}) \\
\qquad\quad : \Pi p\colon \texttt{prop}.\ \Pi p^*\colon\ \Vdash p \Leftrightarrow \neg\neg p.\ \Vdash (\neg p) \Leftrightarrow \neg\neg(\neg p) \\
\qquad \wedge_{\texttt{r}} = \ldots \\
\qquad\quad : \Pi p\colon \texttt{prop}.\ \Pi p^*\colon\ \Vdash p \Leftrightarrow \neg\neg p.\ \Pi q\colon \texttt{prop}.\ \Pi q^*\colon\ \Vdash q \Leftrightarrow \neg\neg q.\ \Vdash (p \wedge q) \Leftrightarrow \neg\neg(p \wedge q) \\
\}
\end{array}
$$

Figure 2 Tertium Non Datur modeled as an MMT-logrel

the MMT-logrel in Figure 1. On the other hand, suppose we had a theory $\texttt{PL}'$ extending PL with e.g. a single atom $a\colon \texttt{prop}$ and that we wanted to state Theorem 10 analogously for $\texttt{PL}'$. Then, our MMT-logrel would also be over $\texttt{PL}'$. We have not yet defined MMT-logrels over theories other than PL, but we can already foreshadow Definition 16 at this point: $\texttt{PL}'_r$ effectively equals $\texttt{PL}_r$ with an additional constant $a_r\colon \texttt{prop}_{\texttt{r}}\ a$ and, analogously to before, MMT-logrels over $\texttt{PL}'_r$ are views out of $\texttt{PL}'_r$. Thus, MMT-logrels over $\texttt{PL}'$ would need to prove not only that tertium non datur is preserved by all constructors of PL, but also that it holds as a base case for a.

The modeling of Theorem 11 proceeds analogously. The MMT-logrel is shown in Figure 2 and the resulting Basic Lemma below.

▶ **Corollary 14** (Formalized Double Negation Elimination). *We have*

$$\Gamma \vdash_{\texttt{PL}} p\colon \texttt{prop} \implies \texttt{DNE}(r(\Gamma)) \vdash_{\texttt{PLND}} \texttt{DNE}(r(p))\colon \Vdash p \Leftrightarrow \neg\neg p$$

4.3 Representing Logical Relations in General

We now generalize our methodology to represent proofs by logical relation over arbitrary MMT-theories.

The general idea stays the same. Given a theory T, we first define an interface theory T_r that for every constant $c \in T$ contains a corresponding constant c_r. This constant captures the respective requirement that MMT-logrels over T have to fulfill at c. Then, we define MMT-logrels over T as views out of T_r. Concretely, for a constant $c\colon A\ [= t]$ in T we synthesize $c_r\colon r(A)\ c\ [= r(t)]$ in T_r, where r is some function on T-terms. While reading the formal definition below, we advise readers to maintain the following intuition concerning this function:

- the definition of r follows the cases for the syntax of terms from Definition 4
- if $A\colon \texttt{type}$ is a type constant, then $\vdash_{T_r} r(A)\colon A \to \texttt{type}$, i.e. $r(A)$ represents a relation on terms of type A (via judgements-as-types)
- if $\vdash_T t\colon A$ is a term of type A, then $\vdash_{T_r} r(t)\colon r(A)\ t$, i.e. $r(t)$ is a witness for t being in the relation at its type

Thus, a synthesized constant $c_r\colon r(A)\ c\,[= r(t)]$ in T_r captures the requirement for MMT-logrels to deliver a proof of c being in the relation at its type A if c was undefined. And if c was defined, there is nothing to assign for MMT-logrels at c as the supplied definiens $r(t)$ is already such a proof.

▶ **Definition 15** (MMT-Logrel Interface Theory). *Let T be a theory. Define $r^T(\cdot)$ inductively on T-terms and -contexts by*

$$
\begin{aligned}
r(\texttt{type}) &= \lambda a\colon \texttt{type}.\ a \to \texttt{type}\\
r(\Pi x\colon A.\ B) &= \lambda f\colon (\Pi x\colon A.\ B).\ \Pi x\colon A.\ \Pi x^*\colon r(A)\ x.\ r(B)\ (f\ x)\\
\\
r(c) &= c_r\\
r(x) &= x^*\\
r(f\ t) &= r(f)\ t\ r(t)\\
r(\lambda x\colon A.\ t) &= \lambda x\colon A.\ \lambda x^*\colon r(A)\ x.\ r(t)\\
\\
r(\cdot) &= \cdot\\
r(\Gamma, x\colon A) &= r(\Gamma), x\colon A, x^*\colon r(A)\ x
\end{aligned}
$$

where c_r refers to a constant in the theory that we define next. Above and in the following, we omit the superscript T if the theory over which logrels are considered is clear from context.

*The **logrel interface theory** T_r for T is defined by*

$$
\begin{aligned}
T_r &= \textbf{include}\ T,\ \{\delta_r \mid \delta \text{ is a declaration in } T\}\\
\delta_r &= \begin{cases} \textbf{include}\ S_r & \text{if } \delta \text{ is } \textbf{include}\ S\\ c_r\colon r(A)\ c\ [= r(t)] & \text{if } \delta \text{ is } c\colon A\ [= t] \end{cases}
\end{aligned}
$$

where we translate every declaration δ to δ_r (in order of appearance in T).

▶ **Definition 16** (MMT-Logrel). *Finally, an MMT-**logrel over** T is a view with domain T_r.*

We exemplarily apply Definition 15 on the theories PL and PLExt from Examples 5 and 6. For PL we get:

$$\textbf{theory } \mathtt{PL}_\mathtt{r} = \left\{\begin{array}{l}\textbf{include } \mathtt{PL}\\ \mathtt{prop}_\mathtt{r} : (\lambda a : \mathtt{type}.\ a \to \mathtt{type})\ \mathtt{prop}\\ \quad \neg_\mathtt{r} : (\lambda f : (\Pi x : \mathtt{prop}.\ \mathtt{prop}).\ \Pi x : \mathtt{prop}.\ \Pi x^* : \mathtt{prop}_\mathtt{r}\ x.\\ \qquad \mathtt{prop}_\mathtt{r}\ (f\ x))\ \neg\\ \quad \wedge_\mathtt{r} : (\lambda f : (\Pi x : \mathtt{prop}.\ \mathtt{prop} \to \mathtt{prop}).\ \Pi x : \mathtt{prop}.\ \Pi x^* : \mathtt{prop}_\mathtt{r}\ x.\\ \qquad (\lambda f : (\Pi x : \mathtt{prop}.\ \mathtt{prop}).\ \Pi x : \mathtt{prop}.\ \Pi x^* : \mathtt{prop}_\mathtt{r}\ x.\\ \qquad\quad \mathtt{prop}_\mathtt{r}\ (f\ x))\ (f\ x))\ \wedge\end{array}\right\}$$

Indeed, up to β-reduction and α-renaming, this coincides with the definition of $\mathtt{PL}_\mathtt{r}$ given previously in Definition 8. In the sequel, we identify T_r with the theory that arises by exhaustive β-reduction, and for notational ease, also up to α-renaming.

As the logrel interface theory for PLExt we get the following theory, where we left out some brackets and types to ease reading.

$$\textbf{theory } \mathtt{PLExt}_\mathtt{r} = \left\{\begin{array}{l}\textbf{include } \mathtt{PLExt}\\ \textbf{include } \mathtt{PL}_\mathtt{r}\\ \neg_\mathtt{r} : \Pi p.\ \Pi p^* : \mathtt{prop}_\mathtt{r}\ p.\ \mathtt{prop}_\mathtt{r}\ \neg p\\ \quad = \lambda p\, p^*\, q\, q^*.\ \neg_\mathtt{r}\ (\neg p \vee \neg q)\ (\vee_\mathtt{r}\ \neg p\ (\neg_\mathtt{r}\ p\ p^*)\ \neg q\ (\neg_\mathtt{r}\ q\ q^*))\\ \supset_\mathtt{r} : \Pi p.\ \Pi p^* : \mathtt{prop}_\mathtt{r}\ p.\ \Pi q.\ \Pi q^* : \mathtt{prop}_\mathtt{r}\ q.\ \mathtt{prop}_\mathtt{r}\ p \supset q\\ \quad = \lambda p\, p^*\, q\, q^*.\ \vee_\mathtt{r}\ \neg p\ (\neg_\mathtt{r}\ p\ p^*)\ q\ q^*\\ \Leftrightarrow_\mathtt{r} : \Pi p.\ \Pi p^* : \mathtt{prop}_\mathtt{r}\ p.\ \Pi q.\ \Pi q^* : \mathtt{prop}_\mathtt{r}\ q.\ \mathtt{prop}_\mathtt{r}\ p \Leftrightarrow q\\ \quad = \lambda p\, p^*\, q\, q^*.\ \wedge_\mathtt{r}\ (p \supset q)\ (\supset_\mathtt{r}\ p\ p^*\ q\ q^*)\ (q \supset p)\ (\supset_\mathtt{r}\ q\ q^*\ p\ p^*)\end{array}\right\}$$

Recall that PLExt extended PL with derived connectives (i.e. defined constants) only. For this reason, $\mathtt{PLExt}_\mathtt{r}$ extends $\mathtt{PL}_\mathtt{r}$ with defined constants only, too. Concretely, we can observe that for each defined constant ($\neg$, $\supset$, $\Leftrightarrow$), the definition of the respective interface constant ($\neg_\mathtt{r}$, $\supset_\mathtt{r}$, $\Leftrightarrow_\mathtt{r}$) makes careful use of the starred parameters ("induction hypotheses") to construct a suitable witness. The way in which the witness is constructed is of course given by the function r from Definition 15. Nevertheless, we invite the reader to closely compare the input terms from the definition of PLExt and the (fully β-reduced and α-renamed) output terms above in order to develop an intuitive understanding of r. We leave it as an insightful exercise to the reader to work out $\mathtt{FOL}_\mathtt{r}$ from Example 6, which in contrast to the examples above also features higher-order constants.

▶ Remark 17 (Subtleties in the definition). There are two subtleties in the definition of r in Definition 15 that we glossed over.

First, to ease notation the inductive definition makes use of Barendrengt's convention [UBN07] and assumes that variable names in input terms always differ from variables names that are newly bound in the output terms. For example, consider the term $\Pi f : A.\ B$ for some arbitrary A and B. Applying r despite Barendrengt's convention would yield

$$r(\Pi f : A.\ B) = \lambda f.\ \Pi f : A.\ \Pi f^* : r(A)\ f.\ r(B)\ \underbrace{(f\ x)}_{\text{wrong}}$$

where the f in the annotated subterm wrongly refers to the innermost f, instead of the outermost one. In implementations, we can easily prevent this by always choosing fresh names when newly introducing variables.

Second, the inductive definition in the first case binds a variable of MMT-type `type` which is forbidden by the Edinburgh Logical Framework: the term $r(\texttt{type})$ does actually not typecheck. A similar issue happens with the second case for suitable A and B. Nonetheless, both cases are harmless as long as r is used in precisely the same way as in the definition: whenever we have an inhabitable type $\vdash A$ inh., we only use $r(A)$ as applied to something. This way, illegal lambdas always disappear after exhaustive β-reduction. Arguably, this technical unsoundness of our presentation is worth committing for the notational clarity it allows.[5] ⌟

▶ **Theorem 18.** *T_r is a well-typed theory for any theory T.*

▶ **Theorem 19** (MMT-Basic Lemma)**.** *Let T be some theory. Then*

$$\Gamma \vdash_T t\colon A \implies r(\Gamma) \vdash_{T_r} r(t)\colon r(A)\ t$$

In particular, if $R\colon T_r \to X$ is an MMT-logrel over T, then

$$\Gamma \vdash_T t\colon A \implies R(r(\Gamma)) \vdash_X R(r(t))\colon R(r(A))\ t$$

Proofs of Theorems 18 and 19. Barring the formulation with interface theories, all but the last claim follow from [RS13] (esp. Theorem 3.9) special-cased to logical relations over the single identity morphism (in their terminology). The last claim follows from MMT views preserving typing judgements [Rab17]. ◀

In Pattern 5 below, we summarize our whole approach of representing proofs by logical relation in MMT. Comparing with previous patterns showing proofs by logical relation conceptually (Pattern 2) and in a formalized fashion for `PL` (Pattern 4), we notice that here we only have two steps. The reason is that for arbitrary theories T we cannot distinguish anymore between defining a logical relation over T and proving the individual cases of the corresponding Basic Lemma. In general, the boundaries between types and constructors blur, and types may very well depend on previously declared constructors, which allows relations at those types to depend on proofs corresponding to the constructors used.

▶ **Pattern 5** (MMT-Proofs by Logical Relation)**.** *Let T be an MMT-theory. Proofs by logical relation over T are formalized in MMT as follows.*

i) ***Define an MMT-logrel*** *and* ***Prove the Basic Lemma:*** *declare a view $R\colon T_r \to X$ for some theory X; the Basic Lemma follows by Theorem 19*
ii) ***Recover desired theorem****: within MMT formalization not yet possible!*

4.4 Evaluation in terms of Modularity

How does our approach of representing proofs by logical relation align with MMT's themes? In Section 3, we said MMT emphasizes on declarative and modular formalizations, and our whole endeavor was to seek an way of representing proofs by logical relation that aligns well with those themes. Below, we argue that our approach indeed fits the bill in as much detail as the scope of this paper allows. We leave giving details and a full evaluation to future work.

[5] Moreover, the problem vanishes when extending LF with a cumulative hierarchy of universes [Mül19, Sect. 6.4] as is implemented in the MMT system.

Declarativity of MMT-logrels is clear since it is inherited from views. To evaluate modularity, we can consider the existing modularity features of MMT (defined constants, includes, diagrams of theories and views) and verify how well these are preserved by our approach. We neither motivate nor define what it means to preserve those features (we refer to [RR20]); instead we argue informally below:

- **definition preservation:** if T contains a defined constant $c\colon A\ [= t]$, then the corresponding constant $c_r\colon r(A)\ c\ [= r(t)]$ in T_r also possesses a definiens.
- **include preservation:** if S is included in T, then S_r is included in T_r. Assuming a fixed semantic domain X, this means that logical relations over S (i.e. views $S_r \to X$) can be likewise included in logical relations over T (i.e. views $T_r \to X$). In other words, given a modular hierarchy of theories structured by includes, we can build a modular hierarchy of logical relations over those theories, too.
- **diagram preservation (functoriality):** similar to the way we translate theories T to T_r, we can also translate views $v\colon S \to T$ to views $v_r\colon S_r \to T_r$ by mapping assignments $c = t$ to $c_r = r(t)$. It is clear that identity views are translated to identity views. We conjecture that we also have $(v \circ w)_r = v_r \circ w_r$ making the overall translation a functor in some suitable category.

Thus, it seems that modularity-wise our approach is promising. Future work might consider the translation sketched above in the theoretical framework of MMT metaprogramming facilities of [RR20]. (Implementation-wise it has already been implemented as part of [RR21].)

Can our approach handle real-world proofs by logical relation? So far we have only applied our representation method to toy problems. We refer to [RS13] for a discussion of how MMT-logrels applied to selected type theories (represented by MMT-theories) yield indeed the notions that one would expect. We also refer to [RR21] in which Florian Rabe and the author observed a use case that necessitates *partial* MMT-logrels.

5 Conclusion

We have motivated logical relations by attempting to prove strong normalization of the simply-typed lambda calculus ($\Gamma \vdash t\colon T \implies t \in \mathcal{SN}$) with a naive induction on typing derivations. This proof attempt got stuck in a case due to induction hypotheses being too weak. The central insight to remedy the stuckness was to strengthen the asserted claim from $t \in \mathcal{SN}$ to some claim $P_T(t)$ until the induction went through and we could recover strong normalization as a corollary. Here, P was a logical relation: a type-indexed family of unary relations $P_T(\cdot)$ on terms of type T. In general, proofs by logical relation proceed by *i*) defining a logical relation, *ii*) proving the Basic Lemma (if a term is well-typed, then it is in the relation at its type), and *iii*) recovering some desired theorem as a corollary.

Our focus was on idiomatically modeling this pattern in MMT/LF, a module system over the Edinburgh Logical Framework. This combination allows representing a wide variety of formal systems in a declarative and modular way by means of theories and views (theory translations). Central to modeling was the viewpoint of theories as interfaces and views $S \to T$ as implementations of S. For a proof by logical relation over T, we defined an interface theory T_r that via constants encoded *i*) stating a logical relation and *ii*) proving the Basic Lemma's individual cases. Proofs by logical relation then emerged as views out of T_r. Our encoding is declarative in nature and also modular: if $S \hookrightarrow T$ (T is an extension

of S), then $S_r \hookrightarrow T_r$. This means that proofs by logical relation over T can build on proofs by logical relation over S.

Further Pointers We have narrated at length only a special case of [RS13], of which we consider two unmentioned elaborations primarily important. First, the authors generalize logical relations over a theory S to logical relations along lists of views $v_1, \dots, v_n \colon S \to T$. Our special case only allows to prove claims universally quantified over S-terms s, while this generalization enables theorems about $v_1(s), \dots, v_n(s)$, i.e. theorems which relate results of applying an arbitrary, but fixed number of views. Second, the authors present many examples instantiating their notions with theories and views that are more realistic than our toy examples.

Future Work It is an open question how we can best expose the Basic Lemma within the formalization itself (i.e. to model step *iii* from above). This question has not only a technical dimension (when implementing in the MMT system), but also a theoretical one: access to the Basic Lemma implies some kind of reflection whose details can be subtle.

Moreover, we only sketched how the mapping of theories T to theories T_r is actually extensible to a functor. It may very well be amenable to a concise definition when phrased in the framework of structure-preserving diagram operators [RR20] on theories and views.

A Proof of Strong Normalization

Recall the homomorphic extension of logical relations to substitutions and contexts that we defined:

$$P_\Gamma(\gamma) := \forall\,(x\colon T) \in \Gamma.\ P_T(x\gamma)$$

▶ **Theorem/Formulation 3.** $\Gamma \vdash t\colon T$ *and* $P_\Gamma(\gamma) \implies P_T(t\gamma)$ *for all substitutions with* $\vdash \gamma\colon \Gamma$ *and where*

$$P_B(b) := b \in \mathcal{SN}$$
$$P_{T_1 \to T_2}(s) := s \in \mathcal{SN} \text{ and } (\forall t.\ P_{T_1}(t) \implies P_{T_2}(s\ t))$$

Proof. We induct on typing derivations as before.

- case TP-APP: we have $t = s_1\ s_2$ and the judgement

$$\frac{\Gamma \vdash s_1\colon T_1 \to T_2 \qquad \Gamma \vdash s_2\colon T_1}{\Gamma \vdash s_1\ s_2\colon T_2}\ (\text{TP-APP})$$

  We need to show $P_{T_2}((s_1\ s_2)\gamma)$. By definition of substitution application, this is equivalent to $P_{T_2}(s_1\gamma\ s_2\gamma)$.
  The induction hypothesis for the judgement on s_2 (instantiated with γ) yields $P_{T_1}(s_2\gamma)$. The induction hypothesis for the judgement on s_1 (instantiated with γ and the just achieved $P_{T_1}(s_2\gamma)$ yields $P_{T_1}(s_2\gamma)$, which is what we needed to show.
- case TP-VAR: necessarily $t = x$ for some variable x and $(x\colon T) \in \Gamma$. We are done by using the assumption of $P_\Gamma(\gamma)$ for x, which yields $P_T(x\gamma)$.
- case TP-ABS: we have $t = \lambda x.\ s$ and the judgement

$$\frac{\Gamma, x\colon T_1 \vdash s\colon T_2}{\Gamma \vdash \lambda x.\ s\colon T_1 \to T_2}\ (\text{TP-ABS})$$

  We need to show $P_{T_1 \to T_2}((\lambda x.\ s)\gamma)$. By the usual definition of substitution application (which we omitted in this paper), this is equivalent to $P_{T_1 \to T_2}(\lambda x\sigma.\ s\sigma\gamma)$ for some capture-avoiding (renaming) substitution σ. This proof goal amounts to showing two things.
  First, we need to show $(\lambda x\sigma.\ s\sigma\gamma) \in \mathcal{SN}$. It is sufficient to show strong normalization of the lambda's body. This follows from the induction hypothesis on the typing judgement on s instantiated with the substitution $\sigma\gamma$.
  Second, let r be a term with $P_{T_1}(r)$, then we need to show $P_{T_2}((\lambda x\sigma.\ s\sigma\gamma)\ r)$. By Lemma 20 below, it is sufficient to show $P_{T_2}(s\sigma\gamma[x\sigma \mapsto r])$ instead. The latter follows from the induction hypothesis on the typing judgement on s instantiated with the substitution $\sigma\gamma[x\sigma \mapsto r]$.

◀

▶ **Lemma 20** (Logical Relation closed under β-expansion)**.** *Let P be the logical relation from Formulation 3. For all types T and terms s and t with $P_T(s)$ and $P_T(t)$ we have*

$$P_T(s[x \mapsto t]) \implies P_T((\lambda x.\ s)\ t)$$

Proof. Left as an exercise to the reader. ◀

▶ **Lemma 21.** *Let P be the logical relation from Formulation 3. Then for all contexts Γ it holds $P_\Gamma(\gamma)$ where id_Γ is the identity substitution on Γ.*

Proof. Analgously to the case of TP-VAR in our proof attempt of Formulation 2. ◀

1	$p \vee \neg p$	
2	$\quad p$	
3	$\quad \neg\neg p$	$\neg_{\mathtt{I}}$, 2
4	$\quad \neg p \vee \neg\neg p$	$\vee_{\mathtt{IR}}$, 3
5	$\quad \neg p$	
6	$\quad \neg p \vee \neg\neg p$	$\vee_{\mathtt{IL}}$, 5
7	$\neg p \vee \neg\neg p$	$\vee_{\mathtt{E}}$, 1, 2–4, 5–6

Figure 3 Tertium Non Datur: proof for the induction case of $\neg$

1	$p \Leftrightarrow \neg\neg p$	
2	$\quad \neg p$	
3	$\quad \neg\neg(\neg p)$	$\neg_{\mathtt{I}}$, 2
4	$\quad \neg\neg(\neg p)$	
5	$\quad \neg(\neg\neg p)$	reit., 4
6	$\quad\quad p$	
7	$\quad\quad \neg\neg p$	$\Leftrightarrow_{\mathtt{ER}}$, 1, 6
8	$\quad\quad \bot$	$\bot_{\mathtt{I}}$, 7, 5
9	$\quad \neg p$	
10	$(\neg p) \Leftrightarrow \neg\neg(\neg p)$	$\Leftrightarrow_{\mathtt{I}}$, 2–3, 4–9

Figure 4 Double Negation Elimination: proof for the induction case of $\neg$

B Proofs of Meta Theorems over Prop. Logic

▶ **Theorem 10** (Tertium Non Datur). *If "a or not a" for all atoms a, then "p or not p" for all propositions p.*

▶ **Theorem 11** (Double Negation Elimination). *If "a iff. not not a" for all atoms a, then "p iff. not not p" for all propositions p.*

Proofs of Theorems 10 and 11. See Figures 3 and 5 and Figures 4 and 6, respectively. ◀

1	$p \vee \neg p$	
2	$q \vee \neg q$	
3	$\quad p$	
4	$\quad\quad q$	
5	$\quad\quad p \wedge q$	$\wedge_{\mathtt{I}}$, 3, 4
6	$\quad\quad (p \wedge q) \vee \neg(p \wedge q)$	$\vee_{\mathtt{IL}}$, 5
7	$\quad\quad \neg q$	
8	$\quad\quad\quad p \wedge q$	
9	$\quad\quad\quad q$	$\wedge_{\mathtt{ER}}$, 8
10	$\quad\quad\quad \bot$	$\bot_{\mathtt{I}}$, 9, 7
11	$\quad\quad \neg(p \wedge q)$	$\neg_{\mathtt{I}}$, 8–10
12	$\quad\quad (p \wedge q) \vee \neg(p \wedge q)$	$\vee_{\mathtt{IR}}$, 8
13	$\quad (p \wedge q) \vee \neg(p \wedge q)$	$\vee_{\mathtt{E}}$, 2, 4–6, 7–9
14	$\quad \neg p$	
15	$\quad p \wedge q$	
16	$\quad p$	$\wedge_{\mathtt{EL}}$, 8
17	$\quad \bot$	$\bot_{\mathtt{I}}$, 16, 14
18	$\neg(p \wedge q)$	$\neg_{\mathtt{I}}$, 15–17
19	$(p \wedge q) \vee \neg(p \wedge q)$	$\vee_{\mathtt{IR}}$, 12
20	$(p \wedge q) \vee \neg(p \wedge q)$	$\vee_{\mathtt{E}}$, 1, 3–10, 11–13

Figure 5 Tertium Non Datur: proof for the induction case of $\wedge$

1	$p \Leftrightarrow \neg\neg p$	
2	$q \Leftrightarrow \neg\neg q$	
3	$\quad p \wedge q$	
4	$\quad \neg\neg(p \wedge q)$	$\neg\neg_{\mathtt{I}}$, 3
5	$\quad \neg\neg(p \wedge q)$	
6	$\quad\quad \neg p$	
7	$\quad\quad\quad p \wedge q$	
8	$\quad\quad\quad p$	$\wedge_{\mathtt{EL}}$, 7
9	$\quad\quad\quad \bot$	$\bot_{\mathtt{I}}$, 8, 6
10	$\quad\quad \neg(p \wedge q)$	$\neg_{\mathtt{I}}$, 7–9
11	$\quad\quad \bot$	$\bot_{\mathtt{I}}$, 10, 5
12	$\quad\quad\quad \neg\neg p$	$\neg_{\mathtt{I}}$, 6–11
13	$\quad\quad\quad p$	$\Leftrightarrow_{\mathtt{EL}}$, 1, 12
14	$\quad\quad\quad\quad \neg q$	
15	$\quad\quad\quad\quad\quad p \wedge q$	
16	$\quad\quad\quad\quad\quad q$	$\wedge_{\mathtt{ER}}$, 15
17	$\quad\quad\quad\quad\quad \bot$	$\bot_{\mathtt{I}}$, 16, 14
18	$\quad\quad\quad\quad \neg(p \wedge q)$	$\neg_{\mathtt{I}}$, 15–17
19	$\quad\quad\quad\quad \bot$	$\bot_{\mathtt{I}}$, 18, 5
20	$\quad\quad\quad \neg\neg q$	$\neg_{\mathtt{I}}$, 14–19
21	$\quad\quad\quad q$	$\Leftrightarrow_{\mathtt{EL}}$, 2, 20
22	$\quad\quad\quad p \wedge q$	$\wedge_{\mathtt{I}}$, 13, 21
23	$\quad\quad (p \wedge q) \Leftrightarrow \neg\neg(p \wedge q)$	$\Leftrightarrow_{\mathtt{I}}$, 3–4, 5–22

Figure 6 Double Negation Elimination: proof for the induction case of $\wedge$

References

[Ahm13] Amal Ahmed. "Logical Relations". Oregon Programming Languages Summer School 2013. July 2013. URL: https://www.cs.uoregon.edu/research/summerschool/summer13/curriculum.html.

[BJP12] J. Bernardy, P. Jansson, and R. Paterson. "Proofs for Free - Parametricity for Dependent Types". In: *Journal of Functional Programming* 22.2 (2012), pp. 107–152.

[Boi04] Olivier Boite. "Proof Reuse with Extended Inductive Types". In: *Theorem Proving in Higher Order Logics*. Ed. by Konrad Slind, Annette Bunker, and Ganesh Gopalakrishnan. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 50–65. ISBN: 978-3-540-30142-4.

[Cod+] Mihai Codescu et al. "Project Abstract: Logic Atlas and Integrator (LATIN)". In: pp. 289–291. URL: https://kwarc.info/people/frabe/Research/CHKMR_latinabs_11.pdf.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. "A framework for defining logics". In: *Journal of the Association for Computing Machinery* 40.1 (1993), pp. 143–184.

[HRR14] Claudio Hermida, Uday S. Reddy, and Edmund P. Robinson. "Logical Relations and Parametricity A Reynolds Programme for Category Theory and Programming Languages". In: *Electronic Notes in Theoretical Computer Science* 303 (2014). Proceedings of the Workshop on Algebra, Coalgebra and Topology (WACT 2013), pp. 149–180. ISSN: 1571-0661. DOI: https://doi.org/10.1016/j.entcs.2014.02.008.

[Mit86] John C. Mitchell. "Representation Independence and Data Abstraction". In: *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '86. St. Petersburg Beach, Florida: Association for Computing Machinery, 1986, pp. 263–276. ISBN: 9781450373470. DOI: 10.1145/512644.512669.

[MMT] *MMT – Language and System for the Uniform Representation of Knowledge*. Project web site. URL: https://uniformal.github.io/ (visited on 01/15/2019).

[MR19] Dennis Müller and Florian Rabe. "Rapid Prototyping Formal Systems in MMT: 5 Case Studies". In: *LFMTP 2019*. Electronic Proceedings in Theoretical Computer Science (EPTCS), 2019. URL: https://kwarc.info/people/frabe/Research/MR_prototyping_19.pdf.

[Mül19] Dennis Müller. "Mathematical Knowledge Management Across Formal Libraries". PhD thesis. Informatics, FAU Erlangen-Nürnberg, Dec. 2019. URL: https://opus4.kobv.de/opus4-fau/files/12359/thesis.pdf.

[Pfe01] Frank Pfenning. "Chapter 17 - Logical Frameworks". In: *Handbook of Automated Reasoning*. Ed. by Alan Robinson and Andrei Voronkov. Handbook of Automated Reasoning. Amsterdam: North-Holland, 2001, pp. 1063–1147. ISBN: 978-0-444-50813-3. DOI: https://doi.org/10.1016/B978-044450813-3/50019-9.

[Pfe18] Frank Pfenning. "Lectures Notes on Parametricity. Lecture 12". Course "Types and Programming Languages" 2018. Nov. 2018. URL: https://www.cs.cmu.edu/~fp/courses/15814-f18/lectures/12-parametricity.pdf.

[PS] Frank Pfenning and Carsten Schürmann. "System Description: Twelf — A Meta-Logical Framework for Deductive Systems". In: *Proceedings of the 16th Conference on Automated Deduction*, pp. 202–206.

[Rab17] Florian Rabe. "How to Identify, Translate, and Combine Logics?" In: *Journal of Logic and Computation* 27.6 (2017), pp. 1753–1798.

[RG18] V. Rajani and D. Garg. "Types for Information Flow Control: Labeling Granularity and Semantic Models". In: *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. 2018, pp. 233–246. DOI: 10.1109/CSF.2018.00024.

[RK13] Florian Rabe and Michael Kohlhase. "A Scalable Module System". In: *Information & Computation* 0.230 (2013), pp. 1–54. URL: https://kwarc.info/frabe/Research/mmt.pdf.

[Rou19] Navid Roux. "Refactoring of Theory Graphs in Knowledge Representation Systems". B.Sc. Thesis. FAU Erlangen-Nürnberg, July 1, 2019. URL: https://gl.kwarc.info/supervision/BSc-archive/blob/master/2019/Roux_Navid.pdf.

[RR20] Navid Roux and Florian Rabe. "Structure-Preserving Diagram Operators". In: *Recent Trends in Algebraic Development Techniques*. Ed. by A. Corradini et al. to appear. Springer, 2020. URL: https://kwarc.info/people/frabe/Research/RR_diagops_20.pdf.

[RR21] Florian Rabe and Navid Roux. "Modular Formalization of Formal Systems". under review. 2021. URL: https://kwarc.info/people/frabe/Research/RR_modlog_21.pdf.

[RS13] Florian Rabe and Kristina Sojakova. "Logical Relations for a Logical Framework". In: *ACM Transactions on Computational Logic* (2013). URL: https://kwarc.info/frabe/Research/RS_logrels_12.pdf.

[Sko19] Lau Skorstengaard. *An Introduction to Logical Relations*. 2019.

[Soz+19] Matthieu Sozeau et al. "Coq Coq Correct! Verification of Type Checking and Erasure for Coq, in Coq". In: *Proc. ACM Program. Lang.* 4.POPL (Dec. 2019). DOI: 10.1145/3371076.

[UBN07] Christian Urban, Stefan Berghofer, and Michael Norrish. "Barendregt's Variable Convention in Rule Inductions". In: *Automated Deduction – CADE-21*. Ed. by Frank Pfenning. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 35–50. ISBN: 978-3-540-73595-3.

[Wad89] Philip Wadler. "Theorems for Free!" In: *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*. FPCA '89. Imperial College, London, United Kingdom: Association for Computing Machinery, 1989, pp. 347–359. ISBN: 0897913280. DOI: 10.1145/99370.99404.

[Wik19] Wikipedia contributors. *Logical framework — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Logical_framework&oldid=913088895. [Online; accessed 12-March-2021]. 2019.

[Zil12] Beta Ziliani. "Strong Normalization for Simply Typed Lambda Calculus". based on lectures by Derek Dreyer. July 2012. URL: https://web.archive.org/web/20170829154544/https://people.mpi-sws.org/~dg/teaching/pt2012/sn.pdf (visited on 01/26/2021).